

# PSE51-docs

---

## Documentation

Version latest

| English

| December 08, 2025

[kurisaw.eu.org](http://kurisaw.eu.org)

2025, KurisaW

# Contents

- 1. overview .....
- 1.1. abort — generate an abnormal process abort .....
- 1.2. abs .....
- 1.3. alarm .....
- 1.4. asctime — convert date and time to a string .....
- 1.5. asctime\_r — convert date and time to a string (REMOVED) .....
- 1.6. atof .....
- 1.7. atoi — convert a string to an integer .....
- 1.8. atol, atoll — convert a string to a long integer .....
- 1.9. atoll .....
- 1.10. bsearch — binary search a sorted table .....
- 1.11. calloc — a memory allocator .....
- 1.12. clearerr .....
- 1.13. clock\_getres, clock\_gettime, clock\_settime — clock and timer functions .....
- 1.14. clock\_getres, clock\_gettime, clock\_settime .....
- 1.15. clock\_nanosleep .....
- 1.16. clock\_settime .....
- 1.17. close, posix\_close .....
- 1.18. confstr — get configurable variables .....
- 1.19. ctime — convert a time value to a date and time string .....
- 1.20. ctime\_r — convert a time value to a date and time string (REMOVED) .....
- 1.21. difftime — compute the difference between two calendar time values .....
- 1.22. div .....
- 1.23. exit .....
- 1.24. fclose .....
- 1.25. fdatasync .....
- 1.26. fdopen .....
- 1.27. feclearexcept — clear floating-point exception .....

- 1.28. `fegetenv`, `fesetenv` — get and set current floating-point environment .....
- 1.29. `fegetexceptflag`, `fesetexceptflag` — get and set floating-point status flags .....
- 1.30. `fegetround`, `fesetround` — get and set current rounding direction .....
- 1.31. `feholdexcept` .....
- 1.32. `feof` .....
- 1.33. `feraiseexcept` — raise floating-point exception .....
- 1.34. `ferror` — test error indicator on a stream .....
- 1.35. `fegetenv`, `fesetenv` — get and set current floating-point environment .....
- 1.36. `fegetexceptflag`, `fesetexceptflag` — get and set floating-point status flags .....
- 1.37. `fegetround`, `fesetround` — get and set current rounding direction .....
- 1.38. `fetestexcept` — test floating-point exception flags .....
- 1.39. `feupdateenv` — update floating-point environment .....
- 1.40. `fflush` — flush a stream .....
- 1.41. `fgetc` — get a byte from a stream .....
- 1.42. `fgets` .....
- 1.43. `fileno` — map a stream pointer to a file descriptor .....
- 1.44. `flockfile`, `ftrylockfile`, `funlockfile` — stdio locking functions .....
- 1.45. `fopen` — open a stream .....
- 1.46. `fprintf` .....
- 1.47. `fputc` — put a byte on a stream .....
- 1.48. `fputs` .....
- 1.49. `fread` .....
- 1.50. `free` — free allocated memory .....
- 1.51. `freopen` .....
- 1.52. `fscanf` .....
- 1.53. `fsync` .....
- 1.54. `ftrylockfile` .....
- 1.55. `flockfile`, `ftrylockfile`, `funlockfile` — stdio locking functions .....
- 1.56. `fwrite` — binary output .....
- 1.57. `getc` — get a byte from a stream .....
- 1.58. `getc_unlocked`, `getchar_unlocked`, `putc_unlocked`, `putchar_unlocked` — st... .....
- 1.59. `getchar` .....
- 1.60. `getc_unlocked`, `getchar_unlocked`, `putc_unlocked`, `putchar_unlocked` — st... .....
- 1.61. `getenv`, `secure_getenv` — get value of an environment variable .....
- 1.62. `gets` .....

- 1.63. `gmtime`, `gmtime_r` — convert a time value to a broken-down UTC time .....
- 1.64. `gmtime`, `gmtime_r` — convert a time value to a broken-down UTC time .....
- 1.65. `imaxabs` — return absolute value .....
- 1.66. `imaxdiv` .....
- 1.67. `isalnum`, `isalnum_l` — test for an alphanumeric character .....
- 1.68. `isalpha`, `isalpha_l` — test for an alphabetic character .....
- 1.69. `isblank`, `isblank_l` — test for a blank character .....
- 1.70. `iscntrl`, `iscntrl_l` - test for a control character .....
- 1.71. `isdigit`, `isdigit_l` — test for a decimal digit .....
- 1.72. `isgraph`, `isgraph_l` — test for a visible character .....
- 1.73. `islower`, `islower_l` - test for a lowercase letter .....
- 1.74. `isprint`, `isprint_l` — test for a printable character .....
- 1.75. `ispunct`, `ispunct_l` — test for a punctuation character .....
- 1.76. `isspace`, `isspace_l` — test for a white-space character .....
- 1.77. `isupper`, `isupper_l` — test for an uppercase letter .....
- 1.78. `isxdigit`, `isxdigit_l` — test for a hexadecimal digit .....
- 1.79. `kill` .....
- 1.80. `labs`, `llabs` — return a long integer absolute value .....
- 1.81. `ldiv`, `lldiv` — compute quotient and remainder of a long division .....
- 1.82. `llabs` .....
- 1.83. `lldiv` .....
- 1.84. `localeconv` — return locale-specific information .....
- 1.85. `localtime`, `localtime_r` — convert a time value to a broken-down local time .....
- 1.86. `localtime`, `localtime_r` - convert a time value to a broken-down local time .....
- 1.87. `longjmp` — non-local goto .....
- 1.88. `malloc` — a memory allocator .....
- 1.89. `memchr` .....
- 1.90. `memcmp` — compare bytes in memory .....
- 1.91. `memcpy` — copy bytes in memory .....
- 1.92. `memmove` .....
- 1.93. `memset` .....
- 1.94. `mktime` — convert broken-down time into time since the Epoch .....
- 1.95. `mlock`, `munlock` — lock or unlock a range of process address space (REA... .....
- 1.96. `mlockall` .....
- 1.97. `mmap` .....

- 1.98. [munlock](#) .....
- 1.99. [munmap](#) — unmap pages of memory .....
- 1.100. [nanosleep](#) — high resolution sleep .....
- 1.101. [open](#), [openat](#) — open file .....
- 1.102. [pause](#) .....
- 1.103.  [perror](#) .....
- 1.104. [printf](#) .....
- 1.105. [pthread\\_atfork](#) .....
- 1.106. [pthread\\_attr\\_destroy](#) .....
- 1.107. [pthread\\_attr\\_getdetachstate](#), [pthread\\_attr\\_setdetachstate](#) — get and set... .....
- 1.108. [pthread\\_attr\\_getguardsize](#), [pthread\\_attr\\_setguardsize](#) — get and set the... .....
- 1.109. [pthread\\_attr\\_getinheritsched](#), [pthread\\_attr\\_setinheritsched](#) — get and s... .....
- 1.110. [pthread\\_attr\\_getschedparam](#), [pthread\\_attr\\_setschedparam](#) .....
- 1.111. [pthread\\_attr\\_getschedpolicy](#), [pthread\\_attr\\_setschedpolicy](#) .....
- 1.112. [pthread\\_attr\\_getscope](#), [pthread\\_attr\\_setscope](#) .....
- 1.113. [pthread\\_attr\\_getstack](#), [pthread\\_attr\\_setstack](#) .....
- 1.114. [pthread\\_attr\\_getstackaddr](#), [pthread\\_attr\\_setstackaddr](#) .....
- 1.115. [pthread\\_attr\\_getstacksize](#), [pthread\\_attr\\_setstacksize](#) — get and set the ... .....
- 1.116. [pthread\\_attr\\_init](#), [pthread\\_attr\\_destroy](#) - initialize and destroy thread attri... .....
- 1.117. [pthread\\_attr\\_getdetachstate](#), [pthread\\_attr\\_setdetachstate](#) — get and set... .....
- 1.118. [pthread\\_attr\\_getguardsize](#) .....
- 1.119. [pthread\\_attr\\_getinheritsched](#), [pthread\\_attr\\_setinheritsched](#) — get and se... .....
- 1.120. [pthread\\_attr\\_getschedparam](#), [pthread\\_attr\\_setschedparam](#) — get and s... .....
- 1.121. [pthread\\_attr\\_getschedpolicy](#) .....
- 1.122. [pthread\\_attr\\_getscope](#), [pthread\\_attr\\_setscope](#) — get and set the conten... .....
- 1.123. [pthread\\_attr\\_getstack](#), [pthread\\_attr\\_setstack](#) — get and set stack attribu... .....
- 1.124. [pthread\\_attr\\_getstackaddr](#), [pthread\\_attr\\_setstackaddr](#) .....
- 1.125. [pthread\\_attr\\_getstacksize](#), [pthread\\_attr\\_setstacksize](#) — get and set the ... .....
- 1.126. [pthread\\_cancel](#) .....
- 1.127. [pthread\\_cleanup\\_pop](#), [pthread\\_cleanup\\_push](#) .....
- 1.128. [pthread\\_cleanup\\_push](#), [pthread\\_cleanup\\_pop](#) .....
- 1.129. [pthread\\_cond\\_broadcast](#), [pthread\\_cond\\_signal](#) — broadcast or signal a ... .....
- 1.130. [pthread\\_cond\\_destroy](#), [pthread\\_cond\\_init](#) .....
- 1.131. [pthread\\_cond\\_destroy](#), [pthread\\_cond\\_init](#) — destroy and initialize condit... .....
- 1.132. [pthread\\_cond\\_broadcast](#), [pthread\\_cond\\_signal](#) — broadcast or signal a ... .....

- 1.133. `pthread_cond_clockwait`, `pthread_cond_timedwait`, `pthread_cond_wait` ...
- 1.134. `pthread_cond_clockwait`, `pthread_cond_timedwait`, `pthread_cond_wait` ...
- 1.135. `pthread_condattr_destroy`, `pthread_condattr_init` .....
- 1.136. `pthread_condattr_getclock`, `pthread_condattr_setclock` .....
- 1.137. `pthread_condattr_destroy`, `pthread_condattr_init` .....
- 1.138. `pthread_condattr_getclock`, `pthread_condattr_setclock` .....
- 1.139. `pthread_create` - thread creation .....
- 1.140. `pthread_detach` — detach a thread .....
- 1.141. `pthread_equal` — compare thread IDs .....
- 1.142. `pthread_exit` — thread termination .....
- 1.143. `pthread_getconcurrency` .....
- 1.144. `pthread_getcpu` .....
- 1.145. `pthread_getschedparam`, `pthread_setschedparam` — dynamic thread sc...
- 1.146. `pthread_getspecific`, `pthread_setspecific` — thread-specific data manage...
- 1.147. `pthread_join` — wait for thread termination .....
- 1.148. `pthread_key_create` — thread-specific data key creation .....
- 1.149. `pthread_key_delete` .....
- 1.150. `pthread_kill` .....
- 1.151. `pthread_mutex_destroy` .....
- 1.152. `pthread_mutex_getprioceiling`, `pthread_mutex_setprioceiling` .....
- 1.153. `pthread_mutex_destroy`, `pthread_mutex_init` .....
- 1.154. `pthread_mutex_lock` .....
- 1.155. `pthread_mutex_getprioceiling`, `pthread_mutex_setprioceiling` .....
- 1.156. `pthread_mutex_trylock` - 锁定和解锁互斥锁 .....
- 1.157. `pthread_mutex_lock`, `pthread_mutex_trylock`, `pthread_mutex_unlock` .....
- 1.158. `pthread_mutexattr_destroy` .....
- 1.159. `pthread_mutexattr_getprioceiling`, `pthread_mutexattr_setprioceiling` .....
- 1.160. `pthread_mutexattr_getprotocol` .....
- 1.161. `pthread_mutexattr_gettype`, `pthread_mutexattr_settype` — get and set th...
- 1.162. `pthread_mutexattr_init`, `pthread_mutexattr_destroy` - destroy and initializ...
- 1.163. `pthread_mutexattr_getprioceiling`, `pthread_mutexattr_setprioceiling` .....
- 1.164. `pthread_mutexattr_setprotocol`, `pthread_mutexattr_getprotocol` .....
- 1.165. `pthread_mutexattr_gettype`, `pthread_mutexattr_settype` - get and set the...
- 1.166. `pthread_once` — dynamic package initialization .....
- 1.167. `pthread_self` — get the calling thread ID .....

- 1.168. `pthread_setcancelstate`, `pthread_setcanceltype`, `pthread_testcancel` .....
- 1.169. `pthread_setcancelstate`, `pthread_setcanceltype`, `pthread_testcancel` — s...
- 1.170. `pthread_setconcurrency`, `pthread_getconcurrency` .....
- 1.171. `pthread_getschedparam`, `pthread_setschedparam` .....
- 1.172. `pthread_setschedprio` .....
- 1.173. `pthread_getspecific`, `pthread_setspecific` — thread-specific data manage...
- 1.174. `pthread_sigmask`, `sigprocmask` — examine and change blocked signals .....
- 1.175. `pthread_setcancelstate`, `pthread_setcanceltype`, `pthread_testcancel` .....
- 1.176. `putc` — put a byte on a stream .....
- 1.177. `putc_unlocked` .....
- 1.178. `putchar` .....
- 1.179. `putchar_unlocked` .....
- 1.180. `puts` .....
- 1.181. `qsort`, `qsort_r` — sort a table of data .....
- 1.182. `raise` .....
- 1.183. `rand`, `strrand` — pseudo-random number generator .....
- 1.184. `rand`, `rand_r`, `strrand` - pseudo-random number generator .....
- 1.185. `read` .....
- 1.186. `realloc`, `reallocarray` — memory reallocators .....
- 1.187. `scanf` .....
- 1.188. `sched_get_priority_max`, `sched_get_priority_min` .....
- 1.189. `sched_get_priority_max`, `sched_get_priority_min` .....
- 1.190. `sched_rr_get_interval` — get execution time limits (REALTIME) .....
- 1.191. `sem_close` — close a named semaphore .....
- 1.192. `sem_destroy` — destroy an unnamed semaphore .....
- 1.193. `sem_getvalue` .....
- 1.194. `sem_init` — initialize an unnamed semaphore .....
- 1.195. `sem_open` — initialize and open a named semaphore .....
- 1.196. `sem_post` — unlock a semaphore .....
- 1.197. `sem_clockwait`, `sem_timedwait` — lock a semaphore .....
- 1.198. `sem_trywait`, `sem_wait` — lock a semaphore .....
- 1.199. `sem_unlink` - remove a named semaphore .....
- 1.200. `sem_trywait`, `sem_wait` — lock a semaphore .....
- 1.201. `setbuf` .....
- 1.202. `setenv` — add or change environment variable .....

- 1.203. `setjmp` - set jump point for a non-local goto .....
- 1.204. `setlocale` — set program locale .....
- 1.205. `setvbuf` — assign buffering to a stream .....
- 1.206. `shm_open` — open a shared memory object (REALTIME) .....
- 1.207. `shm_unlink` — remove a shared memory object (REALTIME) .....
- 1.208. `sigaction` .....
- 1.209. `sigaddset` .....
- 1.210. `sigdelset` - delete a signal from a signal set .....
- 1.211. `sigemptyset` .....
- 1.212. `sigfillset` - initialize and fill a signal set .....
- 1.213. `sigismember` .....
- 1.214. `signal` — signal management .....
- 1.215. `sigpending` .....
- 1.216. `sigprocmask` .....
- 1.217. `sigqueue` .....
- 1.218. `sigsuspend` — wait for a signal .....
- 1.219. `sigtimedwait`, `sigwaitinfo` .....
- 1.220. `sigwait` — wait for queued signals .....
- 1.221. `sigtimedwait`, `sigwaitinfo` — wait for queued signals .....
- 1.222. `snprintf`, `asprintf`, `dprintf`, `fprintf`, `printf`, `sprintf` — print formatted output .....
- 1.223. `sprintf` - print formatted output .....
- 1.224. `rand`, `rand` — pseudo-random number generator .....
- 1.225. `fscanf`, `scanf`, `sscanf` .....
- 1.226. `strcat` .....
- 1.227. `strchr` .....
- 1.228. `strcmp` — compare two strings .....
- 1.229. `strcoll` .....
- 1.230. `strcpy` .....
- 1.231. `strcspn` .....
- 1.232. `strerror`, `strerror_l`, `strerror_r` — get error message string .....
- 1.233. `strerror`, `strerror_l`, `strerror_r` — get error message string .....
- 1.234. `strftime`, `strftime_l` — convert date and time to a string .....
- 1.235. `strlen`, `strnlen` — get length of fixed size string .....
- 1.236. `strncat` — concatenate a string with part of another .....
- 1.237. `strncmp` .....

- 1.238. `strncpy` .....
- 1.239. `strpbrk` .....
- 1.240. `strrchr` — string scanning operation .....
- 1.241. `strspn` — get length of a substring .....
- 1.242. `strstr` — find a substring .....
- 1.243. `strtod`, `strtof`, `strtold` — convert a string to a double-precision number .....
- 1.244. `strtod`, `strtof`, `strtold` - convert a string to a double-precision number .....
- 1.245. `strtoimax`, `strtoumax` — convert string to integer type .....
- 1.246. `strtok`, `strtok_r` — split string into tokens .....
- 1.247. `strtok`, `strtok_r` — split string into tokens .....
- 1.248. `strtol`, `strtoll` — convert a string to a long integer .....
- 1.249. `strtold` .....
- 1.250. `strtoll` .....
- 1.251. `strtoul` .....
- 1.252. `strtoull` — convert a string to an unsigned long long integer .....
- 1.253. `strtoumax` .....
- 1.254. `strxfrm`, `strxfrm_l` — string transformation .....
- 1.255. `sysconf` .....
- 1.256. `time` — get time .....
- 1.257. `timer_create` - create a per-process timer .....
- 1.258. `timer_delete` .....
- 1.259. `timer_getoverrun`, `timer_gettime`, `timer_settime` — per-process timers .....
- 1.260. `timer_getoverrun`, `timer_gettime`, `timer_settime` — per-process timers .....
- 1.261. `timer_getoverrun`, `timer_gettime`, `timer_settime` — per-process timers .....
- 1.262. `tolower`, `tolower_l` — transliterate uppercase characters to lowercase .....
- 1.263. `toupper`, `toupper_l` — transliterate lowercase characters to uppercase .....
- 1.264. `tzset` - set timezone conversion information .....
- 1.265. `uname` - get system name .....
- 1.266. `ungetc` — push byte back into input stream .....
- 1.267. `unsetenv` .....
- 1.268. `va_arg` .....
- 1.269. `va_copy` .....
- 1.270. `va_end` .....
- 1.271. `va_start`, `va_arg`, `va_copy`, `va_end` - handle variable argument list .....
- 1.272. `vfprintf` .....

- 1.273. vfscanf, vscanf, vsscanf — format input of a stdarg argument list .....
- 1.274. vprintf .....
- 1.275. vfscanf, vscanf, vsscanf — format input of a stdarg argument list .....
- 1.276. vsnprintf .....
- 1.277. vsprintf - Format Output of a stdarg Argument List .....

  - 1.277.1. Examples .....
  - 1.277.2. Application Usage .....
  - 1.277.3. Rationale .....
  - 1.277.4. Future Directions .....
  - 1.277.5. See Also .....
  - 1.277.6. Change History .....

- 1.278. vsscanf .....
- 1.279. write .....

# 1. overview

---

## 1.1. `abort` — generate an abnormal process abort

---

### SYNOPSIS

```
#include <stdlib.h>

_Noreturn void abort(void);
```

### DESCRIPTION

The `abort()` function shall cause abnormal process termination to occur, unless a SIGABRT signal that it generates is caught and the signal handler does not return.

The abnormal termination processing shall include the default actions defined for SIGABRT and may include an attempt to effect `fclose()` on all open streams.

The SIGABRT signal shall be sent to the calling thread as if by means of `raise()` with the argument SIGABRT. If this signal does not terminate the process (for example, if the signal is caught and the handler returns), `abort()` may change the disposition of SIGABRT to SIG\_DFL and send the signal (in the same way) again. If a second signal is sent and it does not terminate the process, the behavior is unspecified, except that the `abort()` call shall not return.

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

### RETURN VALUE

The `abort()` function shall not return.

# ERRORS

No errors are defined.

## EXAMPLES

None.

## APPLICATION USAGE

Catching the signal is intended to provide the application developer with a portable means to abort processing, free from possible interference from any implementation-supplied functions.

## RATIONALE

Historically, `abort()` has been implemented by calling other signal manipulation functions such as `raise()`, `sigaction()`, and `pthread_sigmask()`. This means that its operation can be affected by concurrent actions in other threads. For example, if `abort()` attempts to terminate the process by calling `sigaction()` to change the disposition for SIGABRT to SIG\_DFL and then calling `raise()`, another thread could change the disposition in between those two calls, resulting in the process not being terminated. If this happens, the only requirement is that `abort()` does not return. An implementation could call those functions in a loop (which could in theory then execute indefinitely), or could terminate the process by calling `_exit()` (which would ensure termination but result in the wrong wait status). To avoid these issues, implementations are encouraged to implement `abort()` in a manner such that its operation cannot be affected by concurrent actions in other threads. For example, it could first halt the execution of all other threads, or it could terminate the process using a "terminate as if by a signal" system call instead of by raising (a second) SIGABRT.

The ISO/IEC 9899:1999 standard required (and the current standard still requires) the `abort()` function to be async-signal-safe. Since POSIX.1-2024 defers to the ISO C standard, this required a change to the DESCRIPTION from "shall include the effect of `fclose()`" to "may include an attempt to effect `fclose()`."

The revised wording permits some backwards-compatibility and avoids a potential deadlock situation.

The Open Group Base Resolution bwg2002-003 is applied, removing the following XSI shaded paragraph from the DESCRIPTION:

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

There were several reasons to remove this paragraph:

- No special processing of open message catalogs needs to be performed prior to abnormal process termination.
- The main reason to specifically mention that `abort()` includes the effect of `fclose()` on open streams is to flush output queued on the stream. Message catalogs in this context are read-only and, therefore, do not need to be flushed.
- The effect of `fclose()` on a message catalog descriptor is unspecified. Message catalog descriptors are allowed, but not required to be implemented using a file descriptor, but there is no mention in POSIX.1-2024 of a message catalog descriptor using a standard I/O stream FILE object as would be expected by `fclose()`.

## FUTURE DIRECTIONS

A future version of this standard may require `abort()` to be implemented in a manner such that its operation cannot be affected by concurrent actions in other threads.

## SEE ALSO

`_exit()`, `kill()`, `raise()`, `signal()`, `wait()`, `waitid()`

XBD `<stdlib.h>`

## CHANGE HISTORY

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

### Issue 6

Extensions beyond the ISO C standard are marked.

Changes are made to the DESCRIPTION for alignment with the ISO/IEC 9899:1999 standard.

The Open Group Base Resolution bwg2002-003 is applied.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/10 is applied, changing the DESCRIPTION of abnormal termination processing and adding to the RATIONALE section.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/9 is applied, changing "implementation-defined functions" to "implementation-supplied functions" in the APPLICATION USAGE section.

## Issue 8

Austin Group Defect 906 is applied, clarifying how the behavior of `abort()` may be affected by concurrent actions in other threads.

Austin Group Defect 1302 is applied, aligning this function with the ISO/IEC 9899:2018 standard.

## 1.2. `abs`

---

### SYNOPSIS

```
#include <stdlib.h>

int abs(int i);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

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

### RETURN VALUE

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

### ERRORS

No errors are defined.

---

*The following sections are informative.*

### EXAMPLES

None.

### APPLICATION USAGE

Since POSIX.1 requires a two's complement representation of `int`, the absolute value of the negative integer with the largest magnitude `{INT_MIN}` is not

representable, thus `abs(INT_MIN)` is undefined.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `fabs()`
- `labs()`

XBD `<stdlib.h>`

## CHANGE HISTORY

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

### Issue 6

Extensions beyond the ISO C standard are marked.

### Issue 8

Austin Group Defect 1108 is applied, changing the APPLICATION USAGE section.

## 1.3. alarm

---

### SYNOPSIS

```
#include <unistd.h>

unsigned alarm(unsigned seconds);
```

### DESCRIPTION

The `alarm()` function shall cause the system to generate a SIGALRM signal for the process after the number of realtime seconds specified by `seconds` have elapsed. Processor scheduling delays may prevent the process from handling the signal as soon as it is generated.

If `seconds` is 0, a pending alarm request, if any, is canceled.

Alarm requests are not stacked; only one SIGALRM generation can be scheduled in this manner. If the SIGALRM signal has not yet been generated, the call shall result in rescheduling the time at which the SIGALRM signal is generated.

### RETURN VALUE

If there is a previous `alarm()` request with time remaining, `alarm()` shall return a non-zero value that is the number of seconds until the previous request would have generated a SIGALRM signal. Otherwise, `alarm()` shall return 0.

### ERRORS

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

---

*The following sections are informative.*

### EXAMPLES

None.

## APPLICATION USAGE

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

Application developers should note that the type of the argument `seconds` and the return value of `alarm()` is **unsigned**. That means that a Strictly Conforming POSIX System Interfaces Application cannot pass a value greater than the minimum guaranteed value for {UINT\_MAX}, which the ISO C standard sets as 65535, and any application passing a larger value is restricting its portability. A different type was considered, but historical implementations, including those with a 16-bit `int` type, consistently use either **unsigned** or `int`.

Application developers should be aware of possible interactions when the same process uses both the `alarm()` and `sleep()` functions.

## RATIONALE

Many historical implementations (including Version 7 and System V) allow an alarm to occur up to a second early. Other implementations allow alarms up to half a second or one clock tick early or do not allow them to occur early at all. The latter is considered most appropriate, since it gives the most predictable behavior, especially since the signal can always be delayed for an indefinite amount of time due to scheduling. Applications can thus choose the `seconds` argument as the minimum amount of time they wish to have elapse before the signal.

The term "realtime" here and elsewhere (`sleep()`, `times()`) is intended to mean "wall clock" time as common English usage, and has nothing to do with "realtime operating systems". It is in contrast to *virtual time*, which could be misinterpreted if just `time` were used.

In some implementations, including 4.3 BSD, very large values of the `seconds` argument are silently rounded down to an implementation-specific maximum value. This maximum is large enough (to the order of several months) that the effect is not noticeable.

There were two possible choices for alarm generation in multi-threaded applications: generation for the calling thread or generation for the process. The first option would not have been particularly useful since the alarm state is maintained on a per-process basis and the alarm that is established by the last invocation of `alarm()` is the only one that would be active.

Furthermore, allowing generation of an asynchronous signal for a thread would have introduced an exception to the overall signal model. This requires a

compelling reason in order to be justified.

## FUTURE DIRECTIONS

None.

## SEE ALSO

[alarm](#), [exec](#), [fork\(\)](#), [pause\(\)](#), [sigaction\(\)](#), [sleep\(\)](#), [timer\\_create\(\)](#)

XBD ,

## CHANGE HISTORY

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

### Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

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

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/16 is applied, replacing "an implementation-defined maximum value" with "an implementation-specific maximum value" in the RATIONALE.

### Issue 8

Austin Group Defect 1330 is applied, removing obsolescent interfaces.

## 1.4. `asctime` — convert date and time to a string

---

### SYNOPSIS

```
#include <time.h>

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

### DESCRIPTION

The `asctime()` function shall convert the broken-down time in the structure pointed to by `timeptr` into a string in the form:

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

using the equivalent of the following algorithm:

```
char *asctime(const struct tm *timeptr)
{
    static char wday_name[7][3] = {
        "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
    };
    static char mon_name[12][3] = {
        "Jan", "Feb", "Mar", "Apr", "May", "Jun",
        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
    };
    static char result[26];

    sprintf(result, "%3s %3s%3d %.2d:%.2d:%.2d %d\n",
            wday_name[timeptr->tm_wday],
            mon_name[timeptr->tm_mon],
            timeptr->tm_mday, timeptr->tm_hour,
            timeptr->tm_min, timeptr->tm_sec,
            1900 + timeptr->tm_year);
    return result;
}
```

If any of the members of the broken-down time contain values that are outside their normal ranges (see XBD `<time.h>`), the behavior of the `asctime()` function is undefined. Likewise, if the calculated year exceeds four digits or is less than the year 1000, the behavior is undefined.

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

The `asctime()`, `ctime()`, `gmtime()`, and `localtime()` functions shall return values in one of two static objects: a broken-down time structure and an array of type `char`. Execution of any of the functions that return a pointer to one of these object types may overwrite the information in any object of the same type pointed to by the value returned from any previous call to any of them.

The `asctime()` function need not be thread-safe; however, `asctime()` shall avoid data races with all functions other than itself, `ctime()`, `gmtime()`, and `localtime()`.

## RETURN VALUE

Upon successful completion, `asctime()` shall return a pointer to the string. If the function is unsuccessful, it shall return NULL.

## ERRORS

No errors are defined.

## APPLICATION USAGE

This function is included only for compatibility with older implementations. It has undefined behavior if the resulting string would be too long, so the use of this function should be discouraged. On implementations that do not detect output string length overflow, it is possible to overflow the output buffer in such a way as to cause applications to fail, or possible system security violations. Also, this function does not support localized date and time formats. To avoid these problems, applications should use `strftime()` to generate strings from broken-down times.

Values for the broken-down time structure can be obtained by calling `gmtime()` or `localtime()`.

## RATIONALE

The standard developers decided to mark the `asctime()` function obsolescent even though it is in the ISO C standard due to the possibility of buffer overflow. The ISO C standard also provides the `strftime()` function which can be used to avoid these problems.

## FUTURE DIRECTIONS

This function may be removed in a future version, but not until after it has been removed from the ISO C standard.

## SEE ALSO

- `clock()`
- `ctime()`
- `difftime()`
- `futimens()`
- `gmtime()`
- `localtime()`
- `mktime()`
- `strftime()`
- `strptime()`
- `time()`
- XBD `<time.h>`

## CHANGE HISTORY

### First released in Issue 1

Derived from Issue 1 of the SVID.

### Issue 5

- Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.
- The `asctime_r()` function is included for alignment with the POSIX Threads Extension.
- A note indicating that the `asctime()` function need not be reentrant is added to the DESCRIPTION.

## Issue 6

- The `asctime_r()` function is marked as part of the Thread-Safe Functions option.
- Extensions beyond the ISO C standard are marked.
- The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.
- The DESCRIPTION of `asctime_r()` is updated to describe the format of the string returned.
- The `restrict` keyword is added to the `asctime_r()` prototype for alignment with the ISO/IEC 9899:1999 standard.
- IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/17 is applied, adding the CX extension in the RETURN VALUE section requiring that if the `asctime()` function is unsuccessful it returns NULL.

## Issue 7

- Austin Group Interpretation 1003.1-2001 #053 is applied, marking these functions obsolescent.
- Austin Group Interpretation 1003.1-2001 #156 is applied.
- The `asctime_r()` function is moved from the Thread-Safe Functions option to the Base.
- POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0033 [86,429] is applied.

## Issue 8

- Austin Group Defect 469 is applied, clarifying the conditions under which the behavior of `asctime()` is undefined.
- Austin Group Defect 1302 is applied, aligning this function with the ISO/IEC 9899:2018 standard.
- Austin Group Defect 1330 is applied, changing the FUTURE DIRECTIONS section.
- Austin Group Defect 1376 is applied, removing CX shading from some text derived from the ISO C standard and updating it to match the ISO C standard.
- Austin Group Defect 1410 is applied, removing the `asctime_r()` function.

## 1.5. `asctime_r` — convert date and time to a string (REMOVED)

---

### Status

**REMOVED:** The `asctime_r()` function has been removed from the POSIX.1-2024 standard (Issue 8, Austin Group Defect 1410).

### Historical Information

This page documents the historical context of the `asctime_r()` function which was previously part of the POSIX standard but has been removed.

#### NAME (Historical)

`asctime_r` — convert date and time to a string (thread-safe)

#### SYNOPSIS (Historical)

```
#include <time.h>

char *asctime_r(const struct tm *restrict timeptr, char *restrict l
```

### DESCRIPTION

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `asctime_r()` function was the thread-safe version of `asctime()`, designed to convert the broken-down time in the structure pointed to by `timeptr` into a string in the form:

```
Sun Sep 16 01:03:52 1973\n\n0
```

## Key Differences from `asctime()`

- **Thread Safety:** `asctime_r()` was thread-safe, while `asctime()` was not
- **Buffer Management:** `asctime_r()` required the caller to provide a buffer
- **Static Data Avoidance:** `asctime_r()` avoided using static data areas

## Historical Context

### Evolution Through POSIX Versions

**Issue 5:** The `asctime_r()` function was included for alignment with the POSIX Threads Extension.

**Issue 6:**

- The `asctime_r()` function was marked as part of the Thread-Safe Functions option
- The `restrict` keyword was added to the `asctime_r()` prototype for alignment with the ISO/IEC 9899:1999 standard
- Extensions beyond the ISO C standard were marked
- The APPLICATION USAGE section was updated to include a note on the thread-safe function

**Issue 7:**

- The `asctime_r()` function was moved from the Thread-Safe Functions option to the Base
- Austin Group Interpretation 1003.1-2001 #053 was applied, marking these functions obsolescent

**Issue 8:**

- Austin Group Defect 1410 was applied, **removing the `asctime_r()` function** from the standard

## RATIONALE FOR REMOVAL

The standard developers decided to mark the `asctime()` function (and by extension `asctime_r()`) as obsolescent even though it is in the ISO C standard

due to the possibility of buffer overflow. The ISO C standard also provides the `strftime()` function which can be used to avoid these problems.

## APPLICATION USAGE (Historical)

This function was included only for compatibility with older implementations. It had undefined behavior if the resulting string would be too long, so the use of this function should be discouraged.

**Recommended Alternative:** Applications should use `strftime()` to generate strings from broken-down times to avoid buffer overflow problems and to support localized date and time formats.

Values for the broken-down time structure can be obtained by calling `gmtime()` or `localtime()`.

## FUTURE DIRECTIONS

The `asctime_r()` function has been removed and will not be available in future versions of the POSIX standard.

## SEE ALSO

- `asctime()` - the non-thread-safe version (also obsolescent)
- `clock()`
- `ctime()`
- `difftime()`
- `gmtime()`
- `localtime()`
- `mktime()`
- `strftime()` - **Recommended replacement**
- `strptime()`
- `time()`
- XBD `<time.h>`

# CHANGE HISTORY

**First released:** Issue 1. Derived from Issue 1 of the SVID.

## Issue 5:

- Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION
- The `asctime_r()` function is included for alignment with the POSIX Threads Extension
- A note indicating that the `asctime()` function need not be reentrant is added to the DESCRIPTION

## Issue 6:

- The `asctime_r()` function is marked as part of the Thread-Safe Functions option
- Extensions beyond the ISO C standard are marked
- The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area
- The DESCRIPTION of `asctime_r()` is updated to describe the format of the string returned
- The `restrict` keyword is added to the `asctime_r()` prototype for alignment with the ISO/IEC 9899:1999 standard
- IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/17 is applied, adding the CX extension in the RETURN VALUE section requiring that if the `asctime()` function is unsuccessful it returns NULL

## Issue 7:

- Austin Group Interpretation 1003.1-2001 #053 is applied, marking these functions obsolescent
- Austin Group Interpretation 1003.1-2001 #156 is applied
- The `asctime_r()` function is moved from the Thread-Safe Functions option to the Base
- POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0033 [86,429] is applied

## Issue 8:

- Austin Group Defect 469 is applied, clarifying the conditions under which the behavior of `asctime()` is undefined

- Austin Group Defect 1302 is applied, aligning this function with the ISO/IEC 9899:2018 standard
  - Austin Group Defect 1330 is applied, changing the FUTURE DIRECTIONS section
  - Austin Group Defect 1376 is applied, removing CX shading from some text derived from the ISO C standard and updating it to match the ISO C standard
  - **Austin Group Defect 1410 is applied, removing the `asctime_r()` function**
-

## 1.6. atof

---

[ CX ] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

### Synopsis

```
#include <stdlib.h>

double atof(const char *str);
```

### Description

The call `atof(str)` shall be equivalent to:

```
strtod(str, (char **)NULL),
```

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

### Return Value

The `atof()` function returns the converted value.

### Errors

No errors are defined.

### Examples

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

int main(void) {
    const char *str = "123.456";
    double result = atof(str);
```

```
    printf("Converted value: %f\n", result);
    return 0;
}
```

## Application Usage

The `atof()` function is not required to be thread-safe.

## Rationale

The following sections are informative.

## 1.7. atoi — convert a string to an integer

---

### SYNOPSIS

```
#include <stdlib.h>

int atoi(const char *str);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The call `atoi(str)` shall be equivalent to:

```
(int) strtol(str, (char **)NULL, 10)
```

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

### RETURN VALUE

The `atoi()` function shall return the converted value if the value can be represented.

### ERRORS

No errors are defined.

---

*The following sections are informative.*

# EXAMPLES

## Converting an Argument

The following example checks for proper usage of the program. If there is an argument and the decimal conversion of this argument (obtained using `atoi()`) is greater than 0, then the program has a valid number of minutes to wait for an event.

```
#include <stdlib.h>
#include <stdio.h>
...
int minutes_to_event;
...
if (argc < 2 || (minutes_to_event = atoi(argv[1])) <= 0) {
    fprintf(stderr, "Usage: %s minutes\n", argv[0]); exit(1);
}
...
```

# APPLICATION USAGE

The `atoi()` function is subsumed by `strtol()` but is retained because it is used extensively in existing code. If the number is not known to be in range, `strtol()` should be used because `atoi()` is not required to perform any error checking.

# RATIONALE

None.

# FUTURE DIRECTIONS

None.

# SEE ALSO

- `strtol()`
- `<stdlib.h>`

# CHANGE HISTORY

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

## Issue 8

Austin Group Defect 1541 is applied, changing the EXAMPLES section.

---

## 1.8. `atol`, `atoll` — convert a string to a long integer

---

### SYNOPSIS

```
#include <stdlib.h>

long atol(const char *nptr);
long long atoll(const char *nptr);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

Except as noted below, the call `atol(nptra)` shall be equivalent to:

```
strtol(nptra, (char **)NULL, 10)
```

Except as noted below, the call to `atoll(nptra)` shall be equivalent to:

```
strtoll(nptra, (char **)NULL, 10)
```

The handling of errors may differ. If the value cannot be represented, the behavior is undefined.

### RETURN VALUE

These functions shall return the converted value if the value can be represented.

### ERRORS

No errors are defined.

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

If the number is not known to be in range, `strtol()` or `strtoll()` should be used because `atol()` and `atoll()` are not required to perform any error checking.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `strtol()`
- `<stdlib.h>`

## CHANGE HISTORY

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

### Issue 6

The `atoll()` function is added for alignment with the ISO/IEC 9899:1999 standard.

### Issue 7

SD5-XSH-ERN-61 is applied, correcting the DESCRIPTION of `atoll()`.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0046 [892] is applied.

---

## 1.9. atoll

---

### SYNOPSIS

```
#include <stdlib.h>

long atol(const char *nptr);
long long atoll(const char *nptr);
```

### DESCRIPTION

[Option Start] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard. [Option End]

Except as noted below, the call `atol(nptr)` shall be equivalent to:

```
strtol(nptr, (char **)NULL, 10)
```

Except as noted below, the call to `atoll(nptr)` shall be equivalent to:

```
strtoll(nptr, (char **)NULL, 10)
```

The handling of errors may differ. If the value cannot be represented, the behavior is undefined.

### RETURN VALUE

These functions return the converted value.

### ERRORS

No errors are defined.

### EXAMPLES

None provided.

## APPLICATION USAGE

None provided.

## RATIONALE

None provided.

## FUTURE DIRECTIONS

None provided.

## SEE ALSO

[strtol\(\)](#) , [strtoll\(\)](#)

## CHANGE HISTORY

SD5-XSH-ERN-61 is applied, correcting the DESCRIPTION of [atoll\(\)](#).

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0046 [892] is applied.

## 1.10. bsearch — binary search a sorted table

---

### SYNOPSIS

```
#include <stdlib.h>

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



### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `bsearch()` function shall search an array of `nel` objects, the initial element of which is pointed to by `base`, for an element that matches the object pointed to by `key`. The size of each element in the array is specified by `width`. If the `nel` argument has the value zero, the comparison function pointed to by `compar` shall not be called and no match shall be found.

The comparison function pointed to by `compar` shall be called with two arguments that point to the `key` object and to an array element, in that order.

The application shall ensure that the comparison function pointed to by `compar` does not alter the contents of the array. The implementation may reorder elements of the array between calls to the comparison function, but shall not alter the contents of any individual element.

The implementation shall ensure that the first argument is always a pointer to the key.

When the same objects (consisting of `width` bytes, irrespective of their current positions in the array) are passed more than once to the comparison function, the results shall be consistent with one another. That is, the same object shall always compare the same way with the key.

The application shall ensure that the function returns an integer less than, equal to, or greater than 0 if the `key` object is considered, respectively, to be less than, to match, or to be greater than the array element. The application shall ensure that the array consists of all the elements that compare less than, all the elements that

compare equal to, and all the elements that compare greater than the `key` object, in that order.

## RETURN VALUE

The `bsearch()` function shall return a pointer to a matching member of the array, or a null pointer if no match is found. If two or more members compare equal, which member is returned is unspecified.

## ERRORS

No errors are defined.

---

*The following sections are informative.*

## EXAMPLES

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

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

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

#define TABSIZE    1000

struct node {                      /* These are stored in the table. */
    char *string;
    int length;
};

struct node table[TABSIZE];      /* Table to be searched. */

{
    struct node *node_ptr, node;
    /* Routine to compare 2 nodes. */
```

```

int node_compare(const void *, const void *);

.

.

while (scanf("%ms", &node.string) != EOF) {
    node_ptr = (struct node *)bsearch((void *)(&node),
        (void *)table, TABSIZE,
        sizeof(struct node), node_compare);
    if (node_ptr != NULL) {
        (void)printf("string = %20s, length = %d\n",
            node_ptr->string, node_ptr->length);
    } else {
        (void)printf("not found: %s\n", node.string);
    }
    free(node.string);
}
/*
    This routine compares two nodes based on an
    alphabetical ordering of the string field.
*/
int
node_compare(const void *node1, const void *node2)
{
    return strcoll((const struct node *)node1->string,
        ((const struct node *)node2)->string);
}

```

## APPLICATION USAGE

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

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

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

## RATIONALE

The requirement that the second argument (hereafter referred to as **p**) to the comparison function is a pointer to an element of the array implies that for every call all of the following expressions are non-zero:

```

( (char *)p - (char *)base ) % width == 0
(char *)p >= (char *)base

```

```
(char *)p < (char *)base + nel * width
```

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [hcreate\(\)](#)
- [lsearch\(\)](#)
- [qsort\(\)](#)
- [tdelete\(\)](#)

XBD [<stdlib.h>](#)

## CHANGE HISTORY

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

### Issue 6

The normative text is updated to avoid use of the term "must" for application requirements.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/11 is applied, adding to the DESCRIPTION the last sentence of the first non-shaded paragraph, and the following three paragraphs. The RATIONALE section is also updated. These changes are for alignment with the ISO C standard.

### Issue 7

The EXAMPLES section is revised.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0051 [756] is applied.

## 1.11. `calloc` — a memory allocator

---

### SYNOPSIS

```
#include <stdlib.h>

void *calloc(size_t nelem, size_t elsize);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `calloc()` function shall allocate unused space for an array of `nelem` elements each of whose size in bytes is `elsize`. The space shall be initialized to all bits 0.

The order and contiguity of storage allocated by successive calls to `calloc()` is unspecified. The pointer returned if the allocation succeeds shall be suitably aligned so that it may be assigned to a pointer to any type of object with a fundamental alignment requirement and then used to access such an object or an array of such objects in the space allocated (until the space is explicitly freed or reallocated). Each such allocation shall yield a pointer to an object disjoint from any other object. The pointer returned shall point to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer shall be returned. If the size of the space requested is 0, the behavior is implementation-defined: either a null pointer shall be returned, or the behavior shall be as if the size were some non-zero value, except that the behavior is undefined if the returned pointer is used to access an object.

For purposes of determining the existence of a data race, `calloc()` shall behave as though it accessed only memory locations accessible through its arguments and not other static duration storage. The function may, however, visibly modify the storage that it allocates. Calls to `aligned_alloc()`, `calloc()`, `free()`, `malloc()`, [ADV] `posix_memalign()`, [CX] `reallocarray()`, and `realloc()` that allocate or deallocate a particular region of memory shall occur in a single total order (see 4.15.1 Memory Ordering), and each such deallocation call shall synchronize with the next allocation (if any) in this order.

## RETURN VALUE

Upon successful completion, `calloc()` shall return a pointer to the allocated space; if either `nelem` or `elsize` is 0, the application shall ensure that the pointer is not used to access an object.

Otherwise, it shall return a null pointer [CX] and set `errno` to indicate the error.

## ERRORS

The `calloc()` function shall fail if:

- **[ENOMEM]** [CX] Insufficient memory is available, including the case when `nelem` \* `elsize` would overflow.

The `calloc()` function may fail if:

- **[EINVAL]** [CX] `nelem` or `elsize` is 0 and the implementation does not support 0 sized allocations.
- 

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

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

## RATIONALE

See the RATIONALE for `malloc()`.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `aligned_alloc()`

- `free()`

- `malloc()`

- `realloc()`

XBD `<stdlib.h>`

## CHANGE HISTORY

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

### Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The setting of `errno` and the [ENOMEM] error condition are mandatory if an insufficient memory condition occurs.

### Issue 7

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0053 [526] is applied.

### Issue 8

Austin Group Defect 374 is applied, changing the RETURN VALUE and ERRORS sections in relation to 0 sized allocations.

Austin Group Defect 1218 is applied, changing the [ENOMEM] error.

Austin Group Defect 1302 is applied, aligning this function with the ISO/IEC 9899:2018 standard.

Austin Group Defect 1387 is applied, changing the RATIONALE section.

---

## 1.12. clearerr

---

### SYNOPSIS

```
#include <stdio.h>

void clearerr(FILE *stream);
```

### DESCRIPTION

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `clearerr()` function shall clear the end-of-file and error indicators for the stream to which `stream` points.

The `clearerr()` function shall not change the setting of `errno` if `stream` is valid.

### RETURN VALUE

The `clearerr()` function shall not return a value.

### ERRORS

No errors are defined.

---

*The following sections are informative.*

### EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

XBD `<stdio.h>`

## CHANGE HISTORY

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

### Issue 7

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0057 [401] is applied.

---

## 1.13. `clock_getres`, `clock_gettime`, `clock_settime` — clock and timer functions

---

### SYNOPSIS

```
#include <time.h>

int clock_getres(clockid_t clock_id, struct timespec *res);
int clock_gettime(clockid_t clock_id, struct timespec *tp);
int clock_settime(clockid_t clock_id, const struct timespec *tp);
```

### DESCRIPTION

The `clock_getres()` function shall return the resolution of any clock. Clock resolutions are implementation-defined and cannot be set by a process. If the argument `res` is not NULL, the resolution of the specified clock shall be stored in the location pointed to by `res`. If `res` is NULL, the clock resolution is not returned. If the `time` argument of `clock_settime()` is not a multiple of `res`, then the value is truncated to a multiple of `res`.

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

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

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

If the value of the `CLOCK_REALTIME` clock is set via `clock_settime()`, the new value of the clock shall be used to determine the time of expiration for absolute time services based upon the `CLOCK_REALTIME` clock. This applies to the time at which armed absolute timers expire. If the absolute time requested at the invocation of such a time service is before the new value of the clock, the time

service shall expire immediately as if the clock had reached the requested time normally.

Setting the value of the CLOCK\_REALTIME clock via `clock_settime()` shall have no effect on threads that are blocked waiting for a relative time service based upon this clock, including the `nanosleep()` and `thrd_sleep()` functions; nor on the expiration of relative timers based upon this clock. Consequently, these time services shall expire when the requested relative interval elapses, independently of the new or old value of the clock.

All implementations shall support a `clock_id` of CLOCK\_MONOTONIC defined in `<time.h>`. This clock represents the monotonic clock for the system. For this clock, the value returned by `clock_gettime()` represents the amount of time (in seconds and nanoseconds) since an unspecified point in the past (for example, system start-up time, or the Epoch). This point does not change after system start-up time. The value of the CLOCK\_MONOTONIC clock cannot be set via `clock_settime()`. This function shall fail if it is invoked with a `clock_id` argument of CLOCK\_MONOTONIC.

The effect of setting a clock via `clock_settime()` on armed per-process timers associated with a clock other than CLOCK\_REALTIME is implementation-defined.

If the value of the CLOCK\_REALTIME clock is set via `clock_settime()`, the new value of the clock shall be used to determine the time at which the system shall awaken a thread blocked on an absolute `clock_nanosleep()` call based upon the CLOCK\_REALTIME clock. If the absolute time requested at the invocation of such a time service is before the new value of the clock, the call shall return immediately as if the clock had reached the requested time normally.

Setting the value of the CLOCK\_REALTIME clock via `clock_settime()` shall have no effect on any thread that is blocked on a relative `clock_nanosleep()` call. Consequently, the call shall return when the requested relative interval elapses, independently of the new or old value of the clock.

Appropriate privileges to set a particular clock are implementation-defined.

## CPU-Time Clock Support (Option: CPT)

If `_POSIX_CPUTIME` is defined, implementations shall support clock ID values obtained by invoking `clock_getcpuclockid()`, which represent the CPU-time clock of a given process. Implementations shall also support the special `clockid_t` value CLOCK\_PROCESS\_CPUTIME\_ID, which represents the CPU-time clock of the calling process when invoking one of the `clock_*` or `timer_*` functions. For these clock IDs, the values returned by `clock_gettime()` and specified by `clock_settime()` represent the amount of execution time of the process associated with the clock. Changing the value of a CPU-time clock via

`clock_settime()` shall have no effect on the behavior of the sporadic server scheduling policy.

## Thread CPU-Time Clock Support (Option: TCT)

If `_POSIX_THREAD_CPUTIME` is defined, implementations shall support clock ID values obtained by invoking `pthread_getcpu_clockid()`, which represent the CPU-time clock of a given thread. Implementations shall also support the special `clockid_t` value `CLOCK_THREAD_CPUTIME_ID`, which represents the CPU-time clock of the calling thread when invoking one of the `clock_*` or `timer_*` functions. For these clock IDs, the values returned by `clock_gettime()` and specified by `clock_settime()` shall represent the amount of execution time of the thread associated with the clock. Changing the value of a CPU-time clock via `clock_settime()` shall have no effect on the behavior of the sporadic server scheduling policy.

## RETURN VALUE

A return value of 0 shall indicate that the call succeeded. A return value of -1 shall indicate that an error occurred, and `errno` shall be set to indicate the error.

## ERRORS

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

- **[EINVAL]**

The `clock_id` argument does not specify a known clock.

The `clock_gettime()` function shall fail if:

- **[EOVERFLOW]**

The number of seconds will not fit in an object of type `time_t`.

The `clock_settime()` function shall fail if:

- **[EINVAL]**

The `tp` argument to `clock_settime()` is outside the range for the given clock ID.

- **[EINVAL]**

The `tp` argument specified a nanosecond value less than zero or greater than or equal to 1000 million.

- **[EINVAL]**

The value of the `clock_id` argument is CLOCK\_MONOTONIC.

The `clock_settime()` function may fail if:

- **[EPERM]**

The requesting process does not have appropriate privileges to set the specified clock.

---

## APPLICATION USAGE

Note that the absolute value of the monotonic clock is meaningless (because its origin is arbitrary), and thus there is no need to set it. Furthermore, realtime applications can rely on the fact that the value of this clock is never set and, therefore, that time intervals measured with this clock will not be affected by calls to `clock_settime()`.

## EXAMPLES

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- Scheduling Policies
- `clock_getcpu_clockid()`
- `clock_nanosleep()`
- `ctime()`
- `mq_receive()`

- `mq_send()`
- `nanosleep()`
- `pthread_mutex_clockclock()`
- `sem_clockwait()`
- `thrd_sleep()`
- `time()`
- `timer_create()`
- `timer_getoverrun()`

XBD `<time.h>`

## CHANGE HISTORY

### First released in Issue 5

Included for alignment with the POSIX Realtime Extension.

### Issue 6

- The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Timers option.
- The APPLICATION USAGE section is added.
- Clarification is added of the effect of resetting the clock resolution.
- CPU-time clocks and the `clock_getcpu_clockid()` function are added for alignment with IEEE Std 1003.1d-1999.
- Changes for alignment with IEEE Std 1003.1j-2000:
- The DESCRIPTION is updated as follows:
  - The value returned by `clock_gettime()` for CLOCK\_MONOTONIC is specified.
  - The `clock_settime()` function failing for CLOCK\_MONOTONIC is specified.
  - The effects of `clock_settime()` on the `clock_nanosleep()` function with respect to CLOCK\_REALTIME are specified.
- An [EINVAL] error is added to the ERRORS section, indicating that `clock_settime()` fails for CLOCK\_MONOTONIC.

- The APPLICATION USAGE section notes that the CLOCK\_MONOTONIC clock need not and shall not be set by `clock_settime()` since the absolute value of the CLOCK\_MONOTONIC clock is meaningless.
- The `clock_nanosleep()`, `mq_timedreceive()`, `mq_timedsend()`, `pthread_mutex_timedlock()`, `sem_timedwait()`, `timer_create()`, and `timer_settime()` functions are added to the SEE ALSO section.

## Issue 7

- Functionality relating to the Clock Selection option is moved to the Base.
- The `clock_getres()`, `clock_gettime()`, and `clock_settime()` functions are moved from the Timers option to the Base.
- POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0058 [106] is applied.

## Issue 8

- Austin Group Defect 1302 is applied, changing "the `nanosleep()` function" to "the `nanosleep()` and `thrd_sleep()` functions".
  - Austin Group Defect 1346 is applied, requiring support for Monotonic Clock.
-

## 1.14. `clock_getres`, `clock_gettime`, `clock_settime`

---

### SYNOPSIS

```
[CX] #include <time.h>

int clock_getres(clockid_t clock_id, struct timespec *res);
int clock_gettime(clockid_t clock_id, struct timespec *tp);
int clock_settime(clockid_t clock_id, const struct timespec *tp);
```

### DESCRIPTION

The `clock_getres()` function shall return the resolution of any clock. Clock resolutions are implementation-defined and cannot be set by a process. If the argument `res` is not NULL, the resolution of the specified clock shall be stored in the location pointed to by `res`. If `res` is NULL, the clock resolution is not returned. If the `time` argument of `clock_settime()` is not a multiple of `res`, then the value is truncated to a multiple of `res`.

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

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

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

If the value of the `CLOCK_REALTIME` clock is set via `clock_settime()`, the new value of the clock shall be used to determine the time of expiration for absolute time services based upon the `CLOCK_REALTIME` clock. This applies to the time at which armed absolute timers expire. If the absolute time requested at the invocation of such a time service is before the new value of the clock, the time service shall expire immediately as if the clock had reached the requested time normally.

Setting the value of the CLOCK\_REALTIME clock via `clock_settime()` shall have no effect on threads that are blocked waiting for a relative time service based upon this clock, including the `nanosleep()` and `thrd_sleep()` functions; nor on the expiration of relative timers based upon this clock. Consequently, these time services shall expire when the requested relative interval elapses, independently of the new or old value of the clock.

All implementations shall support a `clock_id` of CLOCK\_MONOTONIC defined in `<time.h>`. This clock represents the monotonic clock for the system. For this clock, the value returned by `clock_gettime()` represents the amount of time (in seconds and nanoseconds) since an unspecified point in the past (for example, system start-up time, or the Epoch). This point does not change after system start-up time. The value of the CLOCK\_MONOTONIC clock cannot be set via `clock_settime()`. This function shall fail if it is invoked with a `clock_id` argument of CLOCK\_MONOTONIC.

The effect of setting a clock via `clock_settime()` on armed per-process timers associated with a clock other than CLOCK\_REALTIME is implementation-defined.

If the value of the CLOCK\_REALTIME clock is set via `clock_settime()`, the new value of the clock shall be used to determine the time at which the system shall awaken a thread blocked on an absolute `clock_nanosleep()` call based upon the CLOCK\_REALTIME clock. If the absolute time requested at the invocation of such a time service is before the new value of the clock, the call shall return immediately as if the clock had reached the requested time normally.

Setting the value of the CLOCK\_REALTIME clock via `clock_settime()` shall have no effect on any thread that is blocked on a relative `clock_nanosleep()` call. Consequently, the call shall return when the requested relative interval elapses, independently of the new or old value of the clock.

Appropriate privileges to set a particular clock are implementation-defined.

[CPT] If `_POSIX_CPUTIME` is defined, implementations shall support clock ID values obtained by invoking `clock_getcpuclockid()`, which represent the CPU-time clock of a given process. Implementations shall also support the special `clockid_t` value `CLOCK_PROCESS_CPUTIME_ID`, which represents the CPU-time clock of the calling process when invoking one of the `clock_*()` or `timer_*()` functions. For these clock IDs, the values returned by `clock_gettime()` and specified by `clock_settime()` represent the amount of execution time of the process associated with the clock. Changing the value of a CPU-time clock via `clock_settime()` shall have no effect on the behavior of the sporadic server scheduling policy (see [Scheduling Policies]).

[TCT] If `_POSIX_THREAD_CPUTIME` is defined, implementations shall support clock ID values obtained by invoking `pthread_getcpuclockid()`, which represent the CPU-time clock of a given thread. Implementations shall also support

the special `clockid_t` value `CLOCK_THREAD_CPUTIME_ID`, which represents the CPU-time clock of the calling thread when invoking one of the `clock_*` or `timer_*` functions. For these clock IDs, the values returned by `clock_gettime()` and specified by `clock_settime()` shall represent the amount of execution time of the thread associated with the clock. Changing the value of a CPU-time clock via `clock_settime()` shall have no effect on the behavior of the sporadic server scheduling policy (see [Scheduling Policies]).

## RETURN VALUE

A return value of 0 shall indicate that the call succeeded. A return value of -1 shall indicate that an error occurred, and `errno` shall be set to indicate the error.

## ERRORS

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

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

The `clock_gettime()` function shall fail if:

- **[OVERFLOW]** The number of seconds will not fit in an object of type `time_t`.

The `clock_settime()` function shall fail if:

- **[EINVAL]** The `tp` argument to `clock_settime()` is outside the range for the given clock ID.
- **[EINVAL]** The `tp` argument specified a nanosecond value less than zero or greater than or equal to 1000 million.
- **[EINVAL]** The value of the `clock_id` argument is `CLOCK_MONOTONIC`.

The `clock_settime()` function may fail if:

- **[EPERM]** The requesting process does not have appropriate privileges to set the specified clock.

## EXAMPLES

None.

## APPLICATION USAGE

Note that the absolute value of the monotonic clock is meaningless (because its origin is arbitrary), and thus there is no need to set it. Furthermore, realtime applications can rely on the fact that the value of this clock is never set and, therefore, that time intervals measured with this clock will not be affected by calls to `clock_settime()`.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

[Scheduling Policies], `clock_getcpuclockid()`, `clock_nanosleep()`,  
`ctime()`, `mq_receive()`, `mq_send()`, `nanosleep()`,  
`pthread_mutex_clockclock()`, `sem_clockwait()`, `thrd_sleep()`,  
`time()`, `timer_create()`, `timer_getoverrun()`

XBD `<time.h>`

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

### Issue 6

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Timers option.

The APPLICATION USAGE section is added.

The following changes were made to align with the IEEE P1003.1a draft standard:

- Clarification is added of the effect of resetting the clock resolution.

CPU-time clocks and the `clock_getcpu_clockid()` function are added for alignment with IEEE Std 1003.1d-1999.

The following changes are added for alignment with IEEE Std 1003.1j-2000:

- The DESCRIPTION is updated as follows:
- The value returned by `clock_gettime()` for CLOCK\_MONOTONIC is specified.
- The `clock_settime()` function failing for CLOCK\_MONOTONIC is specified.
- The effects of `clock_settime()` on the `clock_nanosleep()` function with respect to CLOCK\_REALTIME are specified.
- An [EINVAL] error is added to the ERRORS section, indicating that `clock_settime()` fails for CLOCK\_MONOTONIC.
- The APPLICATION USAGE section notes that the CLOCK\_MONOTONIC clock need not and shall not be set by `clock_settime()` since the absolute value of the CLOCK\_MONOTONIC clock is meaningless.
- The `clock_nanosleep()`, `mq_timedreceive()`, `mq_timedsend()`, `pthread_mutex_timedlock()`, `sem_timedwait()`, `timer_create()`, and `timer_settime()` functions are added to the SEE ALSO section.

## Issue 7

Functionality relating to the Clock Selection option is moved to the Base.

The `clock_getres()`, `clock_gettime()`, and `clock_settime()` functions are moved from the Timers option to the Base.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0058 [106] is applied.

## Issue 8

Austin Group Defect 1302 is applied, changing "the `nanosleep()` function" to "the `nanosleep()` and `thrd_sleep()` functions".

Austin Group Defect 1346 is applied, requiring support for Monotonic Clock.

## 1.15. `clock_nanosleep`

### SYNOPSIS

```
#include <time.h>

int clock_nanosleep(clockid_t clock_id, int flags,
                     const struct timespec *rqtp, struct timespec *ri
```

### DESCRIPTION

If the flag `TIMER_ABSTIME` is not set in the `flags` argument, the `clock_nanosleep()` function shall cause the current thread to be suspended from execution until either the time interval specified by the `rqtp` argument has elapsed, or a signal is delivered to the calling thread and its action is to invoke a signal-catching function, or the process is terminated. The clock used to measure the time shall be the clock specified by `clock_id`.

If the flag `TIMER_ABSTIME` is set in the `flags` argument, the `clock_nanosleep()` function shall cause the current thread to be suspended from execution until either the time value of the clock specified by `clock_id` reaches the absolute time specified by the `rqtp` argument, or a signal is delivered to the calling thread and its action is to invoke a signal-catching function, or the process is terminated. If, at the time of the call, the time value specified by `rqtp` is less than or equal to the time value of the specified clock, then `clock_nanosleep()` shall return immediately and the calling process shall not be suspended.

The suspension time caused by this function may be longer than requested because the argument value is rounded up to an integer multiple of the sleep resolution, or because of the scheduling of other activity by the system. But, except for the case of being interrupted by a signal, the suspension time for the relative `clock_nanosleep()` function (that is, with the `TIMER_ABSTIME` flag not set) shall not be less than the time interval specified by `rqtp`, as measured by the corresponding clock. The suspension for the absolute `clock_nanosleep()` function (that is, with the `TIMER_ABSTIME` flag set) shall be in effect at least until the value of the corresponding clock reaches the absolute time specified by `rqtp`, except for the case of being interrupted by a signal.

The use of the `clock_nanosleep()` function shall have no effect on the action or blockage of any signal.

The `clock_nanosleep()` function shall fail if the `clock_id` argument refers to the CPU-time clock of the calling thread. It is unspecified whether `clock_id` values of other CPU-time clocks are allowed.

## RETURN VALUE

If the `clock_nanosleep()` function returns because the requested time has elapsed, its return value shall be zero.

If the `clock_nanosleep()` function returns because it has been interrupted by a signal, it shall return the corresponding error value. For the relative `clock_nanosleep()` function, if the `rmtp` argument is non-NULL, the `timespec` structure referenced by it shall be updated to contain the amount of time remaining in the interval (the requested time minus the time actually slept). The `rqtp` and `rmtp` arguments can point to the same object. If the `rmtp` argument is NULL, the remaining time is not returned. The absolute `clock_nanosleep()` function has no effect on the structure referenced by `rmtp`.

If `clock_nanosleep()` fails, it shall return the corresponding error value.

## ERRORS

The `clock_nanosleep()` function shall fail if:

- **[EINTR]**

The `clock_nanosleep()` function was interrupted by a signal.

- **[EINVAL]**

The `rqtp` argument specified a nanosecond value less than zero or greater than or equal to 1000 million; or the TIMER\_ABSTIME flag was specified in flags and the `rqtp` argument is outside the range for the clock specified by `clock_id`; or the `clock_id` argument does not specify a known clock, or specifies the CPU-time clock of the calling thread.

- **[ENOTSUP]**

The `clock_id` argument specifies a clock for which `clock_nanosleep()` is not supported, such as a CPU-time clock.

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

Calling `clock_nanosleep()` with the value `TIMER_ABSTIME` not set in the `flags` argument and with a `clock_id` of `CLOCK_REALTIME` is equivalent to calling `nanosleep()` with the same `rqtp` and `rmtp` arguments.

## RATIONALE

The `nanosleep()` function specifies that the system-wide clock `CLOCK_REALTIME` is used to measure the elapsed time for this time service. However, with the introduction of the monotonic clock `CLOCK_MONOTONIC` a new relative sleep function is needed to allow an application to take advantage of the special characteristics of this clock.

There are many applications in which a process needs to be suspended and then activated multiple times in a periodic way; for example, to poll the status of a non-interrupting device or to refresh a display device. For these cases, it is known that precise periodic activation cannot be achieved with a relative `sleep()` or `nanosleep()` function call. Suppose, for example, a periodic process that is activated at time  $T_0$ , executes for a while, and then wants to suspend itself until time  $T_0+T$ , the period being  $T$ . If this process wants to use the `nanosleep()` function, it must first call `clock_gettime()` to get the current time, then calculate the difference between the current time and  $T_0+T$  and, finally, call `nanosleep()` using the computed interval. However, the process could be preempted by a different process between the two function calls, and in this case the interval computed would be wrong; the process would wake up later than desired. This problem would not occur with the absolute `clock_nanosleep()` function, since only one function call would be necessary to suspend the process until the desired time. In other cases, however, a relative sleep is needed, and that is why both functionalities are required.

Although it is possible to implement periodic processes using the timers interface, this implementation would require the use of signals, and the reservation of some signal numbers. In this regard, the reasons for including an absolute version of the `clock_nanosleep()` function in POSIX.1-2024 are the same as for the inclusion of the relative `nanosleep()`.

It is also possible to implement precise periodic processes using `pthread_cond_timedwait()`, in which an absolute timeout is specified that

takes effect if the condition variable involved is never signaled. However, the use of this interface is unnatural, and involves performing other operations on mutexes and condition variables that imply an unnecessary overhead. Furthermore, `pthread_cond_timedwait()` is not available in implementations that do not support threads.

Although the interface of the relative and absolute versions of the new high resolution sleep service is the same `clock_nanosleep()` function, the `rmtp` argument is only used in the relative sleep. This argument is needed in the relative `clock_nanosleep()` function to reissue the function call if it is interrupted by a signal, but it is not needed in the absolute `clock_nanosleep()` function call; if the call is interrupted by a signal, the absolute `clock_nanosleep()` function can be invoked again with the same `rqtp` argument used in the interrupted call.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `clock_getres()`
  - `nanosleep()`
  - `pthread_cond_clockwait()`
  - `sleep()`
- XBD `<time.h>`

## CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

### Issue 7

The `clock_nanosleep()` function is moved from the Clock Selection option to the Base.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0068 [909] is applied.

## 1.16. `clock_settime`

---

### Synopsis

```
#include <time.h>

int clock_getres(clockid_t clock_id, struct timespec *res);
int clock_gettime(clockid_t clock_id, struct timespec *tp);
int clock_settime(clockid_t clock_id, const struct timespec *tp);
```

### Description

The `clock_getres()` function shall return the resolution of any clock. Clock resolutions are implementation-defined and cannot be set by a process. If the argument `res` is not NULL, the resolution of the specified clock shall be stored in the location pointed to by `res`. If `res` is NULL, the clock resolution is not returned. If the `time` argument of `clock_settime()` is not a multiple of `res`, then the value is truncated to a multiple of `res`.

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

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

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

#### `CLOCK_REALTIME` Behavior

If the value of the `CLOCK_REALTIME` clock is set via `clock_settime()`, the new value of the clock shall be used to determine the time of expiration for absolute time services based upon the `CLOCK_REALTIME` clock. This applies to the time at which armed absolute timers expire. If the absolute time requested at

the invocation of such a time service is before the new value of the clock, the time service shall expire immediately as if the clock had reached the requested time normally.

Setting the value of the CLOCK\_REALTIME clock via `clock_settime()` shall have no effect on threads that are blocked waiting for a relative time service based upon this clock, including the `nanosleep()` and `thrd_sleep()` functions; nor on the expiration of relative timers based upon this clock. Consequently, these time services shall expire when the requested relative interval elapses, independently of the new or old value of the clock.

## CLOCK\_MONOTONIC Behavior

All implementations shall support a `clock_id` of CLOCK\_MONOTONIC defined in `<time.h>`. This clock represents the monotonic clock for the system. For this clock, the value returned by `clock_gettime()` represents the amount of time (in seconds and nanoseconds) since an unspecified point in the past (for example, system start-up time, or the Epoch). This point does not change after system start-up time. The value of the CLOCK\_MONOTONIC clock cannot be set via `clock_settime()`. This function shall fail if it is invoked with a `clock_id` argument of CLOCK\_MONOTONIC.

## clock\_nanosleep Interaction

If the value of the CLOCK\_REALTIME clock is set via `clock_settime()`, the new value of the clock shall be used to determine the time at which the system shall awaken a thread blocked on an absolute `clock_nanosleep()` call based upon the CLOCK\_REALTIME clock. If the absolute time requested at the invocation of such a time service is before the new value of the clock, the call shall return immediately as if the clock had reached the requested time normally.

Setting the value of the CLOCK\_REALTIME clock via `clock_settime()` shall have no effect on any thread that is blocked on a relative `clock_nanosleep()` call. Consequently, the call shall return when the requested relative interval elapses, independently of the new or old value of the clock.

## Timer Effects

The effect of setting a clock via `clock_settime()` on armed per-process timers associated with a clock other than CLOCK\_REALTIME is implementation-defined.

## Privileges

Appropriate privileges to set a particular clock are implementation-defined.

## CPU-Time Clocks [CPT]

If `_POSIX_CPUTIME` is defined, implementations shall support clock ID values obtained by invoking `clock_getcpuclockid()`, which represent the CPU-time clock of a given process. Implementations shall also support the special `clockid_t` value `CLOCK_PROCESS_CPUTIME_ID`, which represents the CPU-time clock of the calling process when invoking one of the `clock_*` or `timer_*` functions. For these clock IDs, the values returned by `clock_gettime()` and specified by `clock_settime()` represent the amount of execution time of the process associated with the clock. Changing the value of a CPU-time clock via `clock_settime()` shall have no effect on the behavior of the sporadic server scheduling policy.

## Thread CPU-Time Clocks [TCT]

If `_POSIX_THREAD_CPUTIME` is defined, implementations shall support clock ID values obtained by invoking `pthread_getcpuclockid()`, which represent the CPU-time clock of a given thread. Implementations shall also support the special `clockid_t` value `CLOCK_THREAD_CPUTIME_ID`, which represents the CPU-time clock of the calling thread when invoking one of the `clock_*` or `timer_*` functions. For these clock IDs, the values returned by `clock_gettime()` and specified by `clock_settime()` shall represent the amount of execution time of the thread associated with the clock. Changing the value of a CPU-time clock via `clock_settime()` shall have no effect on the behavior of the sporadic server scheduling policy.

## Return Value

Upon successful completion, `clock_getres()`, `clock_gettime()`, and `clock_settime()` shall return 0; otherwise, -1 shall be returned and `errno` set to indicate the error.

## Errors

These functions shall fail if:

- **EINVAL**: The `clock_id` argument does not specify a known clock, or the `tp` argument is outside the range for the specified clock.

The `clock_settime()` function shall fail if:

- **EPERM**: The requesting process does not have the appropriate privilege to set the specified clock.

- **EINVAL**: The `clock_id` argument is CLOCK\_MONOTONIC.

These functions may fail if:

- **EOVERFLOW**: The number of seconds will not fit in a signed 32-bit integer.

## Examples

No examples are provided in this specification.

## Application Usage

The following sections are informative.

## Rationale

None provided.

## Future Directions

None provided.

## See Also

- `clock_getcpuclockid()`
- `clock_nanosleep()`
- `nanosleep()`
- `pthread_getcpuclockid()`
- `thrd_sleep()`
- `<time.h>`

## XSH

Issue 7

*The following sections are informative.*

## 1.17. close, posix\_close

---

### SYNOPSIS

```
#include <unistd.h>

int close(int fildes);
int posix_close(int fildes, int flag);
```

### DESCRIPTION

The `close()` function shall deallocate the file descriptor indicated by `fildes`. To deallocate means to make the file descriptor available for return by subsequent calls to `open()` or other functions that allocate file descriptors. All process-owned file locks that the calling process owns on the file associated with the file descriptor shall be unlocked.

If `close()` is interrupted by a signal that is to be caught, then it is unspecified whether it returns -1 with `errno` set to [EINTR] and `fildes` remaining open, or returns -1 with `errno` set to [EINPROGRESS] and `fildes` being closed, or returns 0 to indicate successful completion; except that if `POSIX_CLOSE_RESTART` is defined as 0, then the option of returning -1 with `errno` set to [EINTR] and `fildes` remaining open shall not occur. If `close()` returns -1 with `errno` set to [EINTR], it is unspecified whether `fildes` can subsequently be passed to any function except `close()` or `posix_close()` without error. For all other error situations (except for [EBADF] where `fildes` was invalid), `fildes` shall be closed. If `fildes` was closed even though the close operation is incomplete, the close operation shall continue asynchronously and the process shall have no further ability to track the completion or final status of the close operation.

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

When all file descriptors associated with an open file description have been closed, the open file description shall be freed.

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

[XSI] If `fildes` refers to the manager side of a pseudo-terminal, and this is the last close, a SIGHUP signal shall be sent to the controlling process, if any, for

which the subsidiary side of the pseudo-terminal is the controlling terminal. It is unspecified whether closing the manager side of the pseudo-terminal flushes all queued input and output.

When there is an outstanding cancelable asynchronous I/O operation against `fildes` when `close()` is called, that I/O operation may be canceled. An I/O operation that is not canceled completes as if the `close()` operation had not yet occurred. All operations that are not canceled shall complete as if the `close()` blocked until the operations completed. The `close()` operation itself need not block awaiting such I/O completion. Whether any I/O operation is canceled, and which I/O operation may be canceled upon `close()`, is implementation-defined.

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

When all file descriptors associated with a socket have been closed, the socket shall be destroyed. If the socket is in connection-mode, and the `SO_LINGER` option is set for the socket with non-zero linger time, and the socket has untransmitted data, then `close()` shall block for up to the current linger interval until all data is transmitted.

The `posix_close()` function shall be equivalent to the `close()` function, except with the modifications based on the `flag` argument as described below. If `flag` is 0, then `posix_close()` shall not return -1 with `errno` set to [EINTR], which implies that `fildes` will always be closed (except for [EBADF], where `fildes` was invalid). If `flag` includes `POSIX_CLOSE_RESTART` and `POSIX_CLOSE_RESTART` is defined as a non-zero value, and `posix_close()` is interrupted by a signal that is to be caught, then `posix_close()` may return -1 with `errno` set to [EINTR], in which case `fildes` shall be left open; however, it is unspecified whether `fildes` can subsequently be passed to any function except `close()` or `posix_close()` without error. If `flag` is invalid, `posix_close()` may fail with `errno` set to [EINVAL], but shall otherwise behave as if `flag` had been 0 and close `fildes`.

## RETURN VALUE

Upon successful completion, 0 shall be returned; otherwise, -1 shall be returned and `errno` set to indicate the error.

# ERRORS

The `close()` and `posix_close()` functions shall fail if:

- **[EBADF]**

The `fildes` argument is not a open file descriptor.

- **[EINPROGRESS]**

The function was interrupted by a signal and `fildes` was closed but the close operation is continuing asynchronously.

The `close()` and `posix_close()` functions may fail if:

- **[EINTR]**

The function was interrupted by a signal, `POSIX_CLOSE_RESTART` is defined as non-zero, and (in the case of `posix_close()`) the `flag` argument included `POSIX_CLOSE_RESTART`, in which case `fildes` is still open.

- **[EIO]**

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

The `posix_close()` function may fail if:

- **[EINVAL]**

The value of the `flag` argument is invalid.

The `close()` and `posix_close()` functions shall not return an [EAGAIN] or [EWOULDBLOCK] error. If `POSIX_CLOSE_RESTART` is zero, the `close()` function shall not return an [EINTR] error. The `posix_close()` function shall not return an [EINTR] error unless `flag` includes a non-zero `POSIX_CLOSE_RESTART`.

---

*The following sections are informative.*

## EXAMPLES

### Reassigning a File Descriptor

The following example closes the file descriptor associated with standard output for the current process, re-assigns standard output to a new file descriptor, and closes the original file descriptor to clean up. This example assumes that the file descriptor 0 (which is the descriptor for standard input) is not closed.

```
#include <unistd.h>
...
int pfd;
...
close(1);
dup(pfd);
close(pfd);
...
```

Incidentally, this is exactly what could be achieved using:

```
dup2(pfd, 1);
close(pfd);
```

## Closing a File Descriptor

In the following example, `close()` is used to close a file descriptor after an unsuccessful attempt is made to associate that file descriptor with a stream.

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

#define LOCKFILE "/etc/ptmp"

...
int pfd;
FILE *fpfd;
...
if ((fpfd = fdopen (pfd, "w")) == NULL) {
    close(pfd);
    unlink(LOCKFILE);
    exit(1);
}
```

## APPLICATION USAGE

An application that had used the `stdio` routine `fopen()` to open a file should use the corresponding `fclose()` routine rather than `close()`. Once a file is closed, the file descriptor no longer exists, since the integer corresponding to it no longer refers to a file.

Implementations may use file descriptors that must be inherited into child processes for the child process to remain conforming, such as for message catalog or tracing purposes. Therefore, an application that calls `close()` on an arbitrary

integer risks non-conforming behavior, and `close()` can only portably be used on file descriptor values that the application has obtained through explicit actions, as well as the three file descriptors corresponding to the standard file streams. In multi-threaded parent applications, the practice of calling `close()` in a loop after `fork()` and before an `exec` call in order to avoid a race condition of leaking an unintended file descriptor into a child process, is therefore unsafe, and the race should instead be combatted by opening all file descriptors with the FD\_CLOEXEC bit set unless the file descriptor is intended to be inherited across `exec`.

Usage of `close()` on file descriptors STDIN\_FILENO, STDOUT\_FILENO, or STDERR\_FILENO should immediately be followed by an operation to reopen these file descriptors. Unexpected behavior will result if any of these file descriptors is left in a closed state (for example, an [EBADF] error from `perror()`) or if an unrelated `open()` or similar call later in the application accidentally allocates a file to one of these well-known file descriptors. Furthermore, a `close()` followed by a reopen operation (e.g., `open()`, `dup()`, etc.) is not atomic; `dup2()` should be used to change standard file descriptors.

## RATIONALE

The use of interruptible device close routines should be discouraged to avoid problems with the implicit closes of file descriptors, such as by `exec`, process termination, or `dup2()`. This volume of POSIX.1-2024 only intends to permit such behavior by specifying the [EINTR] error condition for `close()` and `posix_close()` with non-zero POSIX\_CLOSE\_RESTART, to allow applications a portable way to resume waiting for an event associated with the close operation (for example, a tape drive rewinding) after receiving an interrupt. This standard also permits implementations to define POSIX\_CLOSE\_RESTART to 0 if they do not choose to provide a way to restart an interrupted close action. Although the file descriptor is left open on [EINTR], it might no longer be usable; that is, passing it to any function except `close()` or `posix_close()` might result in an error such as [EIO]. If an application must guarantee that data will not be lost, it is recommended that the application use `fsync()` or `fdatasync()` prior to the close operation, rather than leaving the close operation to deal with pending I/O and risk an interrupt.

Earlier versions of this standard left the state of `fildes` unspecified after errors such as [EINTR] and [EIO]; and implementations differed on whether `close()` left `fildes` open after [EINTR]. This was unsatisfactory once threads were introduced, since multi-threaded applications need to know whether `fildes` has been closed. Applications cannot blindly call `close()` again, because if `fildes` was closed by the first call another thread could have been allocated a file

descriptor with the same value as `fildes`, which must not be closed by the first thread. On the other hand, the alternative of never retrying `close()` would lead to a file descriptor leak in implementations where `close()` did not close `fildes`, although such a leak may be harmless if the process is about to exit or the file descriptor is marked `FD_CLOEXEC` and the process is about to be replaced by `exec`. This standard introduced `posix_close()` with a `flag` argument that allows a choice between the two possible error behaviors, and leaves it unspecified which of the two behaviors is implemented by `close()` (although it is guaranteed to be one of the two behaviors of `posix_close()`, rather than leaving things completely unspecified as in earlier versions of the standard).

Note that the standard requires that `close()` and `posix_close()` must leave `fildes` open after `[EINTR]` (in the cases where `[EINTR]` is permitted) and must close the file descriptor regardless of all other errors (except `[EBADF]`, where `fildes` is already invalid). In general, portable applications should only retry a `close()` after checking for `[EINTR]` (and on implementations where `POSIX_CLOSE_RESTART` is defined to be zero, this retry loop will be dead code), and risk a file descriptor leak if a retry loop is not attempted. It should also be noted that `[EINTR]` is only possible if `close()` can be interrupted; if no signal handlers are installed, then `close()` will not be interrupted. Conversely, if a single-threaded application can guarantee that no file descriptors are opened or closed in signal handlers, then a retry loop without checking for `[EINTR]` will be harmless (since the retry will fail with `[EBADF]`), but guaranteeing that no external libraries introduce the use of threading can be difficult. There are additional guarantees for applications which will only ever be used on systems where `POSIX_CLOSE_RESTART` is defined as 0. These observations should help in determining whether an application needs to have its `close()` calls audited for replacement with `posix_close()`.

It should also be noted that the requirement for `posix_close()` with a `flag` of 0 to always close `fildes`, even if an error is reported, is similar to the requirements on `fclose()` to always release the stream, even if an error is encountered while flushing data.

Implementations that previously always closed `fildes` can meet the new requirements by translating `[EINTR]` to `[EINPROGRESS]` in `close()`; and may define `POSIX_CLOSE_RESTART` to 0 rather than having to add restart semantics. On the other hand, implementations that previously left `fildes` open on `[EINTR]` can map that to `posix_close()` with `POSIX_CLOSE_RESTART`, and must add the semantics of `posix_close()` when `flag` is 0; one possibility is by calling the original `close()` implementation, checking for failure, and on `[EINTR]`, using actions similar to `dup2()` to replace the incomplete close operation with another file descriptor that can be closed immediately by another call to the original `close()`, all before returning to the application. Either way, `close()` should

always map to one of the two behaviors of `posix_close()`, and implementations are encouraged to keep the behavior of `close()` unchanged so as not to break implementation-specific expectations of older applications that were relying on behavior not specified by older versions of this standard.

The standard developers considered introducing a thread-local variable that `close()` would set to indicate whether it had closed `fildes` when returning -1. However, this was rejected in favor of the simpler solution of tightening `close()` to guarantee that `fildes` is closed except for [EINTR], and exposing a choice of whether to expect [EINTR] by adding `posix_close()`. Additionally, while the name `posix_close()` is new to this standard, it is reminiscent of at least one implementation that introduced an alternate system call named `close_nocancel()` in order to allow an application to choose whether restart semantics were desired.

Another consideration was whether implementations might return [EAGAIN] as an extension and whether `close()` should be required to leave the file descriptor open in this case, since [EAGAIN] normally implies an operation should be retried. It seemed very unlikely that any implementation would have a legitimate reason to return [EAGAIN] or [EWOULDBLOCK], and therefore this requirement would mean applications have to include code for an error case that will never be used. Therefore `close()` is now forbidden from returning [EAGAIN] and [EWOULDBLOCK] errors.

Note that the requirement for `close()` on a socket to block for up to the current linger interval is not conditional on the `O_NONBLOCK` setting.

The standard developers rejected a proposal to add `closefrom()` to the standard. Because the standard permits implementations to use inherited file descriptors as a means of providing a conforming environment for the child process, it is not possible to standardize an interface that closes arbitrary file descriptors above a certain value while still guaranteeing a conforming environment.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `dup()`
- `exec`
- `exit()`

- `fclose()`
- `fopen()`
- `fork()`
- `open()`
- `perror()`
- `unlink()`
- XBD `<unistd.h>`

## CHANGE HISTORY

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

### Issue 5

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

### Issue 6

The DESCRIPTION related to a STREAMS-based file or pseudo-terminal is marked as part of the XSI STREAMS Option Group.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EIO] error condition is added as an optional error.
- The DESCRIPTION is updated to describe the state of the `fildes` file descriptor as unspecified if an I/O error occurs and an [EIO] error condition is returned.

Text referring to sockets is added to the DESCRIPTION.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that shared memory objects and memory mapped files (and not typed memory objects) are the types of memory objects to which the paragraph on last closes applies.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/12 is applied, correcting the XSH shaded text relating to the manager side of a pseudo-terminal. The reason for the change is that the behavior of pseudo-terminals and regular terminals should be as much alike as possible in this case; the change achieves that and matches historical behavior.

## Issue 7

Functionality relating to the XSI STREAMS option is marked obsolescent.

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

Austin Group Interpretation 1003.1-2001 #139 is applied, clarifying that the requirement for `close()` on a socket to block for up to the current linger interval is not conditional on the `O_NONBLOCK` setting.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0059 [419], XSH/TC1-2008/0060 [149], and XSH/TC1-2008/0061 [149] are applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0069 [555] is applied.

## Issue 8

Austin Group Defect 529 is applied, adding the `posix_close()` function and changing requirements for the `close()` function relating to [EINTR].

Austin Group Defect 768 is applied, adding OFD-owned file locks.

Austin Group Defect 1330 is applied, removing obsolescent interfaces.

Austin Group Defect 1466 is applied, changing the terminology used for pseudo-terminal devices.

Austin Group Defect 1525 is applied, clarifying that a socket is not destroyed until all file descriptors associated with it have been closed.

---

## 1.18. confstr — get configurable variables

---

### SYNOPSIS

```
#include <unistd.h>

size_t confstr(int name, char *buf, size_t len);
```

### DESCRIPTION

The `confstr()` function shall return 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.

The `name` argument represents the system variable to be queried. The implementation shall support the following name values, defined in `<unistd.h>`. It may support others:

- `_CS_PATH`
- `_CS_POSIX_V8_ILP32_OFF32_CFLAGS`
- `_CS_POSIX_V8_ILP32_OFF32_LDFLAGS`
- `_CS_POSIX_V8_ILP32_OFF32_LIBS`
- `_CS_POSIX_V8_ILP32_OFFBIG_CFLAGS`
- `_CS_POSIX_V8_ILP32_OFFBIG_LDFLAGS`
- `_CS_POSIX_V8_ILP32_OFFBIG_LIBS`
- `_CS_POSIX_V8_LP64_OFF64_CFLAGS`
- `_CS_POSIX_V8_LP64_OFF64_LDFLAGS`
- `_CS_POSIX_V8_LP64_OFF64_LIBS`
- `_CS_POSIX_V8_LPBIG_OFFBIG_CFLAGS`
- `_CS_POSIX_V8_LPBIG_OFFBIG_LDFLAGS`
- `_CS_POSIX_V8_LPBIG_OFFBIG_LIBS`
- `_CS_POSIX_V8_THREADS_CFLAGS`
- `_CS_POSIX_V8_THREADS_LDFLAGS`
- `_CS_POSIX_V8_WIDTH_RESTRICTED_ENVS`
- `_CS_V8_ENV`

[OB] `_CS_POSIX_V7_ILP32_OFF32_CFLAGS`  
[OB] `_CS_POSIX_V7_ILP32_OFF32_LDFLAGS`  
[OB] `_CS_POSIX_V7_ILP32_OFF32_LIBS`  
[OB] `_CS_POSIX_V7_ILP32_OFFBIG_CFLAGS`  
[OB] `_CS_POSIX_V7_ILP32_OFFBIG_LDFLAGS`  
[OB] `_CS_POSIX_V7_ILP32_OFFBIG_LIBS`  
[OB] `_CS_POSIX_V7_LP64_OFF64_CFLAGS`  
[OB] `_CS_POSIX_V7_LP64_OFF64_LDFLAGS`  
[OB] `_CS_POSIX_V7_LP64_OFF64_LIBS`  
[OB] `_CS_POSIX_V7_LPBIG_OFFBIG_CFLAGS`  
[OB] `_CS_POSIX_V7_LPBIG_OFFBIG_LDFLAGS`  
[OB] `_CS_POSIX_V7_LPBIG_OFFBIG_LIBS`  
[OB] `_CS_POSIX_V7_THREADS_CFLAGS`  
[OB] `_CS_POSIX_V7_THREADS_LDFLAGS`  
[OB] `_CS_POSIX_V7_WIDTH_RESTRICTED_ENVS`  
[OB] `_CS_V7_ENV`

If `len` is not 0, and if `name` has a configuration-defined value, `confstr()` shall copy 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()` shall truncate the string to `len`-1 bytes and null-terminate the result. The application can detect that the string was truncated by comparing the value returned by `confstr()` with `len`.

If `len` is 0 and `buf` is a null pointer, then `confstr()` shall still return the integer value as defined below, but shall not return a string. If `len` is 0 but `buf` is not a null pointer, the result is unspecified.

After a call to:

```
confstr(_CS_V8_ENV, buf, sizeof(buf))
```

the string stored in `buf` shall contain a space-separated list of the variable=value environment variable pairs an implementation requires as part of specifying a conforming environment, as described in the implementations' conformance documentation.

If the implementation supports the POSIX shell option, the string stored in `buf` after a call to:

```
confstr(_CS_PATH, buf, sizeof(buf))
```

can be used as a value of the `PATH` environment variable that accesses all of the standard utilities of POSIX.1-2024, that are provided in a manner accessible via the `exec` family of functions, if the return value is less than or equal to `sizeof(buf)`.

## RETURN VALUE

If `name` has a configuration-defined value, `confstr()` shall return the size of buffer that would be needed to hold the entire configuration-defined value including the terminating null. If this return value is greater than `len`, the string returned in `buf` is truncated.

If `name` is invalid, `confstr()` shall return 0 and set `errno` to indicate the error.

If `name` does not have a configuration-defined value, `confstr()` shall return 0 and leave `errno` unchanged.

## ERRORS

The `confstr()` function shall fail if:

- **[EINVAL]**
  - The value of the `name` argument is invalid.
- 

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

An application can distinguish between an invalid `name` parameter value and one that corresponds to a configurable variable that has no configuration-defined value by checking if `errno` is modified. This mirrors the behavior of `sysconf()`.

The original need for this function was to provide a way of finding the configuration-defined default value for the environment variable `PATH`. Since `PATH` can be modified by the user to include directories that could contain utilities replacing the standard utilities in the Shell and Utilities volume of POSIX.1-2024, applications need a way to determine the system-supplied `PATH` environment variable value that contains the correct search path for the standard utilities.

An application could use:

```
confstr(name, (char *)NULL, (size_t)0)
```

to find out how big a buffer is needed for the string value; use `malloc()` to allocate a buffer to hold the string; and call `confstr()` again to get the string. Alternately, it could allocate a fixed, static buffer that is big enough to hold most answers (perhaps 512 or 1024 bytes), but then use `malloc()` to allocate a larger buffer if it finds that this is too small.

## RATIONALE

Application developers can normally determine any configuration variable by means of reading from the stream opened by a call to:

```
popen("command -p getconf variable", "r");
```

The `confstr()` function with a `name` argument of `_CS_PATH` returns a string that can be used as a `PATH` environment variable setting that will reference the standard shell and utilities as described in the Shell and Utilities volume of POSIX.1-2024.

The `confstr()` function copies the returned string into a buffer supplied by the application instead of returning a pointer to a string. This allows a cleaner function in some implementations (such as those with lightweight threads) and resolves questions about when the application must copy the string returned.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `exec`
- `fpathconf()`
- `sysconf()`

XBD `<unistd.h>`

XCU `c17`

# CHANGE HISTORY

First released in Issue 4. Derived from the ISO POSIX-2 standard.

## Issue 5

A table indicating the permissible values of `name` is added to the DESCRIPTION. All those marked EX are new in this version.

## Issue 6

The Open Group Corrigendum U033/7 is applied. The return value for the case returning the size of the buffer now explicitly states that this includes the terminating null.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION is updated with new arguments which can be used to determine configuration strings for C compiler flags, linker/loader flags, and libraries for each different supported programming environment. This is a change to support data size neutrality.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The DESCRIPTION is updated to include text describing how `_CS_PATH` can be used to obtain a `PATH` to access the standard utilities.

The macros associated with the `c89` programming models are marked LEGACY and new equivalent macros associated with `c99` are introduced.

## Issue 7

Austin Group Interpretation 1003.1-2001 #047 is applied, adding the `_CS_V7_ENV` variable.

Austin Group Interpretations 1003.1-2001 #166 is applied to permit an additional compiler flag to enable threads.

The V6 variables for the supported programming environments are marked obsolescent.

The variables for the supported programming environments are updated to be V7.

The LEGACY variables and obsolescent values are removed.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0070 [810] and XSH/TC2-2008/0071 [911] are applied.

## Issue 8

Austin Group Defect 1330 is applied, changing "V7" to "V8" and "V6" to "V7".

---

## 1.19. `ctime` — convert a time value to a date and time string

---

### SYNOPSIS

```
#include <time.h>

char *ctime(const time_t *clock);
```

### DESCRIPTION

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `ctime()` function shall convert the time pointed to by `clock`, representing time in seconds since the Epoch, to local time in the form of a string. It shall be equivalent to:

```
asctime(localtime(clock))
```

The `asctime()`, `ctime()`, `gmtime()`, and `localtime()` functions shall return values in one of two static objects: a broken-down time structure and an array of `char`. Execution of any of the functions that return a pointer to one of these object types may overwrite the information in any object of the same type pointed to by the value returned from any previous call to any of them.

The `ctime()` function need not be thread-safe; however, `ctime()` shall avoid data races with all functions other than itself, `asctime()`, `gmtime()`, and `localtime()`.

### RETURN VALUE

The `ctime()` function shall return the pointer returned by `asctime()` with that broken-down time as an argument.

# ERRORS

No errors are defined.

---

## EXAMPLES

None.

## APPLICATION USAGE

This function is included only for compatibility with older implementations. It has undefined behavior if the resulting string would be too long, so the use of this function should be discouraged. On implementations that do not detect output string length overflow, it is possible to overflow the output buffer in such a way as to cause applications to fail, or possible system security violations. Also, this function does not support localized date and time formats. To avoid these problems, applications should use `strftime()` to generate strings from broken-down times.

Values for the broken-down time structure can be obtained by calling `gmtime()` or `localtime()`.

Attempts to use `ctime()` for times before the Epoch or for times beyond the year 9999 produce undefined results. Refer to `asctime()`.

## RATIONALE

The standard developers decided to mark the `ctime()` function obsolescent even though it is in the ISO C standard due to the possibility of buffer overflow. The ISO C standard also provides the `strftime()` function which can be used to avoid these problems.

## FUTURE DIRECTIONS

This function may be removed in a future version, but not until after it has been removed from the ISO C standard.

## SEE ALSO

- `asctime()`
- `clock()`
- `difftime()`
- `futimens()`
- `gmtime()`
- `localtime()`
- `mktime()`
- `strftime()`
- `strptime()`
- `time()`
- XBD `<time.h>`

## CHANGE HISTORY

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

### Issue 5

- Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.
- The `ctime_r()` function is included for alignment with the POSIX Threads Extension.
- A note indicating that the `ctime()` function need not be reentrant is added to the DESCRIPTION.

### Issue 6

- Extensions beyond the ISO C standard are marked.
- In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.
- The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

## Issue 7

- Austin Group Interpretation 1003.1-2001 #156 is applied.
- SD5-XSH-ERN-25 is applied, updating the APPLICATION USAGE.
- Austin Group Interpretation 1003.1-2001 #053 is applied, marking these functions obsolescent.
- The `ctime_r()` function is moved from the Thread-Safe Functions option to the Base.
- POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0066 [321,428] is applied.
- POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0075 [664] is applied.

## Issue 8

- Austin Group Defect 1302 is applied, aligning this function with the ISO/IEC 9899:2018 standard.
  - Austin Group Defect 1330 is applied, changing the FUTURE DIRECTIONS section.
  - Austin Group Defect 1376 is applied, removing CX shading from some text derived from the ISO C standard and updating it to match the ISO C standard.
  - Austin Group Defect 1410 is applied, removing the `ctime_r()` function.
-

## 1.20. `ctime_r` — convert a time value to a date and time string (REMOVED)

---

### SYNOPSIS

```
#include <time.h>

char *ctime(const time_t *clock);
```

### DESCRIPTION

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `ctime()` function shall convert the time pointed to by `clock`, representing time in seconds since the Epoch, to local time in the form of a string. It shall be equivalent to:

```
asctime(localtime(clock))
```

The `asctime()`, `ctime()`, `gmtime()`, and `localtime()` functions shall return values in one of two static objects: a broken-down time structure and an array of `char`. Execution of any of the functions that return a pointer to one of these object types may overwrite the information in any object of the same type pointed to by the value returned from any previous call to any of them.

The `ctime()` function need not be thread-safe; however, `ctime()` shall avoid data races with all functions other than itself, `asctime()`, `gmtime()`, and `localtime()`.

### RETURN VALUE

The `ctime()` function shall return the pointer returned by `asctime()` with that broken-down time as an argument.

# ERRORS

No errors are defined.

---

## EXAMPLES

None.

## APPLICATION USAGE

This function is included only for compatibility with older implementations. It has undefined behavior if the resulting string would be too long, so the use of this function should be discouraged. On implementations that do not detect output string length overflow, it is possible to overflow the output buffer in such a way as to cause applications to fail, or possible system security violations. Also, this function does not support localized date and time formats. To avoid these problems, applications should use `strftime()` to generate strings from broken-down times.

Values for the broken-down time structure can be obtained by calling `gmtime()` or `localtime()`.

Attempts to use `ctime()` for times before the Epoch or for times beyond the year 9999 produce undefined results. Refer to `asctime()`.

## RATIONALE

The standard developers decided to mark the `ctime()` function obsolescent even though it is in the ISO C standard due to the possibility of buffer overflow. The ISO C standard also provides the `strftime()` function which can be used to avoid these problems.

## FUTURE DIRECTIONS

This function may be removed in a future version, but not until after it has been removed from the ISO C standard.

## SEE ALSO

- `asctime()`
- `clock()`
- `difftime()`
- `futimens()`
- `gmtime()`
- `localtime()`
- `mktime()`
- `strftime()`
- `strptime()`
- `time()`
- XBD `<time.h>`

## CHANGE HISTORY

### First released in Issue 1

Derived from Issue 1 of the SVID.

### Issue 5

- Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.
- The `ctime_r()` function is included for alignment with the POSIX Threads Extension.
- A note indicating that the `ctime()` function need not be reentrant is added to the DESCRIPTION.

### Issue 6

- Extensions beyond the ISO C standard are marked.
- In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

- The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

## Issue 7

- Austin Group Interpretation 1003.1-2001 #156 is applied.
- SD5-XSH-ERN-25 is applied, updating the APPLICATION USAGE.
- Austin Group Interpretation 1003.1-2001 #053 is applied, marking these functions obsolescent.
- The `ctime_r()` function is moved from the Thread-Safe Functions option to the Base.
- POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0066 [321,428] is applied.
- POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0075 [664] is applied.

## Issue 8

- Austin Group Defect 1302 is applied, aligning this function with the ISO/IEC 9899:2018 standard.
  - Austin Group Defect 1330 is applied, changing the FUTURE DIRECTIONS section.
  - Austin Group Defect 1376 is applied, removing CX shading from some text derived from the ISO C standard and updating it to match the ISO C standard.
  - **Austin Group Defect 1410 is applied, removing the `ctime_r()` function.**
-

## 1.21. difftime — compute the difference between two calendar time values

---

### SYNOPSIS

```
#include <time.h>

double difftime(time_t time1, time_t time0);
```

### DESCRIPTION

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `difftime()` function shall compute the difference between two calendar times (as returned by `time()`): `time1 - time0`.

### RETURN VALUE

The `difftime()` function shall return the difference expressed in seconds as a type `double`.

### ERRORS

No errors are defined.

---

*The following sections are informative.*

### EXAMPLES

None.

# APPLICATION USAGE

None.

# RATIONALE

None.

# FUTURE DIRECTIONS

None.

# SEE ALSO

- `asctime()`
- `clock()`
- `ctime()`
- `futimens()`
- `gmtime()`
- `localtime()`
- `mktime()`
- `strftime()`
- `strptime()`
- `time()`

XBD `<time.h>`

# CHANGE HISTORY

First released in Issue 4. Derived from the ISO C standard.

---

## 1.22. div

---

### Synopsis

```
#include <stdlib.h>

div_t div(int numer, int denom);
```

### Description

The `div()` function shall compute the quotient and remainder of the division of the numerator `numer` by the denominator `denom`. If the division is inexact, the resulting quotient is the integer of lesser magnitude that is the nearest to the algebraic quotient. If the result cannot be represented, the behavior is undefined; otherwise, `quot*denom+rem` shall equal `numer`.

### Return Value

The `div()` function shall return a structure of type `div_t`, comprising both the quotient and the remainder. The structure includes the following members, in any order:

```
int quot; /* quotient */
int rem; /* remainder */
```

### Examples

None provided.

### Application Usage

None provided.

### Rationale

None provided.

## Future Directions

None provided.

## See Also

None provided.

## Copyright

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

## 1.23. exit

---

### SYNOPSIS

```
#include <stdlib.h>

void exit(int status);
```

### DESCRIPTION

The `exit()` function shall cause normal process termination to occur. No functions registered by the `at_quick_exit()` function shall be called. If a process calls the `exit()` function more than once, or calls the `quick_exit()` function in addition to the `exit()` function, the behavior is undefined.

The value of `status` can be 0, `EXIT_SUCCESS`, `EXIT_FAILURE`, or any other value, though only the least significant 8 bits (that is, `status` & 0377) shall be available from `wait()` and `waitpid()`; the full value shall be available from `waitid()` and in the `siginfo_t` passed to a signal handler for `SIGCHLD`.

The `exit()` function shall first call all functions registered by `atexit()`, in the reverse order of their registration, except that a function is called after any previously registered functions that had already been called at the time it was registered. Each function is called as many times as it was registered. If, during the call to any such function, a call to the `longjmp()` function is made that would terminate the call to the registered function, the behavior is undefined.

If a function registered by a call to `atexit()` fails to return, the remaining registered functions shall not be called and the rest of the `exit()` processing shall not be completed.

The `exit()` function shall then flush all open streams with unwritten buffered data. For each stream which is the active handle to its underlying file descriptor, and for which the file is not already at EOF and is capable of seeking, the file offset of the underlying open file description shall be set to the file position of the stream. For each open stream, the `exit()` function shall perform the equivalent of a `close()` on the file descriptor that is associated with the stream. Finally, the process shall be terminated with the same consequences as described in *Consequences of Process Termination*.

# RETURN VALUE

The `exit()` function does not return.

## ERRORS

No errors are defined.

## EXAMPLES

See APPLICATION USAGE.

## APPLICATION USAGE

When a stream that has unwritten buffered data is flushed by `exit()` there is no way for the calling process to discover whether or not `exit()` successfully wrote the data to the underlying file descriptor. Therefore, it is strongly recommended that applications always ensure there is no unwritten buffered data in any stream when calling `exit()`, or returning from the initial call to `main()`, with a `status` value that indicates no errors occurred.

For example, the following code demonstrates one way to ensure that `stdout` has already been successfully flushed before calling `exit()` with status 0. If the flush fails, the file descriptor underlying `stdout` is closed so that `exit()` will not try to repeat the failed write operation. If the flush succeeds, a final check with `ferror()` is performed to ensure that there were no write errors during earlier flush operations (that were not handled at the time).

```
int status = 0;
if (fflush(stdout) != 0) {
    perror("appname: standard output");
    close(fileno(stdout));
    status = 1;
}
else if (ferror(stdout)) {
    fputs("appname: write error on standard output\n", stderr);
    status = 1;
}
exit(status);
```

See also [\\_Exit\(\)](#).

# RATIONALE

See `_Exit()`.

# FUTURE DIRECTIONS

None.

# SEE ALSO

- `_Exit()`
- `at_quick_exit()`
- `atexit()`
- `exec`
- `fflush()`
- `longjmp()`
- `quick_exit()`
- `tmpfile()`
- `wait()`
- `waitid()`
- `<stdlib.h>`

# CHANGE HISTORY

## Issue 7

Austin Group Interpretation 1003.1-2001 #031 is applied, separating the `_Exit()` and `_exit()` functions from the `exit()` function.

Austin Group Interpretation 1003.1-2001 #085 is applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0100 [594] is applied.

## Issue 8

Austin Group Defect 610 is applied, clarifying the effects of `exit()` on open streams.

Austin Group Defect 1302 is applied, aligning this function with the ISO/IEC 9899:2018 standard.

Austin Group Defect 1490 is applied, changing the EXAMPLES and APPLICATION USAGE sections.

Austin Group Defect 1629 is applied, changing the APPLICATION USAGE section.

---

## 1.24. `fclose`

---

### SYNOPSIS

```
#include <stdio.h>

int fclose(FILE *stream);
```

### DESCRIPTION

The `fclose()` function shall cause the stream pointed to by `stream` to be flushed and the associated file to be closed. Any unwritten buffered data for the stream shall be written to the file; any unread buffered data shall be discarded. Whether or not the call succeeds, the stream shall be disassociated from the file and any buffer set by the `setbuf()` or `setvbuf()` function shall be disassociated from the stream. If the associated buffer was automatically allocated, it shall be deallocated.

If the file is not already at EOF, and the file is one capable of seeking, the file offset of the underlying open file description shall be set to the file position of the stream if the stream is the active handle to the underlying file description.

The `fclose()` function shall mark for update the last data modification and last file status change timestamps of the underlying file, if the stream was writable, and if buffered data remains that has not yet been written to the file. The `fclose()` function shall perform the equivalent of a `close()` on the file descriptor that is associated with the stream pointed to by `stream`.

After the call to `fclose()`, any use of `stream` results in undefined behavior.

### RETURN VALUE

Upon successful completion, `fclose()` shall return 0; otherwise, it shall return EOF and set `errno` to indicate the error.

### ERRORS

The `fclose()` function shall fail if:

- `[EAGAIN]`

- The O\_NONBLOCK flag is set for the file descriptor underlying `stream` and the thread would be delayed in the write operation.

- **[EBADF]**

- The file descriptor underlying stream is not valid.

- **[EFBIG]**

- An attempt was made to write a file that exceeds the maximum file size.

- **[EFBIG]**

- An attempt was made to write a file that exceeds the file size limit of the process. A SIGXFSZ signal shall also be generated for the thread.

- **[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.

- **[EINTR]**

- The `fclose()` function was interrupted by a signal.

- **[EIO]**

- The process is a member of a background process group attempting to write to its controlling terminal, TOSTOP is set, the calling thread is not blocking SIGTTOU, the process is not ignoring SIGTTOU, and the process group of the process is orphaned. This error may also be returned under implementation-defined conditions.

- **[ENOMEM]**

- The underlying stream was created by `open_memstream()` or `open_wmemstream()` and insufficient memory is available.

- **[ENOSPC]**

- There was no free space remaining on the device containing the file or in the buffer used by the `fmemopen()` function.

- **[EPIPE]**

- An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A SIGPIPE signal shall also be sent to the thread.

The `fclose()` function may fail if:

- **[ENXIO]**

- A request was made of a nonexistent device, or the request was outside the capabilities of the device.

## EXAMPLES

None.

## APPLICATION USAGE

Since after the call to `fclose()` any use of `stream` results in undefined behavior, `fclose()` should not be used on `stdin`, `stdout`, or `stderr` except immediately before process termination, so as to avoid triggering undefined behavior in other standard interfaces that rely on these streams. If there are any `atexit()` handlers registered by the application, such a call to `fclose()` should not occur until the last handler is finishing. Once `fclose()` has been used to close `stdin`, `stdout`, or `stderr`, there is no standard way to reopen any of these streams.

Use of `freopen()` to change `stdin`, `stdout`, or `stderr` instead of closing them avoids the danger of a file unexpectedly being opened as one of the special file descriptors `STDIN_FILENO`, `STDOUT_FILENO`, or `STDERR_FILENO` at a later time in the application.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- 2.5 Standard I/O Streams
- `atexit()`
- `close()`
- `fmemopen()`
- `fopen()`

- `freopen()`
- `getrlimit()`
- `open_memstream()`
- `<stdio.h>`

## CHANGE HISTORY

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

### Issue 5

Large File Summit extensions are added.

### Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EFBIG] error is added as part of the large file support extensions.
- The [ENXIO] optional error condition is added.

The DESCRIPTION is updated to note that the stream and any buffer are disassociated whether or not the call succeeds. This is for alignment with the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/28 is applied, updating the [EAGAIN] error in the ERRORS section from "the process would be delayed" to "the thread would be delayed".

### Issue 7

Austin Group Interpretation 1003.1-2001 #002 is applied, clarifying the interaction of file descriptors and streams.

The [ENOSPC] error condition is updated and the [ENOMEM] error is added from The Open Group Technical Standard, 2006, Extended API Set Part 1.

Changes are made related to support for finegrained timestamps.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0113 [87], XSH/TC1-2008/0114 [79], and XSH/TC1-2008/0115 [14] are applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0104 [555] is applied.

## Issue 8

Austin Group Defect 308 is applied, clarifying the handling of [EFBIG] errors.

Austin Group Defect 1669 is applied, removing XSI shading from part of the [EFBIG] error relating to the file size limit for the process.

## 1.25. `fdatasync`

---

### SYNOPSIS

```
#include <unistd.h>

int fdatasync(int fildes);
```

### DESCRIPTION

The `fdatasync()` function shall force all currently queued I/O operations associated with the file indicated by file descriptor `fildes` to the synchronized I/O completion state.

The functionality shall be equivalent to `fsync()` with the symbol `_POSIX_SYNCHRONIZED_IO` defined, with the exception that all I/O operations shall be completed as defined for synchronized I/O data integrity completion.

### RETURN VALUE

If successful, the `fdatasync()` function shall return the value 0; otherwise, the function shall return the value -1 and set `errno` to indicate the error. If the `fdatasync()` function fails, outstanding I/O operations are not guaranteed to have been completed.

### ERRORS

The `fdatasync()` function shall fail if:

- **[EBADF]**

The `fildes` argument is not a valid file descriptor.

- **[EINVAL]**

This implementation does not support synchronized I/O for this file.

In the event that any of the queued I/O operations fail, `fdatasync()` shall return the error conditions defined for `read()` and `write()`.

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

Note that even if the file descriptor is not open for writing, if there are any pending write requests on the underlying file, then that I/O will be completed prior to the return of `fdatasync()`.

An application that modifies a directory, for example, by creating a file in the directory, can invoke `fdatasync()` on the directory to ensure that the directory's entries are synchronized, although for most applications this should not be necessary (see XBD [4.11 File System Cache](#)).

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `aio_fsync()`
- `fcntl()`
- `fsync()`
- `open()`
- `read()`
- `write()`
- XBD `<unistd.h>`

## CHANGE HISTORY

**First released in Issue 5.** Included for alignment with the POSIX Realtime Extension.

## Issue 6

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Synchronized Input and Output option.

The `fdatasync()` function is marked as part of the Synchronized Input and Output option.

## Issue 7

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0110 [501] is applied.

## Issue 8

Austin Group Defect 672 is applied, changing the APPLICATION USAGE section.

## 1.26. `fdopen`

---

### SYNOPSIS

```
[CX] #include <stdio.h>

FILE *fdopen(int fildes, const char *mode);
```

### DESCRIPTION

The `fdopen()` function shall associate a stream with a file descriptor.

The `mode` argument points to a character string that is valid for `fopen()`. If the string begins with one of the following characters, then the stream shall be associated with `fildes` as specified. Otherwise, the behavior is undefined.

#### Mode Characters

'r'

- If `mode` includes '+', the associated stream shall be open for update (reading and writing)
- Otherwise, the stream shall be open for reading only
- If the open file description referenced by `fildes` has O\_APPEND set, it shall remain set

'w'

- If `mode` includes '+', the associated stream shall be open for update (reading and writing)
- Otherwise, the stream shall be open for writing only
- The file shall not be truncated by the `fdopen()` call
- If the open file description referenced by `fildes` has O\_APPEND set, it shall remain set

'a'

- If `mode` includes '+', the associated stream shall be open for update (reading and writing)
- Otherwise, the stream shall be open for writing only

- If the open file description referenced by `fildes` has O\_APPEND clear, it is unspecified whether O\_APPEND is set by the `fdopen()` call or remains clear

## Additional Mode Options

- The presence of 'x' in `mode` shall have no effect
- The FD\_CLOEXEC flag of `fildes` shall be unchanged if 'e' is not present, and shall be set by the `fdopen()` call if 'e' is present
- Additional values for the `mode` argument may be supported by an implementation

## Requirements

The application shall ensure that the mode of the stream as expressed by the `mode` argument is allowed by the file access mode of the open file description to which `fildes` refers.

The file position indicator associated with the new stream is set to the position indicated by the file offset associated with the file descriptor.

The error and end-of-file indicators for the stream shall be cleared. The `fdopen()` function may cause the last data access timestamp of the underlying file to be marked for update.

[SHM] If `fildes` refers to a shared memory object, the result of the `fdopen()` function is unspecified.

[TYM] If `fildes` refers to a typed memory object, the result of the `fdopen()` function is unspecified.

The `fdopen()` function shall preserve the offset maximum previously set for the open file description corresponding to `fildes`.

## RETURN VALUE

Upon successful completion, `fdopen()` shall return a pointer to a stream; otherwise, a null pointer shall be returned and `errno` set to indicate the error.

## ERRORS

The `fdopen()` function shall fail if:

- **[EMFILE]** {STREAM\_MAX} streams are currently open in the calling process.

The `fdopen()` function may fail if:

- **[EBADF]** The `fd` argument is not a valid file descriptor.
  - **[EINVAL]** The `mode` argument is not a valid mode.
  - **[EMFILE]** {FOPEN\_MAX} streams are currently open in the calling process.
  - **[ENOMEM]** Insufficient space to allocate a buffer.
- 

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

File descriptors are obtained from calls like `open()`, `dup()`, `creat()`, or `pipe()`, which open files but do not return streams.

## RATIONALE

The file descriptor may have been obtained from `open()`, `creat()`, `pipe()`, `dup()`, `fcntl()`, or `socket()`; inherited through `fork()`, `posix_spawn()`, or `exec`; or perhaps obtained by other means.

The meanings of the `mode` arguments of `fdopen()` and `fopen()` differ. With `fdopen()`, write ('w') mode cannot create or truncate a file, and append ('a') mode cannot create a file. Inclusion of a 'b' in the `mode` argument is allowed for consistency with `fopen()`; the 'b' has no effect on the resulting stream. Implementations differ as to whether specifying append ('a') mode causes the O\_APPEND flag to be set if it was clear, but they are encouraged to do so. Since `fdopen()` does not create a file, the 'x' mode modifier is silently ignored. The 'e' mode modifier is not strictly necessary for `fdopen()`, since FD\_CLOEXEC must not be changed when it is absent; however, it is standardized here since that modifier is necessary to avoid a data race in multi-threaded applications using `freopen()`, and consistency dictates that all functions accepting `mode` strings should allow the same set of strings.

# FUTURE DIRECTIONS

None.

## SEE ALSO

- 2.5.1 Interaction of File Descriptors and Standard I/O Streams
- `fclose()`
- `fmemopen()`
- `fopen()`
- `open()`
- `open_memstream()`
- `posix_spawn()`
- `socket()`
- XBD

## CHANGE HISTORY

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

### Issue 5

- The DESCRIPTION is updated for alignment with the POSIX Realtime Extension.
- Large File Summit extensions are added.

### Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, the use and setting of the `mode` argument are changed to include binary streams.
- In the DESCRIPTION, text is added for large file support to indicate setting of the offset maximum in the open file description.
- All errors identified in the ERRORS section are added.

- In the DESCRIPTION, text is added that the `fdopen()` function may cause `st_atime` to be updated.

The following changes were made to align with the IEEE P1003.1a draft standard:

- Clarification is added that it is the responsibility of the application to ensure that the mode is compatible with the open file descriptor.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that `fdopen()` results are unspecified for typed memory objects.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/30 is applied, making corrections to the RATIONALE.

## Issue 7

- SD5-XSH-ERN-149 is applied, adding the {STREAM\_MAX} [EMFILE] error condition.
- Changes are made related to support for finegrained timestamps.
- POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0121 [409] is applied.

## Issue 8

- Austin Group Defects 411 and 1526 are applied, changing the requirements for the `mode` argument.
-

## 1.27. feclearexcept — clear floating-point exception

---

### SYNOPSIS

```
#include <fenv.h>

int feclearexcept(int excepts);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `feclearexcept()` function shall attempt to clear the supported floating-point exceptions represented by `excepts`.

### RETURN VALUE

If the argument is zero or if all the specified exceptions were successfully cleared, `feclearexcept()` shall return zero. Otherwise, it shall return a non-zero value.

### ERRORS

No errors are defined.

---

*The following sections are informative.*

### EXAMPLES

None.

### APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `fegetexceptflag()`
- `feraiseexcept()`
- `fetestexcept()`

XBD `<fenv.h>`

## CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

## 1.28. fegetenv, fesetenv — get and set current floating-point environment

---

### SYNOPSIS

```
#include <fenv.h>

int fegetenv(fenv_t *envp);
int fesetenv(const fenv_t *envp);
```

### DESCRIPTION

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `fegetenv()` function shall attempt to store the current floating-point environment in the object pointed to by `envp`.

The `fesetenv()` function shall attempt to establish the floating-point environment represented by the object pointed to by `envp`. The argument `envp` shall point to an object set by a call to `fegetenv()` or `feholdexcept()`, or equal a floating-point environment macro. The `fesetenv()` function does not raise floating-point exceptions, but only installs the state of the floating-point status flags represented through its argument.

### RETURN VALUE

If the representation was successfully stored, `fegetenv()` shall return zero. Otherwise, it shall return a non-zero value. If the environment was successfully established, `fesetenv()` shall return zero. Otherwise, it shall return a non-zero value.

### ERRORS

No errors are defined.

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `feholdexcept()`
- `feupdateenv()`
- `<fenv.h>`

## CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

---

## 1.29. fegetexceptflag, fesetexceptflag — get and set floating-point status flags

---

### SYNOPSIS

```
#include <fenv.h>

int fegetexceptflag(fexcept_t *flagp, int excepts);
int fesetexceptflag(const fexcept_t *flagp, int excepts);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `fegetexceptflag()` function shall attempt to store an implementation-defined representation of the states of the floating-point status flags indicated by the argument `excepts` in the object pointed to by the argument `flagp`.

The `fesetexceptflag()` function shall attempt to set the floating-point status flags indicated by the argument `excepts` to the states stored in the object pointed to by `flagp`. The value pointed to by `flagp` shall have been set by a previous call to `fegetexceptflag()` whose second argument represented at least those floating-point exceptions represented by the argument `excepts`. This function does not raise floating-point exceptions, but only sets the state of the flags.

### RETURN VALUE

If the representation was successfully stored, `fegetexceptflag()` shall return zero. Otherwise, it shall return a non-zero value.

If the `excepts` argument is zero or if all the specified exceptions were successfully set, `fesetexceptflag()` shall return zero. Otherwise, it shall return a non-zero value.

# ERRORS

No errors are defined.

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `feclearexcept()`
- `feraiseexcept()`
- `fetestexcept()`
- XBD `<fenv.h>`

## CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.  
ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

## 1.30. fegetround, fesetround — get and set current rounding direction

---

### SYNOPSIS

```
#include <fenv.h>

int fegetround(void);
int fesetround(int round);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `fegetround()` function shall get the current rounding direction.

The `fesetround()` function shall establish the rounding direction represented by its argument `round`. If the argument is not equal to the value of a rounding direction macro, the rounding direction is not changed.

### RETURN VALUE

The `fegetround()` function shall return the value of the rounding direction macro representing the current rounding direction or a negative value if there is no such rounding direction macro or the current rounding direction is not determinable.

The `fesetround()` function shall return a zero value if and only if the requested rounding direction was established.

### ERRORS

No errors are defined.

---

*The following sections are informative.*

## EXAMPLES

The following example saves, sets, and restores the rounding direction, reporting an error and aborting if setting the rounding direction fails:

```
#include <fenv.h>
#include <assert.h>

void f(int round_dir)
{
    #pragma STDC FENV_ACCESS ON
    int save_round;
    int setround_ok;

    save_round = fegetround();
    setround_ok = fesetround(round_dir);
    assert(setround_ok == 0);
    /* ... */
    fesetround(save_round);
    /* ... */
}
```

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

XBD [`<fenv.h>`](#)

## CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

## 1.31. feholdexcept

---

### SYNOPSIS

```
#include <fenv.h>

int feholdexcept(fenv_t *envp);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `feholdexcept()` function shall save the current floating-point environment in the object pointed to by `envp`, clear the floating-point status flags, and then install a non-stop (continue on floating-point exceptions) mode, if available, for all floating-point exceptions.

### RETURN VALUE

The `feholdexcept()` function shall return zero if and only if non-stop floating-point exception handling was successfully installed.

### ERRORS

No errors are defined.

*The following sections are informative.*

### EXAMPLES

None.

### APPLICATION USAGE

None.

## RATIONALE

The `feholdexcept()` function should be effective on typical IEC 60559:1989 standard implementations which have the default non-stop mode and at least one other mode for trap handling or aborting. If the implementation provides only the non-stop mode, then installing the non-stop mode is trivial.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `fegetenv()`
- `feupdateenv()`
- XBD `<fenv.h>`

## CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

## 1.32. feof

---

### SYNOPSIS

```
#include <stdio.h>

int feof(FILE *stream);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `feof()` function shall test the end-of-file indicator for the stream pointed to by `stream`.

[CX] The `feof()` function shall not change the setting of `errno` if `stream` is valid.

### RETURN VALUE

The `feof()` function shall return non-zero if and only if the end-of-file indicator is set for `stream`.

### ERRORS

No errors are defined.

---

*The following sections are informative.*

### EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `clearerr()`
- `ferror()`
- `fopen()`

XBD `<stdio.h>`

## CHANGE HISTORY

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

### Issue 7

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0124 [401] is applied.

## 1.33. `feraiseexcept` — raise floating-point exception

---

### SYNOPSIS

```
#include <fenv.h>

int feraiseexcept(int excepts);
```

### DESCRIPTION

The `feraiseexcept()` function shall attempt to raise the supported floating-point exceptions represented by the `excepts` argument. The order in which these floating-point exceptions are raised is unspecified, except that if the `excepts` argument represents IEC 60559 valid coincident floating-point exceptions for atomic operations (namely overflow and inexact, or underflow and inexact), then overflow or underflow shall be raised before inexact. Whether the `feraiseexcept()` function additionally raises the inexact floating-point exception whenever it raises the overflow or underflow floating-point exception is implementation-defined.

### RETURN VALUE

If the argument is zero or if all the specified exceptions were successfully raised, `feraiseexcept()` shall return zero. Otherwise, it shall return a non-zero value.

### ERRORS

No errors are defined.

### EXAMPLES

None.

### APPLICATION USAGE

The effect is intended to be similar to that of floating-point exceptions raised by arithmetic operations. Hence, enabled traps for floating-point exceptions raised by

this function are taken.

## RATIONALE

Raising overflow or underflow is allowed to also raise inexact because on some architectures the only practical way to raise an exception is to execute an instruction that has the exception as a side-effect. The function is not restricted to accept only valid coincident expressions for atomic operations, so the function can be used to raise exceptions accrued over several operations.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `feclearexcept()`
- `fegetexceptflag()`
- `fetestexcept()`
- `<fenv.h>`

## CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

### Issue 7

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0111 [543] is applied.

---

## 1.34. `ferror` — test error indicator on a stream

---

### SYNOPSIS

```
#include <stdio.h>

int ferror(FILE *stream);
```

### DESCRIPTION

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `ferror()` function shall test the error indicator for the stream pointed to by `stream`.

The `ferror()` function shall not change the setting of `errno` if `stream` is valid.

### RETURN VALUE

The `ferror()` function shall return non-zero if and only if the error indicator is set for `stream`.

### ERRORS

No errors are defined.

---

### EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `clearerr()`
- `feof()`
- `fopen()`

XBD `<stdio.h>`

## CHANGE HISTORY

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

### Issue 7

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0125 [401] is applied.

---

## 1.35. fegetenv, fesetenv — get and set current floating-point environment

---

### SYNOPSIS

```
#include <fenv.h>

int fegetenv(fenv_t *envp);
int fesetenv(const fenv_t *envp);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `fegetenv()` function shall attempt to store the current floating-point environment in the object pointed to by `envp`.

The `fesetenv()` function shall attempt to establish the floating-point environment represented by the object pointed to by `envp`. The argument `envp` shall point to an object set by a call to `fegetenv()` or `feholdexcept()`, or equal a floating-point environment macro. The `fesetenv()` function does not raise floating-point exceptions, but only installs the state of the floating-point status flags represented through its argument.

### RETURN VALUE

If the representation was successfully stored, `fegetenv()` shall return zero. Otherwise, it shall return a non-zero value. If the environment was successfully established, `fesetenv()` shall return zero. Otherwise, it shall return a non-zero value.

### ERRORS

No errors are defined.

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `feholdexcept()`
- `feupdateenv()`
- `<fenv.h>`

## CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

---

## 1.36. fegetexceptflag, fesetexceptflag — get and set floating-point status flags

---

### SYNOPSIS

```
#include <fenv.h>

int fegetexceptflag(fexcept_t *flagp, int excepts);
int fesetexceptflag(const fexcept_t *flagp, int excepts);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `fegetexceptflag()` function shall attempt to store an implementation-defined representation of the states of the floating-point status flags indicated by the argument `excepts` in the object pointed to by the argument `flagp`.

The `fesetexceptflag()` function shall attempt to set the floating-point status flags indicated by the argument `excepts` to the states stored in the object pointed to by `flagp`. The value pointed to by `flagp` shall have been set by a previous call to `fegetexceptflag()` whose second argument represented at least those floating-point exceptions represented by the argument `excepts`. This function does not raise floating-point exceptions, but only sets the state of the flags.

### RETURN VALUE

If the representation was successfully stored, `fegetexceptflag()` shall return zero. Otherwise, it shall return a non-zero value. If the `excepts` argument is zero or if all the specified exceptions were successfully set, `fesetexceptflag()` shall return zero. Otherwise, it shall return a non-zero value.

### ERRORS

No errors are defined.

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `feclearexcept()`
- `feraiseexcept()`
- `fetestexcept()`
- XBD `<fenv.h>`

## CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

## 1.37. fegetround, fesetround — get and set current rounding direction

---

### SYNOPSIS

```
#include <fenv.h>

int fegetround(void);
int fesetround(int round);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `fegetround()` function shall get the current rounding direction.

The `fesetround()` function shall establish the rounding direction represented by its argument `round`. If the argument is not equal to the value of a rounding direction macro, the rounding direction is not changed.

### RETURN VALUE

The `fegetround()` function shall return the value of the rounding direction macro representing the current rounding direction or a negative value if there is no such rounding direction macro or the current rounding direction is not determinable.

The `fesetround()` function shall return a zero value if and only if the requested rounding direction was established.

### ERRORS

No errors are defined.

---

*The following sections are informative.*

## EXAMPLES

The following example saves, sets, and restores the rounding direction, reporting an error and aborting if setting the rounding direction fails:

```
#include <fenv.h>
#include <assert.h>
void f(int round_dir)
{
    #pragma STDC FENV_ACCESS ON
    int save_round;
    int setround_ok;
    save_round = fegetround();
    setround_ok = fesetround(round_dir);
    assert(setround_ok == 0);
    /* ... */
    fesetround(save_round);
    /* ... */
}
```

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

XBD [`<fenv.h>`](#)

## CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

## 1.38. fetestexcept — test floating-point exception flags

---

### SYNOPSIS

```
#include <fenv.h>

int fetestexcept(int excepts);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `fetestexcept()` function shall determine which of a specified subset of the floating-point exception flags are currently set. The `excepts` argument specifies the floating-point status flags to be queried.

### RETURN VALUE

The `fetestexcept()` function shall return the value of the bitwise-inclusive OR of the floating-point exception macros corresponding to the currently set floating-point exceptions included in `excepts`.

### ERRORS

No errors are defined.

---

*The following sections are informative.*

### EXAMPLES

The following example calls function `f()` if an invalid exception is set, and then function `g()` if an overflow exception is set:

```
#include <fenv.h>
/* ... */
{
    #pragma STDC FENV_ACCESS ON
    int set_excepts;
    feclearexcept(FE_INVALID | FE_OVERFLOW);
    // maybe raise exceptions
    set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
    if (set_excepts & FE_INVALID) f();
    if (set_excepts & FE_OVERFLOW) g();
    /* ... */
}
```

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [feclearexcept\(\)](#)
- [fegetexceptflag\(\)](#)
- [feraiseexcept\(\)](#)

XBD [<fenv.h>](#)

## CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

## 1.39. feupdateenv — update floating-point environment

---

### SYNOPSIS

```
#include <fenv.h>
int feupdateenv(const fenv_t *envp);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `feupdateenv()` function shall attempt to save the currently raised floating-point exceptions in its automatic storage, attempt to install the floating-point environment represented by the object pointed to by `envp`, and then attempt to raise the saved floating-point exceptions. The argument `envp` shall point to an object set by a call to `feholdexcept()` or `fegetenv()`, or equal a floating-point environment macro.

### RETURN VALUE

The `feupdateenv()` function shall return a zero value if and only if all the required actions were successfully carried out.

### ERRORS

No errors are defined.

*The following sections are informative.*

### EXAMPLES

The following example shows sample code to hide spurious underflow floating-point exceptions:

```
#include <fenv.h>
double f(double x)
```

```
{  
    #pragma STDC FENV_ACCESS ON  
    double result;  
    fenv_t save_env;  
    feholdexcept(&save_env);  
    // compute result  
    if /* test spurious underflow */)  
        feclearexcept(FE_UNDERFLOW);  
    feupdateenv(&save_env);  
    return result;  
}
```

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [fegetenv\(\)](#)
- [feholdexcept\(\)](#)
- XBD [`<fenv.h>`](#)

## CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

## 1.40. `fflush` — flush a stream

---

### SYNOPSIS

```
#include <stdio.h>

int fflush(FILE *stream);
```

### DESCRIPTION

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

If `stream` points to an output stream or an update stream in which the most recent operation was not input, `fflush()` shall cause any unwritten data for that stream to be written to the file, and the last data modification and last file status change timestamps of the underlying file shall be marked for update.

For a stream open for reading with an underlying file description, if the file is not already at EOF, and the file is one capable of seeking, the file offset of the underlying open file description shall be set to the file position of the stream, and any characters pushed back onto the stream by `ungetc()` or `ungetwc()` that have not subsequently been read from the stream shall be discarded (without further changing the file offset).

If `stream` is a null pointer, `fflush()` shall perform this flushing action on all streams for which the behavior is defined above.

### RETURN VALUE

Upon successful completion, `fflush()` shall return 0; otherwise, it shall set the error indicator for the stream, return EOF, and set `errno` to indicate the error.

### ERRORS

The `fflush()` function shall fail if:

- `[EAGAIN]`

- The O\_NONBLOCK flag is set for the file descriptor underlying `stream` and the thread would be delayed in the write operation.

- **[EBADF]**

- The file descriptor underlying `stream` is not valid.

- **[EFBIG]**

- An attempt was made to write a file that exceeds the maximum file size.

- **[EFBIG]**

- An attempt was made to write a file that exceeds the file size limit of the process. A SIGXFSZ signal shall also be generated for the thread.

- **[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.

- **[EINTR]**

- The `fflush()` function was interrupted by a signal.

- **[EIO]**

- The process is a member of a background process group attempting to write to its controlling terminal, TOSTOP is set, the calling thread is not blocking SIGTTOU, the process is not ignoring SIGTTOU, and the process group of the process is orphaned. This error may also be returned under implementation-defined conditions.

- **[ENOMEM]**

- The underlying stream was created by `open_memstream()` or `open_wmemstream()` and insufficient memory is available.

- **[ENOSPC]**

- There was no free space remaining on the device containing the file or in the buffer used by the `fmemopen()` function.

- **[EPIPE]**

- An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A SIGPIPE signal shall also be sent to the thread.

The `fflush()` function may fail if:

- **[ENXIO]**

- A request was made of a nonexistent device, or the request was outside the capabilities of the device.

## EXAMPLES

### Sending Prompts to Standard Output

The following example uses `printf()` calls to print a series of prompts for information the user must enter from standard input. The `fflush()` calls force the output to standard output. The `fflush()` function is used because standard output is usually buffered and the prompt may not immediately be printed on the output or terminal. The `getline()` function calls read strings from standard input and place the results in variables, for use later in the program.

```

char *user;
char *oldpasswd;
char *newpasswd;
ssize_t llen;
size_t blen;
struct termios term;
tcflag_t saveflag;

printf("User name: ");
fflush(stdout);
blen = 0;
llen = getline(&user, &blen, stdin);
user[llen-1] = 0;
tcgetattr(fileno(stdin), &term);
saveflag = term.c_lflag;
term.c_lflag &= ~ECHO;
tcsetattr(fileno(stdin), TCSANOW, &term);
printf("Old password: ");
fflush(stdout);
blen = 0;
llen = getline(&oldpasswd, &blen, stdin);
oldpasswd[llen-1] = 0;

printf("\nNew password: ");
fflush(stdout);
blen = 0;
llen = getline(&newpasswd, &blen, stdin);
newpasswd[llen-1] = 0;
term.c_lflag = saveflag;
tcsetattr(fileno(stdin), TCSANOW, &term);
free(user);
free(oldpasswd);
free(newpasswd);

```

# APPLICATION USAGE

None.

## RATIONALE

Data buffered by the system may make determining the validity of the position of the current file descriptor impractical. Thus, enforcing the repositioning of the file descriptor after `fflush()` on streams open for `read()` is not mandated by POSIX.1-2024.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [2.5 Standard I/O Streams](#)
- `fmemopen()`
- `getrlimit()`
- `open_memstream()`
- 

## CHANGE HISTORY

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

### Issue 5

Large File Summit extensions are added.

### Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EFBIG] error is added as part of the large file support extensions.

- The [ENXIO] optional error condition is added.

The RETURN VALUE section is updated to note that the error indicator shall be set for the stream. This is for alignment with the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/31 is applied, updating the [EAGAIN] error in the ERRORS section from "the process would be delayed" to "the thread would be delayed".

## Issue 7

Austin Group Interpretation 1003.1-2001 #002 is applied, clarifying the interaction of file descriptors and streams.

The [ENOSPC] error condition is updated and the [ENOMEM] error is added from The Open Group Technical Standard, 2006, Extended API Set Part 1.

The EXAMPLES section is revised.

Changes are made related to support for finegrained timestamps.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0126 [87], XSH/TC1-2008/0127 [79], and XSH/TC1-2008/0128 [14] are applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0112 [816] and XSH/TC2-2008/0113 [626] are applied.

## Issue 8

Austin Group Defect 308 is applied, clarifying the handling of [EFBIG] errors.

Austin Group Defect 1669 is applied, removing XSI shading from part of the [EFBIG] error relating to the file size limit for the process.

## 1.41. fgetc — get a byte from a stream

---

### SYNOPSIS

```
#include <stdio.h>

int fgetc(FILE *stream);
```

### DESCRIPTION

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

If the end-of-file indicator for the input stream pointed to by `stream` is not set and a next byte is present, the `fgetc()` function shall obtain the next byte as an **unsigned char** converted to an **int**, from the input stream pointed to by `stream`, and advance the associated file position indicator for the stream (if defined). Since `fgetc()` operates on bytes, reading a character consisting of multiple bytes (or "a multi-byte character") may require multiple calls to `fgetc()`.

The `fgetc()` function may mark the last data access timestamp of the file associated with `stream` for update. The last data access timestamp shall be marked for update by the first successful execution of `fgetc()`, `fgets()`, `fread()`, `fscanf()`, `getc()`, `getchar()`, `getdelim()`, `getline()`, or `scanf()` using `stream` that returns data not supplied by a prior call to `ungetc()`.

### RETURN VALUE

Upon successful completion, `fgetc()` shall return the next byte from the input stream pointed to by `stream`. If the end-of-file indicator for the stream is set, or if the stream is at end-of-file, the end-of-file indicator for the stream shall be set and `fgetc()` shall return EOF. If an error occurs, the error indicator for the stream shall be set, `fgetc()` shall return EOF, and shall set `errno` to indicate the error.

# ERRORS

The `fgetc()` function shall fail if data needs to be read and:

- **[EAGAIN]** The `O_NONBLOCK` flag is set for the file descriptor underlying `stream` and the thread would be delayed in the `fgetc()` operation.
- **[EBADF]** The file descriptor underlying `stream` is not a valid file descriptor open for reading.
- **[EINTR]** The read operation was terminated due to the receipt of a signal, and no data was transferred.
- **[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 calling thread is blocking SIGTTIN or the process is ignoring SIGTTIN or the process group of the process is orphaned. This error may also be generated for implementation-defined reasons.
- **[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.

The `fgetc()` function may fail if:

- **[ENOMEM]** Insufficient storage space is available.
- **[ENXIO]** A request was made of a nonexistent device, or the request was outside the capabilities of the device.

# EXAMPLES

None.

# APPLICATION USAGE

If the integer value returned by `fgetc()` is stored into a variable of type `char` and then compared against the integer constant `EOF`, the comparison may never succeed, because sign-extension of a variable of type `char` on widening to integer is implementation-defined.

The `ferror()` or `feof()` functions must be used to distinguish between an error condition and an end-of-file condition.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- 2.5 Standard I/O Streams
- `feof()`
- `ferror()`
- `fgets()`
- `fread()`
- `fscanf()`
- `getchar()`
- `getc()`
- `ungetc()`
- XBD `<stdio.h>`

## CHANGE HISTORY

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

### Issue 5

Large File Summit extensions are added.

### Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EIO] and [EOVERFLOW] mandatory error conditions are added.

- The [ENOMEM] and [ENXIO] optional error conditions are added.

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

- The DESCRIPTION is updated to clarify the behavior when the end-of-file indicator for the input stream is not set.
- The RETURN VALUE section is updated to note that the error indicator shall be set for the stream.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/32 is applied, updating the [EAGAIN] error in the ERRORS section from "the process would be delayed" to "the thread would be delayed".

## Issue 7

Austin Group Interpretation 1003.1-2001 #051 is applied, updating the list of functions that mark the last data access timestamp for update.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0129 [79] and XSH/TC1-2008/0130 [14] are applied.

## Issue 8

Austin Group Defect 1330 is applied, removing obsolescent interfaces.

Austin Group Defect 1624 is applied, changing the RETURN VALUE section.

## 1.42. fgets

---

### SYNOPSIS

```
#include <stdio.h>

char *fgets(char *restrict s, int n, FILE *restrict stream);
```

### DESCRIPTION

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `fgets()` function shall read bytes from `stream` into the array pointed to by `s` until `n`-1 bytes are read, or a `<newline>` is read and transferred to `s`, or an end-of-file condition is encountered. A null byte shall be written immediately after the last byte read into the array. If the end-of-file condition is encountered before any bytes are read, the contents of the array pointed to by `s` shall not be changed.

The `fgets()` function may mark the last data access timestamp of the file associated with `stream` for update. The last data access timestamp shall be marked for update by the first successful execution of `fgetc()`, `fgets()`, `fread()`, `fscanf()`, `getc()`, `getchar()`, `getdelim()`, `getline()`, or `scanf()` using `stream` that returns data not supplied by a prior call to `ungetc()`.

### RETURN VALUE

Upon successful completion, `fgets()` shall return `s`. If the stream is at end-of-file, the end-of-file indicator for the stream shall be set and `fgets()` shall return a null pointer. If an error occurs, the error indicator for the stream shall be set, `fgets()` shall return a null pointer, and shall set `errno` to indicate the error.

### ERRORS

Refer to `fgetc()`.

# EXAMPLES

## Reading Input

The following example uses `fgets()` to read lines of input. It assumes that the file it is reading is a text file and that lines in this text file are no longer than 16384 (or `{LINE_MAX}`) if it is less than 16384 on the implementation where it is running) bytes long. (Note that the standard utilities have no line length limit if `sysconf(_SC_LINE_MAX)` returns -1 without setting `errno`. This example assumes that `sysconf(_SC_LINE_MAX)` will not fail.)

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

#define MYLIMIT 16384

char *line;
int line_max;

if (LINE_MAX >= MYLIMIT) {
    /* Use maximum line size of MYLIMIT. If LINE_MAX is
       bigger than our limit, sysconf() cannot report a
       smaller limit. */
    line_max = MYLIMIT;
} else {
    long limit = sysconf(_SC_LINE_MAX);
    line_max = (limit < 0 || limit > MYLIMIT) ? MYLIMIT : (int)limit;
}

/* line_max + 1 leaves room for the null byte added by fgets(). */
line = malloc(line_max + 1);
if (line == NULL) {
    /* out of space */
    ...
    return error;
}

while (fgets(line, line_max + 1, fp) != NULL) {
    /* Verify that a full line has been read ... */
    /* If not, report an error or prepare to treat the
       next time through the loop as a read of a
       continuation of the current line. */
    ...
    /* Process line ... */
    ...
}
```

```
free(line);
```

```
...
```

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- 2.5 Standard I/O Streams
- [fgetc\(\)](#)
- [fopen\(\)](#)
- [fread\(\)](#)
- [fscanf\(\)](#)
- [getc\(\)](#)
- [getchar\(\)](#)
- [getdelim\(\)](#)
- [ungetc\(\)](#)
- XBD [`<stdio.h>`](#)



# CHANGE HISTORY

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

## Issue 6

Extensions beyond the ISO C standard are marked.

The prototype for `fgets()` is changed for alignment with the ISO/IEC 9899:1999 standard.

## Issue 7

Austin Group Interpretation 1003.1-2001 #051 is applied, updating the list of functions that mark the last data access timestamp for update.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0134 [182] and XSH/TC1-2008/0135 [14] are applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0114 [468] is applied.

## Issue 8

Austin Group Defect 1330 is applied, removing obsolescent interfaces.

Austin Group Defect 1624 is applied, changing the RETURN VALUE section.

---

## 1.43. fileno — map a stream pointer to a file descriptor

---

### SYNOPSIS

```
#include <stdio.h>

int fileno(FILE *stream);
```

### DESCRIPTION

The `fileno()` function shall return the integer file descriptor associated with the stream pointed to by `stream`.

### RETURN VALUE

Upon successful completion, `fileno()` shall return the integer value of the file descriptor associated with `stream`. Otherwise, the value -1 shall be returned and `errno` set to indicate the error.

### ERRORS

The `fileno()` function shall fail if:

- **[EBADF]**

The stream is not associated with a file.

The `fileno()` function may fail if:

- **[EBADF]**

The file descriptor underlying `stream` is not a valid file descriptor.

### EXAMPLES

None.

### APPLICATION USAGE

None.

## RATIONALE

Without some specification of which file descriptors are associated with these streams, it is impossible for an application to set up the streams for another application it starts with `fork()` and `exec`. In particular, it would not be possible to write a portable version of the `sh` command interpreter (although there may be other constraints that would prevent that portability).

## FUTURE DIRECTIONS

None.

## SEE ALSO

- *2.5.1 Interaction of File Descriptors and Standard I/O Streams*
- `dirfd()`
- `fdopen()`
- `fopen()`
- `stdin`
- **XBD** `<stdio.h>`

## CHANGE HISTORY

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

### Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EBADF] optional error condition is added.

### Issue 7

SD5-XBD-ERN-99 is applied, changing the definition of the [EBADF] error.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0115 [589] is applied.

---

## 1.44. flockfile, ftrylockfile, funlockfile — stdio locking functions

---

### SYNOPSIS

```
#include <stdio.h>

void flockfile(FILE *file);
int ftrylockfile(FILE *file);
void funlockfile(FILE *file);
```

### DESCRIPTION

These functions shall provide for explicit application-level locking of the locks associated with standard I/O streams (see 2.5 Standard I/O Streams). These functions can be used by a thread to delineate a sequence of I/O statements that are executed as a unit.

The `flockfile()` function shall acquire for a thread ownership of a `(FILE *)` object.

The `ftrylockfile()` function shall acquire for a thread ownership of a `(FILE *)` object if the object is available; `ftrylockfile()` is a non-blocking version of `flockfile()`.

The `funlockfile()` function shall relinquish the ownership granted to the thread. The behavior is undefined if a thread other than the current owner calls the `funlockfile()` function.

The functions shall behave as if there is a lock count associated with each `(FILE *)` object. This count is implicitly initialized to zero when the `(FILE *)` object is created. The `(FILE *)` object is unlocked when the count is zero. When the count is positive, a single thread owns the `(FILE *)` object. When the `flockfile()` function is called, if the count is zero or if the count is positive and the caller owns the `(FILE *)` object, the count shall be incremented. Otherwise, the calling thread shall be suspended, waiting for the count to return to zero. Each call to `funlockfile()` shall decrement the count. This allows matching calls to `flockfile()` (or successful calls to `ftrylockfile()`) and `funlockfile()` to be nested.

## RETURN VALUE

None for `flockfile()` and `funlockfile()`.

The `ftrylockfile()` function shall return zero for success and non-zero to indicate that the lock cannot be acquired.

## ERRORS

No errors are defined.

## APPLICATION USAGE

Applications using these functions may be subject to priority inversion, as discussed in XBD 3.275 Priority Inversion.

A call to `exit()` can block until locked streams are unlocked because a thread having ownership of a (`FILE*`) object blocks all function calls that reference that (`FILE*`) object (except those with names ending in `_unlocked`) from other threads, including calls to `exit()`.

Note: a `FILE` lock is not a file lock (see XBD 3.143 File Lock).

## RATIONALE

The `flockfile()` and `funlockfile()` functions provide an orthogonal mutual-exclusion lock for each `FILE`. The `ftrylockfile()` function provides a non-blocking attempt to acquire a `FILE` lock, analogous to `pthread_mutex_trylock()`.

These locks behave as if they are the same as those used internally by `stdio` for thread-safety. This both provides thread-safety of these functions without requiring a second level of internal locking and allows functions in `stdio` to be implemented in terms of other `stdio` functions.

Application developers and implementors should be aware that there are potential deadlock problems on `FILE` objects. For example, the line-buffered flushing semantics of `stdio` (requested via `{_IOLBF}`) require that certain input operations sometimes cause the buffered contents of implementation-defined line-buffered output streams to be flushed. If two threads each hold the lock on the other's `FILE`, deadlock ensues. This type of deadlock can be avoided by acquiring `FILE` locks in a consistent order. In particular, the line-buffered output stream deadlock can

typically be avoided by acquiring locks on input streams before locks on output streams if a thread would be acquiring both.

In summary, threads sharing stdio streams with other threads can use `flockfile()` and `funlockfile()` to cause sequences of I/O performed by a single thread to be kept bundled. The only case where the use of `flockfile()` and `funlockfile()` is required is to provide a scope protecting uses of the `*_unlocked` functions/macros. This moves the cost/performance tradeoff to the optimal point.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `exit()`
- `getc_unlocked()`
- XBD 3.275 Priority Inversion
- `<stdio.h>`

## CHANGE HISTORY

### First released in Issue 5

Included for alignment with the POSIX Threads Extension.

### Issue 6

These functions are marked as part of the Thread-Safe Functions option.

### Issue 7

The `flockfile()`, `ftrylockfile()`, and `funlockfile()` functions are moved from the Thread-Safe Functions option to the Base.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0140 [118] is applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0116 [611] is applied.

## Issue 8

Austin Group Defect 1118 is applied, clarifying that a FILE lock is not a file lock.

Austin Group Defect 1302 is applied, replacing parts of the text with a reference to 2.5 Standard I/O Streams.

## 1.45. `fopen` — open a stream

---

### SYNOPSIS

```
#include <stdio.h>

FILE *fopen(const char *restrict pathname, const char *restrict mode)
```

### DESCRIPTION

Except for the "exclusive access" requirement (see below), the functionality described on this reference page is aligned with the ISO C standard. Any other conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard for all `fopen()` functionality except in relation to "exclusive access".

The `fopen()` function shall open the file whose pathname is the string pointed to by `pathname`, and associates a stream with it.

The `mode` argument points to a character string. The behavior is unspecified if any character occurs more than once in the string. If the string begins with one of the following characters, then the file shall be opened in the indicated mode. Otherwise, the behavior is undefined.

- `'r'`: Open file for reading.
- `'w'`: Truncate to zero length or create file for writing.
- `'a'`: Append; open or create file for writing at end-of-file.

The remainder of the string can contain any of the following characters, in any order, and further affect how the file is opened:

- `'b'`: This character shall have no effect, but is allowed for ISO C standard conformance.
- `'e'`: The underlying file descriptor shall have the `FD_CLOEXEC` flag atomically set.
- `'x'`: If the first character of `mode` is `'w'` or `'a'`, then the function shall fail if the file already exists, or cannot be created; if the file does not exist and can be created, it shall be created with an implementation-defined form of exclusive (also known as non-shared) access, if supported by the underlying file system, provided the resulting file permissions are the same as they would be without

the 'x' modifier. If the first character of mode is 'r', the effect is implementation-defined.

**Note:**

The ISO C standard requires exclusive access "to the extent that the underlying file system supports exclusive access", but does not define what it means by this. Taken at face value—that systems must do whatever they are capable of, at the file system level, in order to exclude access by others—this would require POSIX.1 systems to set the file permissions in a way that prevents access by other users and groups. Consequently, this volume of POSIX.1-2024 does not defer to the ISO C standard as regards the "exclusive access" requirement.

- '+': The file shall be opened for update (both reading and writing), rather than just reading or just writing.

Opening a file with read mode ('r' as the first character in the `mode` argument) shall fail if the file does not exist or cannot be read.

Opening a file with append mode ('a' as the first character in the `mode` argument) shall cause all subsequent writes to the file to be forced to the then current end-of-file, regardless of intervening calls to `fseek()`.

When a file is opened with update mode ('+' in the `mode` argument), both input and output can be performed on the associated stream. However, the application shall ensure that output is not directly followed by input without an intervening call to `fflush()` or to a file positioning function (`fseek()`, `fsetpos()`, or `rewind()`), and input is not directly followed by output without an intervening call to a file positioning function, unless the input operation encounters end-of-file.

When opened, a stream is fully buffered if and only if it can be determined not to refer to an interactive device. The error and end-of-file indicators for the stream shall be cleared.

If the first character in `mode` is 'w' or 'a' and the file did not previously exist, upon successful completion, `fopen()` shall mark for update the last data access, last data modification, and last file status change timestamps of the file and the last file status change and last data modification timestamps of the parent directory.

If the first character in `mode` is 'w' or 'a' and the file did not previously exist, the `fopen()` function shall create a file as if it called the `open()` function with a value appropriate for the path argument interpreted from `pathname`, a value for the oflag argument as specified below, and a value of `S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH` for the third argument.

If the first character in `mode` is 'w' and the file did previously exist, upon successful completion, `fopen()` shall mark for update the last data modification and last file status change timestamps of the file.

After a successful call to the `fopen()` function, the orientation of the stream shall be cleared, the encoding rule shall be cleared, and the associated `mbstate_t` object shall be set to describe an initial conversion state.

The file descriptor associated with the opened stream shall be allocated and opened as if by a call to `open()` using the following flags:

<code>fopen()</code> Mode First Character	<code>fopen()</code> Mode Includes '+'	Initial <code>open()</code> Flags
'r'	no	<code>O_RDONLY</code>
'w'	no	<code>O_WRONLY</code>
'a'	no	<code>O_WRONLY</code>
'r'	yes	<code>O_RDWR</code>
'w'	yes	<code>O_RDWR</code>
'a'	yes	<code>O_RDWR</code>

If, and only if, the 'e' mode string character is specified, the `O_CLOEXEC` flag shall be OR'ed into the initial `open()` flags specified in the above table.

If, and only if, the 'x' mode string character is specified together with either 'w' or 'a', the `O_EXCL` flag shall be OR'ed into the initial `open()` flags specified in the above table.

If mode includes 'x' and the underlying file system supports exclusive access (see above) enabled by the use of implementation-specific flags to `open()`, then the behavior shall be as if those flags are also included.

When using mode strings specified by this standard, the implementation shall behave as if no other flags had been passed to `open()`.

## RETURN VALUE

Upon successful completion, `fopen()` shall return a pointer to the object controlling the stream. Otherwise, a null pointer shall be returned, and `errno` shall be set to indicate the error.

# ERRORS

The `fopen()` function shall fail if:

- **[EACCES]**: Search permission is denied on a component of the path prefix, or the file exists and the permissions specified by mode are denied, or the file does not exist and write permission is denied for the parent directory of the file to be created.
- **[EEXIST]**: The mode argument begins with w or a and includes x, but the file already exists.
- **[EILSEQ]**: The mode argument begins with w or a, the file did not previously exist, and the last pathname component is not a portable filename and cannot be created in the target directory.
- **[EINTR]**: A signal was caught during `fopen()`.
- **[EISDIR]**: The named file is a directory and mode requires write access.
- **[ELOOP]**: A loop exists in symbolic links encountered during resolution of the pathname argument.
- **[EMFILE]**: All file descriptors available to the process are currently open.
- **[EMFILE]**: {STREAM\_MAX} streams are currently open in the calling process.
- **[ENAMETOOLONG]**: The length of a pathname exceeds {PATH\_MAX}, or pathname resolution of a symbolic link produced an intermediate result with a length that exceeds {PATH\_MAX}.
- **[ENFILE]**: The maximum allowable number of files is currently open in the system.
- **[ENOENT]**: The mode string begins with 'r' and a component of pathname does not name an existing file, or mode begins with 'w' or 'a' and a component of the path prefix of pathname does not name an existing file, or pathname is an empty string.
- **[ENOENT] or [ENOTDIR]**: The pathname argument contains at least one non-\ character and ends with one or more trailing \ characters. If pathname without the trailing \ characters would name an existing file, an [ENOENT] error shall not occur.
- **[ENOSPC]**: The directory or file system that would contain the new file cannot be expanded, the file does not exist, and the file was to be created.
- **[ENOTDIR]**: A component of the path prefix names an existing file that is neither a directory nor a symbolic link to a directory, or the pathname argument contains at least one non-\ character and ends with one or more

trailing \ characters and the last pathname component names an existing file that is neither a directory nor a symbolic link to a directory.

- **[ENXIO]**: The named file is a character special or block special file, and the device associated with this special file does not exist.
- **[EOVERFLOW]**: The named file is a regular file and the size of the file cannot be represented correctly in an object of type off\_t.
- **[EROFS]**: The named file resides on a read-only file system and mode requires write access.

The `fopen()` function may fail if:

- **[EINVAL]**: The value of the mode argument is not valid.
- **[ELOOP]**: More than {SYMLOOP\_MAX} symbolic links were encountered during resolution of the pathname argument.
- **[EMFILE]**: {FOPEN\_MAX} streams are currently open in the calling process.
- **[ENAMETOOLONG]**: The length of a component of a pathname is longer than {NAME\_MAX}.
- **[ENOMEM]**: Insufficient storage space is available.
- **[ETXTBSY]**: The file is a pure procedure (shared text) file that is being executed and mode requires write access.

## EXAMPLES

### Opening a File

The following example tries to open the file named file for reading. The `fopen()` function returns a file pointer that is used in subsequent `fgets()` and `fclose()` calls. If the program cannot open the file, it just ignores it.

```
#include <stdio.h>
...
FILE *fp;
...
void rgrep(const char *file)
{
...
    if ((fp = fopen(file, "r")) == NULL)
        return;
...
}
```

## APPLICATION USAGE

If an application needs to create a file in a way that fails if the file already exists, and either requires that it does not have exclusive access to the file or does not need exclusive access, it should use `open()` with the `O_CREAT` and `O_EXCL` flags instead of using `fopen()` with an 'x' in the mode. A stream can then be created, if needed, by calling `fdopen()` on the file descriptor returned by `open()`.

## RATIONALE

The 'e' mode character is provided as a convenience to avoid a data race in multi-threaded applications. Without it, a file descriptor is leaked into a child process created by one thread in the window between another thread creating a file descriptor with `fopen()` and then using `fileno()` and `fcntl()` to set the `FD_CLOEXEC` flag. It is also possible to avoid the race by using `open()` with `O_CLOEXEC` followed by `fdopen()`, however, there is no safe alternative for the `freopen()` function, and consistency dictates that the 'e' modifier should be standardized for all functions that accept mode strings.

The ISO C standard only recognizes the '+', 'b', and 'x' characters in certain positions of the mode string, leaving other arrangements as unspecified, and only permits 'x' in mode strings beginning with 'w'. This standard specifically requires support for all characters other than the first in the mode string to be recognized in any order. Thus, "wx" and "wx" behave the same, and while "wx+" is unspecified in the ISO C standard, this standard requires it to have the same behavior as "w+x". This standard also requires that 'x' work for mode strings beginning with 'a', as well as having implementation-defined behavior for mode strings beginning with 'r'. Therefore, while `open()` has undefined behavior if `O_EXCL` is specified without `O_CREAT`, the same is not true of `fopen()`.

When 'x' is in mode, the ISO C standard requires that the file is created with exclusive access to the extent that the underlying system supports exclusive access. Although POSIX.1 does not specify any method of enabling exclusive access, it allows for the existence of an implementation-specific flag, or flags, that enable it. Note that they should be file creation flags if a file is being created, not file access mode flags (that is, ones that are included in `O_ACCMODE`) or file status flags, so that they do not affect the value returned by `fcntl()` with `F_GETFL`. On implementations that have such flags, if support for them is file system dependent and exclusive access is requested when using `fopen()` to create a file on a file system that does not support it, the flags must not be used if they would cause `fopen()` to fail.

Some implementations support mandatory file locking as a means of enabling exclusive access to a file. Locks are set in the normal way, but instead of only preventing others from setting conflicting locks they prevent others from accessing the contents of the locked part of the file in a way that conflicts with the lock. However, unless the implementation has a way of setting a whole-file write lock on file creation, this does not satisfy the requirement in the ISO C standard that the file is "created with exclusive access to the extent that the underlying system supports exclusive access". (Having `fopen()` create the file and set a lock on the file as two separate operations is not the same, and it would introduce a race condition whereby another process could open the file and write to it (or set a lock) in between the two operations.) However, on all implementations that support mandatory file locking, its use is discouraged; therefore, it is recommended that implementations which support mandatory file locking do not add a means of creating a file with a whole-file exclusive lock set, so that `fopen()` is not required to enable mandatory file locking in order to conform to the ISO C standard. An implementation that has a means of creating a file with a whole-file exclusive lock set would need to provide a way to change the behavior of `fopen()` depending on whether the calling process is executing in a POSIX.1 conforming environment or an ISO C conforming environment.

The typical implementation-defined behavior for mode "rx" is to ignore the 'x', but the standard developers did not wish to mandate this behavior. For example, an implementation could allow shared access for reading; that is, disallow a file that has been opened this way from also being opened for writing.

Implementations are encouraged to have `fopen()` and `freopen()` report an [EILSEQ] error if mode begins with 'w' or 'a', the file did not previously exist, and the last component of pathname contains any bytes that have the encoded value of a \ character.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- **2.5 Standard I/O Streams**
- `creat()`
- `fclose()`
- `fdopen()`
- `fmemopen()`

- `freopen()`
- `open_memstream()`
- 

## CHANGE HISTORY

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

### Issue 5

Large File Summit extensions are added.

### Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, text is added to indicate setting of the offset maximum in the open file description. This change is to support large files.
- In the ERRORS section, the [EOVERFLOW] condition is added. This change is to support large files.
- The [ELOOP] mandatory error condition is added.
- The [EINVAL], [EMFILE], [ENAMETOOLONG], [ENOMEM], and [ETXTBSY] optional error conditions are added.

The normative text is updated to avoid use of the term "must" for application requirements.

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

- The prototype for `fopen()` is updated.
- The DESCRIPTION is updated to note that if the argument mode points to a string other than those listed, then the behavior is undefined.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

## Issue 7

Austin Group Interpretation 1003.1-2001 #025 is applied, clarifying the file creation mode.

Austin Group Interpretation 1003.1-2001 #143 is applied.

Austin Group Interpretation 1003.1-2001 #159 is applied, clarifying requirements for the flags set on the open file description.

SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

SD5-XSH-ERN-149 is applied, changing the {STREAM\_MAX} [EMFILE] error condition from a "may fail" to a "shall fail".

Changes are made related to support for finegrained timestamps.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0156 [291,433], XSH/TC1-2008/0157 [146,433], XSH/TC1-2008/0158 [324], and XSH/TC1-2008/0159 [14] are applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0122 [822] is applied.

## Issue 8

Austin Group Defect 251 is applied, encouraging implementations to disallow the creation of filenames containing any bytes that have the encoded value of a \ character.

Austin Group Defect 293 is applied, adding the [EILSEQ] error.

Austin Group Defects 411 and 1524 are applied, adding the 'e' and 'x' mode string characters.

Austin Group Defect 1200 is applied, correcting the argument name in the [ELOOP] errors.

Austin Group Defect 1302 is applied, aligning this function with the ISO/IEC 9899:2018 standard.

---

## 1.46. fprintf

### SYNOPSIS

```
#include <stdio.h>

/* [CX] */ int asprintf(char **restrict ptr, const char *restrict
/* [CX] */ int dprintf(int fildes, const char *restrict format, ..

int fprintf(FILE *restrict stream, const char *restrict format, ..
int printf(const char *restrict format, ...);
int snprintf(char *restrict s, size_t n,
             const char *restrict format, ...);
int sprintf(char *restrict s, const char *restrict format, ...);
```

### DESCRIPTION

*/ [CX] /* Except for asprintf(), dprintf(), and the behavior of the %lc conversion when passed a null wide character, the functionality described on this reference page is aligned with the ISO C standard. Any other conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard for all fprintf(), printf(), snprintf(), and sprintf() functionality except in relation to the %lc conversion when passed a null wide character.

The fprintf() function shall place output on the named output stream. The printf() function shall place output on the standard output stream stdout. The sprintf() function shall place output followed by the null byte, '0', in consecutive bytes starting at \*s; it is the user's responsibility to ensure that enough space is available.

*/ [CX] /* The asprintf() function shall be equivalent to sprintf(), except that the output string shall be written to dynamically allocated memory, allocated as if by a call to malloc(), of sufficient length to hold the resulting string, including a terminating null byte. If the call to asprintf() is successful, the address of this dynamically allocated string shall be stored in the location referenced by ptr.

*/ [CX] /* The dprintf() function shall be equivalent to the fprintf() function, except that dprintf() shall write output to the file associated with the file descriptor specified by the fildes argument rather than place output on a stream.

The snprintf() function shall be equivalent to sprintf(), with the addition of the n argument which limits the number of bytes written to the buffer referred to by s. If n is zero, nothing shall be written and s may be a null pointer. Otherwise, output

bytes beyond the  $n-1$ st shall be discarded instead of being written to the array, and a null byte is written at the end of the bytes actually written into the array.

If copying takes place between objects that overlap as a result of a call to `sprintf()` or `snprintf()`, the results are undefined.

Each of these functions converts, formats, and prints its arguments under control of the format. The application shall ensure that the format is a character string, beginning and ending in its initial shift state, if any. The format is composed of zero or more directives: ordinary characters, which are simply copied to the output stream, and conversion specifications, each of which shall result in the fetching of zero or more arguments. The results are undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments shall be evaluated but are otherwise ignored.

*/ [CX] /* Conversions can be applied to the  $n$ th argument after the format in the argument list, rather than to the next unused argument. In this case, the conversion specifier character `%` (see below) is replaced by the sequence `"%n$"`, where  $n$  is a decimal integer in the range  $[1, \{NL\_ARGMAX\}]$ , giving the position of the argument in the argument list. This feature provides for the definition of format strings that select arguments in an order appropriate to specific languages (see the EXAMPLES section).

The format can contain either numbered argument conversion specifications (that is, those introduced by `"%n$"` and optionally containing the `"*m$"` forms of field width and precision), or unnumbered argument conversion specifications (that is, those introduced by the `%` character and optionally containing the `*` form of field width and precision), but not both. The only exception to this is that `%%` can be mixed with the `"%n$"` form. The results of mixing numbered and unnumbered argument specifications in a format string are undefined. When numbered argument specifications are used, specifying the  $N$ th argument requires that all the leading arguments, from the first to the  $(N-1)$ th, are specified in the format string.

*/ [CX] /* In format strings containing the `"%n$"` form of conversion specification, numbered arguments in the argument list can be referenced from the format string as many times as required.

In format strings containing the `%` form of conversion specification, each conversion specification uses the first unused argument in the argument list.

*/ [CX] /* All forms of the `fprintf()` functions allow for the insertion of a language-dependent radix character in the output string. The radix character is defined in the current locale (category `LC_NUMERIC`). In the `POSIX` locale, or in a locale where the radix character is not defined, the radix character shall default to a `('.)'`.

Each conversion specification is introduced by the `'%` character */ [CX] /* or by the character sequence `"%n$"`, after which the following appear in sequence:

- Zero or more flags (in any order), which modify the meaning of the conversion specification.
- An optional minimum field width. If the converted value has fewer bytes than the field width, it shall be padded with characters by default on the left; it shall be padded on the right if the left-adjustment flag ('-'), described below, is given to the field width. The field width takes the form of an ('), / [CX] / or in conversion specifications introduced by "%n\$" the "m\$" string, described below, or a decimal integer.
- An optional precision that gives the minimum number of digits to appear for the d, i, o, u, x, and X conversion specifiers; the number of digits to appear after the radix character for the a, A, e, E, f, and F conversion specifiers; the maximum number of significant digits for the g and G conversion specifiers; or the maximum number of bytes to be printed from a string in the s / [XSI] / and S conversion specifiers. The precision takes the form of a ('.) followed either by an ('), / [CX] / or in conversion specifications introduced by "%n\$" the "m\$" string, described below, or an optional decimal digit string, where a null digit string is treated as zero. If a precision appears with any other conversion specifier, the behavior is undefined.
- An optional length modifier that specifies the size of the argument.
- A conversion specifier character that indicates the type of conversion to be applied.

A field width, or precision, or both, may be indicated by an ('). *In this case an argument of type int supplies the field width or precision. Applications shall ensure that arguments specifying field width, or precision, or both appear in that order before the argument, if any, to be converted. A negative field width is taken as a '-' flag followed by a positive field width. A negative precision is taken as if the precision were omitted.* / [CX] / *In format strings containing conversion specifications introduced by "%n\$", in addition to being indicated by the decimal digit string, a field width may be indicated by the sequence "m\$" and precision by the sequence ".\*m\$", where m is a decimal integer in the range [1,{NL\_ARGMAX}] giving the position in the argument list (after the format argument) of an integer argument containing the field width or precision, for example:*

```
printf("%1$d:%2$.*3$d:%4$.*3$d\n", hour, min, precision, sec);
```

## Flag Characters

The flag characters and their meanings are:

**'** (apostrophe)

**/ [CX] /** The integer portion of the result of a decimal conversion (%i, %d, %u, %f, %F, %g, or %G) shall be formatted with thousands' grouping characters. For other conversions the behavior is undefined. The non-monetary grouping character is used.

**-**

The result of the conversion shall be left-justified within the field. The conversion is right-justified if this flag is not specified.

**+**

The result of a signed conversion shall always begin with a sign ('+' or '-'). The conversion shall begin with a sign only when a negative value is converted if this flag is not specified.

If the first character of a signed conversion is not a sign or if a signed conversion results in no characters, a shall be prefixed to the result. This means that if the and '+' flags both appear, the flag shall be ignored.

**#**

Specifies that the value is to be converted to an alternative form. For o conversion, it shall increase the precision, if and only if necessary, to force the first digit of the result to be a zero (if the value and precision are both 0, a single 0 is printed). For x or X conversion specifiers, a non-zero result shall have 0x (or 0X) prefixed to it. For a, A, e, E, f, F, g, and G conversion specifiers, the result shall always contain a radix character, even if no digits follow the radix character. Without this flag, a radix character appears in the result of these conversions only if a digit follows it. For g and G conversion specifiers, trailing zeros shall not be removed from the result as they normally are. For other conversion specifiers, the behavior is undefined.

**0**

For d, i, o, u, x, X, a, A, e, E, f, F, g, and G conversion specifiers, leading zeros (following any indication of sign or base) are used to pad to the field width rather than performing space padding, except when converting an infinity or NaN. If the '0' and '-' flags both appear, the '0' flag is ignored. For d, i, o, u, x, and X conversion specifiers, if a precision is specified, the '0' flag shall be ignored. **/ [CX] /** If the '0' and flags both appear, the grouping characters are inserted before zero padding. For other conversions, the behavior is undefined.

## Length Modifiers

The length modifiers and their meanings are:

**hh**

Specifies that a following d, i, o, u, x, or X conversion specifier applies to a signed

char or unsigned char argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to signed char or unsigned char before printing); or that a following n conversion specifier applies to a pointer to a signed char argument.

## **h**

Specifies that a following d, i, o, u, x, or X conversion specifier applies to a short or unsigned short argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to short or unsigned short before printing); or that a following n conversion specifier applies to a pointer to a short argument.

## **I (ell)**

Specifies that a following d, i, o, u, x, or X conversion specifier applies to a long or unsigned long argument; that a following n conversion specifier applies to a pointer to a long argument; that a following c conversion specifier applies to a wint\_t argument; that a following s conversion specifier applies to a pointer to a wchar\_t argument; or has no effect on a following a, A, e, E, f, F, g, or G conversion specifier.

## **II (ell-ell)**

Specifies that a following d, i, o, u, x, or X conversion specifier applies to a long long or unsigned long long argument; or that a following n conversion specifier applies to a pointer to a long long argument.

## **j**

Specifies that a following d, i, o, u, x, or X conversion specifier applies to an intmax\_t or uintmax\_t argument; or that a following n conversion specifier applies to a pointer to an intmax\_t argument.

## **z**

Specifies that a following d, i, o, u, x, or X conversion specifier applies to a size\_t or the corresponding signed integer type argument; or that a following n conversion specifier applies to a pointer to a signed integer type corresponding to a size\_t argument.

## **t**

Specifies that a following d, i, o, u, x, or X conversion specifier applies to a ptrdiff\_t or the corresponding unsigned type argument; or that a following n conversion specifier applies to a pointer to a ptrdiff\_t argument.

## **L**

Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to a long double argument.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

## Conversion Specifiers

The conversion specifiers and their meanings are:

### **d, i**

The int argument shall be converted to a signed decimal in the style "[-]dddd". The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1. The result of converting zero with an explicit precision of zero shall be no characters.

### **o**

The unsigned argument shall be converted to unsigned octal format in the style "ddddd". The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1. The result of converting zero with an explicit precision of zero shall be no characters.

### **u**

The unsigned argument shall be converted to unsigned decimal format in the style "ddddd". The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1. The result of converting zero with an explicit precision of zero shall be no characters.

### **x**

The unsigned argument shall be converted to unsigned hexadecimal format in the style "ddddd"; the letters "abcdef" are used. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1. The result of converting zero with an explicit precision of zero shall be no characters.

### **X**

Equivalent to the x conversion specifier, except that letters "ABCDEF" are used instead of "abcdef".

### **f, F**

The double argument shall be converted to decimal notation in the style "[-]ddd.ddd", where the number of digits after the radix character is equal to the precision specification. If the precision is missing, it shall be taken as 6; if the precision is explicitly zero and no '#' flag is present, no radix character shall appear. If a radix character appears, at least one digit appears before it. The low-order digit shall be rounded in an implementation-defined manner.

A double argument representing an infinity shall be converted in one of the styles "[-]inf" or "[-]infinity"; which style is implementation-defined. A double argument representing a NaN shall be converted in one of the styles "[-]nan(n-char-

sequence)" or "[-]nan"; which style, and the meaning of any n-char-sequence, is implementation-defined. The F conversion specifier produces "INF", "INFINITY", or "NAN" instead of "inf", "infinity", or "nan", respectively.

### e, E

The double argument shall be converted in the style "[-]d.ddd e $\pm$ dd", where there is one digit before the radix character (which is non-zero if the argument is non-zero) and the number of digits after it is equal to the precision; if the precision is missing, it shall be taken as 6; if the precision is zero and no '#' flag is present, no radix character shall appear. The low-order digit shall be rounded in an implementation-defined manner. The E conversion specifier shall produce a number with 'E' instead of 'e' introducing the exponent. The exponent shall always contain at least two digits. If the value is zero, the exponent shall be zero.

A double argument representing an infinity or NaN shall be converted in the style of an f or F conversion specifier.

### g, G

The double argument representing a floating-point number shall be converted in the style f or e (or in the style F or E in the case of a G conversion specifier), depending on the value converted and the precision. Let P equal the precision if non-zero, 6 if the precision is omitted, or 1 if the precision is zero. Then, if a conversion with style E would have an exponent of X:

- If  $P > X \geq -4$ , the conversion shall be with style f (or F) and precision  $P - (X + 1)$ .
- Otherwise, the conversion shall be with style e (or E) and precision P-1.

Finally, unless the '#' flag is used, any trailing zeros shall be removed from the fractional portion of the result and the decimal-point character shall be removed if there is no fractional portion remaining.

A double argument representing an infinity or NaN shall be converted in the style of an f or F conversion specifier.

### a, A

A double argument representing a floating-point number shall be converted in the style "[-]0x h.ffff p $\pm$ d", where there is one hexadecimal digit (which shall be non-zero if the argument is a normalized floating-point number and is otherwise unspecified) before the decimal-point character and the number of hexadecimal digits after it is equal to the precision; if the precision is missing and FLT\_RADIX is a power of 2, then the precision shall be sufficient for an exact representation of the value; if the precision is missing and FLT\_RADIX is not a power of 2, then the precision shall be sufficient to distinguish values of type double, except that trailing zeros may be omitted; if the precision is zero and the '#' flag is not specified, no decimal-point character shall appear. The letters "abcdef" shall be used for a conversion and the letters "ABCDEF" for A conversion. The A conversion specifier produces a number with 'X' and 'P' instead of 'x' and 'p'. The exponent shall always

contain at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent shall be zero.

A double argument representing an infinity or NaN shall be converted in the style of an f or F conversion specifier.

**c**

The int argument shall be converted to an unsigned char, and the resulting byte shall be written.

If an l (ell) qualifier is present, / [CX] / the wint\_t argument shall be converted to a multi-byte sequence as if by a call to wcrtomb() with a pointer to storage of at least MB\_CUR\_MAX bytes, the wint\_t argument converted to wchar\_t, and an initial shift state, and the resulting byte(s) written.

**s**

The argument shall be a pointer to an array of char. Bytes from the array shall be written up to (but not including) any terminating null byte. If the precision is specified, no more than that many bytes shall be written. If the precision is not specified or is greater than the size of the array, the application shall ensure that the array contains a null byte.

If an l (ell) qualifier is present, the argument shall be a pointer to an array of type wchar\_t. Wide characters from the array shall be converted to characters (each as if by a call to the wcrtomb() function, with the conversion state described by an mbstate\_t object initialized to zero before the first wide character is converted) up to and including a terminating null wide character. The resulting characters shall be written up to (but not including) the terminating null character (byte). If no precision is specified, the application shall ensure that the array contains a null wide character. If a precision is specified, no more than that many characters (bytes) shall be written (including shift sequences, if any), and the array shall contain a null wide character if, to equal the character sequence length given by the precision, the function would need to access a wide character one past the end of the array. In no case shall a partial character be written.

**p**

The argument shall be a pointer to void. The value of the pointer is converted to a sequence of printable characters, in an implementation-defined manner.

**n**

The argument shall be a pointer to an integer into which is written the number of bytes written to the output so far by this call to one of the fprintf() functions. No argument is converted.

**C**

/ [XSI] / Equivalent to lc.

**S**

*I [XSI]* / Equivalent to ls.

%

Write a '%' character; no argument shall be converted. The application shall ensure that the complete conversion specification is %%.

If a conversion specification does not match one of the above forms, the behavior is undefined. If any argument is not the correct type for the corresponding conversion specification, the behavior is undefined.

In no case shall a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field shall be expanded to contain the conversion result. Characters generated by fprintf() and printf() are printed as if fputc() had been called.

For the a and A conversion specifiers, if FLT\_RADIX is a power of 2, the value shall be correctly rounded to a hexadecimal floating number with the given precision.

For a and A conversions, if FLT\_RADIX is not a power of 2 and the result is not exactly representable in the given precision, the result should be one of the two adjacent numbers in hexadecimal floating style with the given precision, with the extra stipulation that the error should have a correct sign for the current rounding direction.

For the e, E, f, F, g, and G conversion specifiers, if the number of significant decimal digits is at most DECIMAL\_DIG, then the result should be correctly rounded. If the number of significant decimal digits is more than DECIMAL\_DIG but the source value is exactly representable with DECIMAL\_DIG digits, then the result should be an exact representation with trailing zeros. Otherwise, the source value is bounded by two adjacent decimal strings L < U, both having DECIMAL\_DIG significant digits; the value of the resultant decimal string D should satisfy L <= D <= U, with the extra stipulation that the error should have a correct sign for the current rounding direction.

*I [CX]* / The last data modification and last file status change timestamps of the file shall be marked for update:

1. Between the call to a successful execution of fprintf() or printf() and the next successful completion of a call to fflush() or fclose() on the same stream or a call to exit() or abort()
2. Upon successful completion of a call to dprintf()

## RETURN VALUE

Upon successful completion, the / [CX] / `dprintf()`, `fprintf()`, and `printf()` functions shall return the number of bytes transmitted.

/ [CX] / Upon successful completion, the `asprintf()` function shall return the number of bytes written to the allocated string stored in the location referenced by `ptr`, excluding the terminating null byte.

Upon successful completion, the `sprintf()` function shall return the number of bytes written to `s`, excluding the terminating null byte.

Upon successful completion, the `snprintf()` function shall return the number of bytes that would be written to `s` had `n` been sufficiently large excluding the terminating null byte.

If an error was encountered, these functions shall return a negative value / [CX] / and set `errno` to indicate the error. For `asprintf()`, if memory allocation was not possible, or if some other error occurs, the function shall return a negative value, and the contents of the location referenced by `ptr` are undefined, but shall not refer to allocated memory.

If the value of `n` is zero on a call to `snprintf()`, nothing shall be written, the number of bytes that would have been written had `n` been sufficiently large excluding the terminating null shall be returned, and `s` may be a null pointer.

## ERRORS

For the conditions under which / [CX] / `dprintf()`, `fprintf()`, and `printf()` fail and may fail, refer to `fputc()` or `fputwc()`.

In addition, all forms of `fprintf()` shall fail if:

### [EILSEQ]

/ [CX] / A wide-character code that does not correspond to a valid character has been detected.

### [EOVERFLOW]

/ [CX] / The value to be returned is greater than `{INT_MAX}`.

/ [CX] / The `asprintf()` function shall fail if:

### [ENOMEM]

Insufficient storage space is available.

/ [CX] / The `dprintf()` function may fail if:

### [EBADF]

The `fildes` argument is not a valid file descriptor.

*/ [CX]* / The `dprintf()`, `fprintf()`, and `printf()` functions may fail if:

**[ENOMEM]**

*/ [CX]* / Insufficient storage space is available.

---

*The following sections are informative.*

## EXAMPLES

### Printing Language-Independent Date and Time

The following statement can be used to print date and time using a language-independent format:

```
printf(format, weekday, month, day, hour, min);
```

For American usage, format could be a pointer to the following string:

```
"%s, %s %d, %d:%.2d\n"
```

This example would produce the following message:

```
Sunday, July 3, 10:02
```

For German usage, format could be a pointer to the following string:

```
"%1$s, %3$d. %2$s, %4$d:%5$.2d\n"
```

This definition of format would produce the following message:

```
Sonntag, 3. Juli, 10:02
```

### Printing File Information

The following example prints information about the type, permissions, and number of links of a specific file in a directory.

The first two calls to `printf()` use data decoded from a previous `stat()` call. The user-defined `strperm()` function shall return a string similar to the one at the beginning of the output for the following command:

```
ls -l
```

The next call to `printf()` outputs the owner's name if it is found using `getpwuid()`; the `getpwuid()` function shall return a `passwd` structure from which the name of the user is extracted. If the user name is not found, the program instead prints out the numeric value of the user ID.

The next call prints out the group name if it is found using `getgrgid()`; `getgrgid()` is very similar to `getpwuid()` except that it shall return group information based on the group number. Once again, if the group is not found, the program prints the numeric value of the group for the entry.

The final call to `printf()` prints the size of the file.

```
#include <stdio.h>
#include <sys/types.h>
#include <pwd.h>
#include <grp.h>

char *strperm (mode_t);
...
struct stat statbuf;
struct passwd *pwd;
struct group *grp;
...
printf("%10.10s", strperm (statbuf.st_mode));
printf("%4d", statbuf.st_nlink);

if ((pwd = getpwuid(statbuf.st_uid)) != NULL)
    printf(" %-8.8s", pwd->pw_name);
else
    printf(" %-8ld", (long) statbuf.st_uid);

if ((grp = getgrgid(statbuf.st_gid)) != NULL)
    printf(" %-8.8s", grp->gr_name);
else
    printf(" %-8ld", (long) statbuf.st_gid);

printf("%9jd", (intmax_t) statbuf.st_size);
...
```

## Printing a Localized Date String

The following example gets a localized date string. The `nl_langinfo()` function shall return the localized date string, which specifies the order and layout of the date. The `strftime()` function takes this information and, using the `tm` structure for values, places the date and time information into `datestring`. The `printf()` function then outputs `datestring` and the name of the entry.

```
#include <stdio.h>
#include <time.h>
#include <langinfo.h>
...
struct dirent *dp;
struct tm *tm;
char datestring[256];
...
strftime(datestring, sizeof(datestring), nl_langinfo (D_T_FMT), tm
printf(" %s %s\n", datestring, dp->d_name);
...
```

## Printing Error Information

The following example uses `fprintf()` to write error information to standard error.

In the first group of calls, the program tries to open the password lock file named `LOCKFILE`. If the file already exists, this is an error, as indicated by the `O_EXCL` flag on the `open()` function. If the call fails, the program assumes that someone else is updating the password file, and the program exits.

The next group of calls saves a new password file as the current password file by creating a link between `LOCKFILE` and the new password file `PASSWDFILE`.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

#define LOCKFILE "/etc/ptmp"
#define PASSWDFILE "/etc/passwd"

...
int pfd;
...
if ((pfd = open(LOCKFILE, O_WRONLY | O_CREAT | O_EXCL,
    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
{
    fprintf(stderr, "Cannot open /etc/ptmp. Try again later.\n");
    exit(1);
}
...
if (link(LOCKFILE,PASSWDFILE) == -1) {
    fprintf(stderr, "Link error: %s\n", strerror(errno));
```

```
    exit(1);
}
...
```

## Printing Usage Information

The following example checks to make sure the program has the necessary arguments, and uses `fprintf()` to print usage information if the expected number of arguments is not present.

```
#include <stdio.h>
#include <stdlib.h>
...
char *Options = "hdbtl";
...
if (argc < 2) {
    fprintf(stderr, "Usage: %s -%s <file>\n", argv[0], Options); exit(1);
}
...
```

## Formatting a Decimal String

The following example prints a key and data pair on `stdout`. Note use of the ('\*') in the format string; this ensures the correct number of decimal places for the element based on the number of elements requested.

```
#include <stdio.h>
...
long i;
char *keystr;
int elementlen, len;
...
while (len < elementlen) {
    ...
    printf("%s Element%0*ld\n", keystr, elementlen, i);
    ...
}
```

## Creating a Pathname

The following example creates a pathname using information from a previous `getpwnam()` function that returned the password database entry of the user.

```

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
...
char *pathname;
struct passwd *pw;
size_t len;
...
// digits required for pid_t is number of bits times
// log2(10) = approx 10/33
len = strlen(pw->pw_dir) + 1 + 1+(sizeof(pid_t)*80+32)/33 +
      sizeof ".out";
pathname = malloc(len);
if (pathname != NULL)
{
    snprintf(pathname, len, "%s/%jd.out", pw->pw_dir,
             (intmax_t)getpid());
    ...
}

```

## Reporting an Event

The following example loops until an event has timed out. The pause() function waits forever unless it receives a signal. The fprintf() statement should never occur due to the possible return values of pause().

```

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
...
while (!event_complete) {
    ...
    if (pause() != -1 || errno != EINTR)
        fprintf(stderr, "pause: unknown error: %s\n", strerror(errno));
    ...
}

```

## Printing Monetary Information

The following example uses strfmon() to convert a number and store it as a formatted monetary string named convbuf. If the first number is printed, the program prints the format and the description; otherwise, it just prints the number.

```

#include <monetary.h>
#include <stdio.h>
...
struct tblfmt {
    char *format;
    char *description;
};

struct tblfmt table[] = {
    { "%n", "default formatting" },
    { "%11n", "right align within an 11 character field" },
    { "%#5n", "aligned columns for values up to 99999" },
    { "%=*#5n", "specify a fill character" },
    { "%=0#5n", "fill characters do not use grouping" },
    { "%^#5n", "disable the grouping separator" },
    { "%^#5.0n", "round off to whole units" },
    { "%^#5.4n", "increase the precision" },
    { "%(#5n", "use an alternative pos/neg style" },
    { "%!(#5n", "disable the currency symbol" },
};

...
float input[3];
int i, j;
char convbuf[100];
...
strfmmon(convbuf, sizeof(convbuf), table[i].format, input[j]);

if (j == 0) {
    printf("%s %s      %s\n", table[i].format,
           convbuf, table[i].description);
}
else {
    printf("      %s\n", convbuf);
}
...

```

## Printing Wide Characters

The following example prints a series of wide characters. Suppose that "L @ @" expands to three bytes:

```

wchar_t wz [3] = L"@@";           // Zero-terminated
wchar_t wn [3] = L"@@";           // Untermminated

fprintf (stdout,"%ls", wz);      // Outputs 6 bytes
fprintf (stdout,"%ls", wn);      // Undefined because wn has no terminator
fprintf (stdout,"%4ls", wz);     // Outputs 3 bytes
fprintf (stdout,"%4ls", wn);     // Outputs 3 bytes; no terminator needed
fprintf (stdout,"%9ls", wz);     // Outputs 6 bytes

```

```
fprintf (stdout,"%9ls", wn); // Outputs 9 bytes; no terminator needed
fprintf (stdout,"%10ls", wz); // Outputs 6 bytes
fprintf (stdout,"%10ls", wn); // Undefined because wn has no terminator
```

In the last line of the example, after processing three characters, nine bytes have been output. The fourth character must then be examined to determine whether it converts to one byte or more. If it converts to more than one byte, the output is only nine bytes. Since there is no fourth character in the array, the behavior is undefined.

## APPLICATION USAGE

If the application calling `fprintf()` has any objects of type `wint_t` or `wchar_t`, it must also include the header to have these objects defined.

The space allocated by a successful call to `asprintf()` should be subsequently freed by a call to `free()`.

## RATIONALE

If an implementation detects that there are insufficient arguments for the format, it is recommended that the function should fail and report an `[EINVAL]` error.

The behavior specified for the `%lc` conversion differs slightly from the specification in the ISO C standard, in that printing the null wide character produces a null byte instead of 0 bytes of output as would be required by a strict reading of the ISO C standard's direction to behave as if applying the `%ls` specifier to a `wchar_t` array whose first element is the null wide character. Requiring a multi-byte output for every possible wide character, including the null character, matches historical practice, and provides consistency with `%c` in `fprintf()` and with both `%c` and `%lc` in `fwprintf()`. It is anticipated that a future edition of the ISO C standard will change to match the behavior specified here.

## FUTURE DIRECTIONS

None.

## SEE ALSO

2.5 Standard I/O Streams, `fputc()`, `fscanf()`, `setlocale()`, `strfmon()`, `strlcat()`, `wcrtomb()`, `wcslcat()`

## CHANGE HISTORY

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

### Issue 5

Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the l (ell) qualifier can now be used with c and s conversion specifiers.

The snprintf() function is new in Issue 5.

### Issue 6

Extensions beyond the ISO C standard are marked.

The normative text is updated to avoid use of the term "must" for application requirements.

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

- The prototypes for fprintf(), printf(), snprintf(), and sprintf() are updated, and the XSI shading is removed from snprintf().
- The description of snprintf() is aligned with the ISO C standard. Note that this supersedes the snprintf() description in The Open Group Base Resolution bwg98-006, which changed the behavior from Issue 5.
- The DESCRIPTION is updated.

The DESCRIPTION is updated to use the terms "conversion specifier" and "conversion specification" consistently.

ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

An example of printing wide characters is added.

### Issue 7

Austin Group Interpretation 1003.1-2001 #161 is applied, updating the DESCRIPTION of the 0 flag.

Austin Group Interpretation 1003.1-2001 #170 is applied.

ISO/IEC 9899:1999 standard, Technical Corrigendum 2 #68 (SD5-XSH-ERN-70) is applied, revising the description of g and G.

SD5-XSH-ERN-174 is applied.

The `dprintf()` function is added from The Open Group Technical Standard, 2006, Extended API Set Part 1.

Functionality relating to the `%n$` form of conversion specification and the flag is moved from the XSI option to the Base.

Changes are made related to support for finegrained timestamps.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0163 [302], XSH/TC1-2008/0164 [316], XSH/TC1-2008/0165 [316], XSH/TC1-2008/0166 [451,291], and XSH/TC1-2008/0167 [14] are applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0126 [894], XSH/TC2-2008/0127 [557], and XSH/TC2-2008/0128 [936] are applied.

## Issue 8

Austin Group Defect 986 is applied, adding `strlcat()` and `wcslcat()` to the SEE ALSO section.

Austin Group Defect 1020 is applied, clarifying that the `snprintf()` argument `n` limits the number of bytes written to `s`; it is not necessarily the same as the size of `s`.

Austin Group Defect 1021 is applied, changing "output error" to "error" in the RETURN VALUE section.

Austin Group Defect 1137 is applied, clarifying the use of `"%n$"` and `"*m$"` in conversion specifications.

Austin Group Defect 1205 is applied, changing the description of the `%` conversion specifier.

Austin Group Defect 1219 is applied, removing the `snprintf()`-specific [EOVERFLOW] error.

Austin Group Defect 1496 is applied, adding the `asprintf()` function.

Austin Group Defect 1562 is applied, clarifying that it is the application's responsibility to ensure that the format is a character string, beginning and ending in its initial shift state, if any.

Austin Group Defect 1647 is applied, changing the description of the `c` conversion specifier and updating the statement that this volume of POSIX.1-2024 defers to the ISO C standard so that it excludes the `%lc` conversion when passed a null wide character.

---

## 1.47. fputc — put a byte on a stream

---

### SYNOPSIS

```
#include <stdio.h>

int fputc(int c, FILE *stream);
```

### DESCRIPTION

The `fputc()` function shall write the byte specified by `c` (converted to an `unsigned char`) to the output stream pointed to by `stream`, at the position indicated by the associated file-position indicator for the stream (if defined), and shall advance the indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the byte shall be appended to the output stream.

The last data modification and last file status change timestamps of the file shall be marked for update between the successful execution of `fputc()` and the next successful completion of a call to `fflush()` or `fclose()` on the same stream or a call to `exit()` or `abort()`.

### RETURN VALUE

Upon successful completion, `fputc()` shall return the value it has written. Otherwise, it shall return EOF, the error indicator for the stream shall be set, and `errno` shall be set to indicate the error.

### ERRORS

The `fputc()` function shall fail if either the `stream` is unbuffered or the `stream`'s buffer needs to be flushed, and:

- **[EAGAIN]**

The `O_NONBLOCK` flag is set for the file descriptor underlying `stream` and the thread would be delayed in the write operation.

- **[EBADF]**

The file descriptor underlying `stream` is not a valid file descriptor open for writing.

- **[EFBIG]**

An attempt was made to write to a file that exceeds the maximum file size.

- **[EFBIG]**

An attempt was made to write to a file that exceeds the file size limit of the process. A SIGXFSZ signal shall also be generated for the thread.

- **[EFBIG]**

The file is a regular file and an attempt was made to write at or beyond the offset maximum.

- **[EINTR]**

The write operation was terminated due to the receipt of a signal, and no data was transferred.

- **[EIO]**

A physical I/O error has occurred, or the process is a member of a background process group attempting to write to its controlling terminal, TOSTOP is set, the calling thread is not blocking SIGTTOU, the process is not ignoring SIGTTOU, and the process group of the process is orphaned. This error may also be returned under implementation-defined conditions.

- **[ENOSPC]**

There was no free space remaining on the device containing the file.

- **[EPIPE]**

An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A SIGPIPE signal shall also be sent to the thread.

The `fputc()` function may fail if:

- **[ENOMEM]**

Insufficient storage space is available.

- **[ENXIO]**

A request was made of a nonexistent device, or the request was outside the capabilities of the device.

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

2.5 Standard I/O Streams, `ferror()`, `fopen()`, `getrlimit()`, `putc()`,  
`puts()`, `setbuf()`  
XBD `<stdio.h>`

## CHANGE HISTORY

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

### Issue 5

Large File Summit extensions are added.

### Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EIO] and [EFBIG] mandatory error conditions are added.
- The [ENOMEM] and [ENXIO] optional error conditions are added.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/37 is applied, updating the [EAGAIN] error in the ERRORS section from "the process would be delayed" to "the thread would be delayed".

### Issue 7

Changes are made related to support for finegrained timestamps.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0168 [79] and XSH/TC1-2008/0169 [14] are applied.

## Issue 8

Austin Group Defect 308 is applied, clarifying the handling of [EFBIG] errors.

Austin Group Defect 1669 is applied, removing XSI shading from part of the [EFBIG] error relating to the file size limit for the process.

## 1.48. fputs

---

### SYNOPSIS

```
#include <stdio.h>

int fputs(const char *restrict s, FILE *restrict stream);
```

### DESCRIPTION

The `fputs()` function shall write the null-terminated string pointed to by `s` to the stream pointed to by `stream`. The terminating null byte shall not be written.

The last data modification and last file status change timestamps of the file shall be marked for update between the successful execution of `fputs()` and the next successful completion of a call to `fflush()` or `fclose()` on the same stream or a call to `exit()` or `abort()`.

### RETURN VALUE

Upon successful completion, `fputs()` shall return a non-negative number. Otherwise, it shall return EOF, set an error indicator for the stream, and set `errno` to indicate the error.

### ERRORS

Refer to `fputc()`.

### EXAMPLES

#### Printing to Standard Output

The following example gets the current time, converts it to a string using `localtime()` and `asctime()`, and prints it to standard output using `fputs()`. It then prints the number of minutes to an event for which it is waiting.

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

```
...
time_t now;
int minutes_to_event;
...
time(&now);
printf("The time is ");
fputs(asctime(localtime(&now)), stdout);
printf("There are still %d minutes to the event.\n",
       minutes_to_event);
...
```

## APPLICATION USAGE

The `puts()` function appends a `<newline>` while `fputs()` does not.

This volume of POSIX.1-2024 requires that successful completion simply return a non-negative integer. There are at least three known different implementation conventions for this requirement:

- Return a constant value.
- Return the last character written.
- Return the number of bytes written. Note that this implementation convention cannot be adhered to for strings longer than `{INT_MAX}` bytes as the value would not be representable in the return type of the function. For backwards-compatibility, implementations can return the number of bytes for strings of up to `{INT_MAX}` bytes, and return `{INT_MAX}` for all longer strings.

## RATIONALE

The `fputs()` function is one whose source code was specified in the referenced *The C Programming Language*. In the original edition, the function had no defined return value, yet many practical implementations would, as a side-effect, return the value of the last character written as that was the value remaining in the accumulator used as a return value. In the second edition of the book, either the fixed value 0 or EOF would be returned depending upon the return value of `ferror()`; however, for compatibility with extant implementations, several implementations would, upon success, return a positive value representing the last byte written.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- 2.5 Standard I/O Streams
- `fopen()`
- `putc()`
- `puts()`
- XBD `<stdio.h>`

## CHANGE HISTORY

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

### Issue 6

Extensions beyond the ISO C standard are marked.

The `fputs()` prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

### Issue 7

Changes are made related to support for finegrained timestamps.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0170 [174,412], XSH/TC1-2008/0171 [412], and XSH/TC1-2008/0172 [14] are applied.

---

## 1.49. fread

---

### SYNOPSIS

```
#include <stdio.h>

size_t fread(void *restrict ptr, size_t size, size_t nitems,
             FILE *restrict stream);
```

### DESCRIPTION

The `fread()` function shall read into the array pointed to by `ptr` up to `nitems` elements whose size is specified by `size` in bytes, from the stream pointed to by `stream`. For each object, `size` calls shall be made to the `fgetc()` function and the results stored, in the order read, in an array of `unsigned char` exactly overlaying the object. The file position indicator for the stream (if defined) shall be advanced by the number of bytes successfully read. If an error occurs, the resulting value of the file position indicator for the stream is unspecified. If a partial element is read, its value is unspecified.

The `fread()` function may mark the last data access timestamp of the file associated with `stream` for update. The last data access timestamp shall be marked for update by the first successful execution of `fgetc()`, `fgets()`, `fread()`, `fscanf()`, `getc()`, `getchar()`, `getdelim()`, `getline()`, or `scanf()` using `stream` that returns data not supplied by a prior call to `ungetc()`.

### RETURN VALUE

The `fread()` function shall return the number of elements successfully read, which shall be less than `nitems` only if an error or end-of-file is encountered, or `size` is zero. If `size` or `nitems` is 0, `fread()` shall return 0 and the contents of the array and the state of the stream shall remain unchanged. Otherwise, if an error occurs, the error indicator for the stream shall be set, and `errno` shall be set to indicate the error.

### ERRORS

Refer to `fgetc()`.

*The following sections are informative.*

## EXAMPLES

### Reading from a Stream

The following example transfers a single 100-byte fixed length record from the `fp` stream into the array pointed to by `buf`.

```
#include <stdio.h>
...
size_t elements_read;
char buf[100];
FILE *fp;
...
elements_read = fread(buf, sizeof(buf), 1, fp);
...
```

If a read error occurs, `elements_read` will be zero but the number of bytes read from the stream could be anything from zero to `sizeof(buf) -1`.

The following example reads multiple single-byte elements from the `fp` stream into the array pointed to by `buf`.

```
#include <stdio.h>
...
size_t bytes_read;
char buf[100];
FILE *fp;
...
bytes_read = fread(buf, 1, sizeof(buf), fp);
...
```

If a read error occurs, `bytes_read` will contain the number of bytes read from the stream.

## APPLICATION USAGE

The `ferror()` or `feof()` functions must be used to distinguish between an error condition and an end-of-file condition.

Because of possible differences in element length and byte ordering, files written using `fwrite()` are application-dependent, and possibly cannot be read using `fread()` by a different application or by the same application on a different processor.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [2.5 Standard I/O Streams](#)
- `feof()`
- `ferror()`
- `fgetc()`
- `fopen()`
- `fscanf()`
- `getc()`
- 

## CHANGE HISTORY

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

### Issue 6

Extensions beyond the ISO C standard are marked.

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

- The `fread()` prototype is updated.
- The DESCRIPTION is updated to describe how the bytes from a call to `fgetc()` are stored.

### Issue 7

Changes are made related to support for finegrained timestamps.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0178 [232] and XSH/TC1-2008/0179 [14] are applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0129 [926] is applied.

## Issue 8

Austin Group Defect 1196 is applied, clarifying the RETURN VALUE section.

Austin Group Defect 1330 is applied, removing obsolescent interfaces.

Austin Group Defect 1624 is applied, changing the RETURN VALUE section.

---

## 1.50. free — free allocated memory

---

### SYNOPSIS

```
#include <stdlib.h>

void free(void *ptr);
```

### DESCRIPTION

The `free()` function shall cause the space pointed to by `ptr` to be deallocated; that is, made available for further allocation. If `ptr` is a null pointer, no action shall occur. Otherwise, if the argument does not match a pointer earlier returned by `aligned_alloc()`, `calloc()`, `malloc()`, `posix_memalign()`, `realloc()`, `reallocarray()`, or a function in POSIX.1-2024 that allocates memory as if by `malloc()`, or if the space has been deallocated by a call to `free()`, `reallocarray()`, or `realloc()`, the behavior is undefined.

Any use of a pointer that refers to freed space results in undefined behavior.

The `free()` function shall not modify `errno` if `ptr` is a null pointer or a pointer previously returned as if by `malloc()` and not yet deallocated.

For purposes of determining the existence of a data race, `free()` shall behave as though it accessed only memory locations accessible through its argument and not other static duration storage. The function may, however, visibly modify the storage that it deallocates. Calls to `aligned_alloc()`, `calloc()`, `free()`, `malloc()`, `posix_memalign()`, `reallocarray()` and `realloc()` that allocate or deallocate a particular region of memory shall occur in a single total order, and each such deallocation call shall synchronize with the next allocation (if any) in this order.

### RETURN VALUE

The `free()` function shall not return a value.

### ERRORS

No errors are defined.

## EXAMPLES

None.

## APPLICATION USAGE

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

Because the `free()` function does not modify `errno` for valid pointers, it is safe to use it in cleanup code without corrupting earlier errors, such as in this example code:

```
// buf was obtained by malloc(buflen)
ret = write(fd, buf, buflen);
if (ret < 0) {
    free(buf);
    return ret;
}
```

However, earlier versions of this standard did not require this, and the same example had to be written as:

```
// buf was obtained by malloc(buflen)
ret = write(fd, buf, buflen);
if (ret < 0) {
    int save = errno;
    free(buf);
    errno = save;
    return ret;
}
```

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `aligned_alloc()`
- `calloc()`
- `malloc()`
- `posix_memalign()`
- `realloc()`
- `<stdlib.h>`

## CHANGE HISTORY

### First released in Issue 1

Derived from Issue 1 of the SVID.

### Issue 6

Reference to the `valloc()` function is removed.

### Issue 7

The DESCRIPTION is updated to clarify that if the pointer returned is not by a function that allocates memory as if by `malloc()`, then the behavior is undefined.

### Issue 8

Austin Group Defect 385 is applied, adding a requirement that `free()` does not modify `errno` when passed a pointer to an object than can be freed.

Austin Group Defect 1218 is applied, adding `reallocarray()`.

Austin Group Defect 1302 is applied, aligning this function with the ISO/IEC 9899:2018 standard.

---

## 1.51. freopen

---

### SYNOPSIS

```
#include <stdio.h>

FILE *freopen(const char *restrict pathname,
              const char *restrict mode,
              FILE *restrict stream);
```

### DESCRIPTION

[Option Start] Except for the "exclusive access" requirement (see `fopen()`), the functionality described on this reference page is aligned with the ISO C standard. Any other conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard for all `freopen()` functionality except in relation to "exclusive access".

[Option End]

The `freopen()` function shall first attempt to flush the stream associated with `stream` as if by a call to `fflush(stream)`. Failure to flush the stream successfully shall be ignored. If `pathname` is not a null pointer, `freopen()` shall close any file descriptor associated with `stream`. Failure to close the file descriptor successfully shall be ignored. The error and end-of-file indicators for the stream shall be cleared.

The `freopen()` function shall open the file whose `pathname` is the string pointed to by `pathname` and associate the stream pointed to by `stream` with it. The `mode` argument shall be used just as in `fopen()`.

The original stream shall be closed regardless of whether the subsequent open succeeds.

If `pathname` is a null pointer, the `freopen()` function shall attempt to change the mode of the stream to that specified by `mode`, as if the name of the file currently associated with the stream had been used. In this case, the file descriptor associated with the stream need not be closed if the call to `freopen()` succeeds. It is implementation-defined which changes of mode are permitted (if any), and under what circumstances.

After a successful call to the `freopen()` function, the orientation of the stream shall be cleared, [Option Start] the encoding rule shall be cleared, [Option End] and the associated `mbstate_t` object shall be set to describe an initial conversion state.

[Option Start] If pathname is not a null pointer, or if pathname is a null pointer and the specified mode change necessitates the file descriptor associated with the stream to be closed and reopened, the file descriptor associated with the reopened stream shall be allocated and opened as if by a call to open() with the flags specified for fopen() with the same mode argument. [Option End]

## RETURN VALUE

Upon successful completion, freopen() shall return the value of stream. Otherwise, a null pointer shall be returned, [Option Start] and errno shall be set to indicate the error. [Option End]

## ERRORS

The freopen() function shall fail if:

- **[EACCES]**

[Option Start] Search permission is denied on a component of the path prefix, or the file exists and the permissions specified by mode are denied, or the file does not exist and write permission is denied for the parent directory of the file to be created. [Option End]

- **[EBADF]**

[Option Start] The file descriptor underlying the stream is not a valid file descriptor when pathname is a null pointer. [Option End]

- **[EILSEQ]**

[Option Start] The mode argument begins with w or a, the file did not previously exist, and the last pathname component is not a portable filename and cannot be created in the target directory. [Option End]

- **[EEXIST]**

[Option Start] The mode argument begins with w or a and includes x, but the file already exists. [Option End]

- **[EINTR]**

[Option Start] A signal was caught during freopen(). [Option End]

- **[EISDIR]**

[Option Start] The named file is a directory and mode requires write access. [Option End]

- **[ELOOP]**

[Option Start] A loop exists in symbolic links encountered during resolution of

the pathname argument. [Option End]

- **[EMFILE]**

[Option Start] All file descriptors available to the process are currently open.  
[Option End]

- **[ENAMETOOLONG]**

[Option Start] The length of a component of a pathname is longer than {NAME\_MAX}. [Option End]

- **[ENFILE]**

[Option Start] The maximum allowable number of files is currently open in the system. [Option End]

- **[ENOENT]**

[Option Start] The mode string begins with 'r' and a component of pathname does not name an existing file, or mode begins with 'w' or 'a' and a component of the path prefix of pathname does not name an existing file, or pathname is an empty string. [Option End]

- **[ENOENT] or [ENOTDIR]**

[Option Start] The pathname argument contains at least one non- character and ends with one or more trailing characters. If pathname without the trailing characters would name an existing file, an [ENOENT] error shall not occur.  
[Option End]

- **[ENOSPC]**

[Option Start] The directory or file system that would contain the new file cannot be expanded, the file does not exist, and it was to be created. [Option End]

- **[ENOTDIR]**

[Option Start] A component of the path prefix names an existing file that is neither a directory nor a symbolic link to a directory, or the pathname argument contains at least one non- character and ends with one or more trailing characters and the last pathname component names an existing file that is neither a directory nor a symbolic link to a directory. [Option End]

- **[ENXIO]**

[Option Start] The named file is a character special or block special file, and the device associated with this special file does not exist. [Option End]

- **[EOVERFLOW]**

[Option Start] The named file is a regular file and the size of the file cannot be represented correctly in an object of type off\_t. [Option End]

- **[EROFS]**

[Option Start] The named file resides on a read-only file system and mode requires write access. [Option End]

The `fopen()` function may fail if:

- **[EBADF]**

[Option Start] The mode with which the file descriptor underlying the stream was opened does not support the requested mode when pathname is a null pointer. [Option End]

- **[EINVAL]**

[Option Start] The value of the mode argument is not valid. [Option End]

- **[ELOOP]**

[Option Start] More than `{SYMLOOP_MAX}` symbolic links were encountered during resolution of the pathname argument. [Option End]

- **[ENAMETOOLONG]**

[Option Start] The length of a pathname exceeds `{PATH_MAX}`, or pathname resolution of a symbolic link produced an intermediate result with a length that exceeds `{PATH_MAX}`. [Option End]

- **[ENOMEM]**

[Option Start] Insufficient storage space is available. [Option End]

- **[ENXIO]**

[Option Start] A request was made of a nonexistent device, or the request was outside the capabilities of the device. [Option End]

- **[ETXTBSY]**

[Option Start] The file is a pure procedure (shared text) file that is being executed and mode requires write access. [Option End]

---

*The following sections are informative.*

## EXAMPLES

### Directing Standard Output to a File

The following example logs all standard output to the `/tmp/logfile` file.

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

```

```
FILE *fp;  
...  
fp = freopen ("/tmp/logfile", "a+", stdout);  
...
```

## APPLICATION USAGE

The `freopen()` function is typically used to attach the pre-opened streams associated with `stdin`, `stdout`, and `stderr` to other files.

Since implementations are not required to support any stream mode changes when the pathname argument is `NULL`, portable applications cannot rely on the use of `freopen()` to change the stream mode, and use of this feature is discouraged. The feature was originally added to the ISO C standard in order to facilitate changing `stdin` and `stdout` to binary mode. Since a '`b`' character in the mode has no effect on POSIX systems, this use of the feature is unnecessary in POSIX applications. However, even though the '`b`' is ignored, a successful call to `freopen(NULL, "wb", stdout)` does have an effect. In particular, for regular files it truncates the file and sets the file-position indicator for the stream to the start of the file. It is possible that these side-effects are an unintended consequence of the way the feature was specified in the ISO/IEC 9899:1999 standard (and still is in the current standard), but unless or until the ISO C standard is changed, applications which successfully call `freopen(NULL, "wb", stdout)` will behave in unexpected ways on conforming systems in situations such as:

```
{ appl file1; appl file2; } > file3
```

which will result in `file3` containing only the output from the second invocation of `appl`.

See also the APPLICATION USAGE for `fopen()`.

## RATIONALE

See the RATIONALE for `fopen()`.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- 2.5 Standard I/O Streams
- fclose()
- fdopen()
- fflush()
- fmemopen()
- fopen()
- mbsinit()
- open()
- open\_memstream()
- 

## CHANGE HISTORY

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

### Issue 5

The DESCRIPTION is updated to indicate that the orientation of the stream is cleared and the conversion state of the stream is set to an initial conversion state by a successful call to the freopen() function.

Large File Summit extensions are added.

### Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, text is added to indicate setting of the offset maximum in the open file description. This change is to support large files.
- In the ERRORS section, the [EOVERFLOW] condition is added. This change is to support large files.
- The [ELOOP] mandatory error condition is added.
- A second [ENAMETOOLONG] is added as an optional error condition.

- The [EINVAL], [ENOMEM], [ENXIO], and [ETXTBSY] optional error conditions are added.

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

- The `freopen()` prototype is updated.
- The DESCRIPTION is updated.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

The DESCRIPTION is updated regarding failure to close, changing the "file" to "file descriptor".

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/40 is applied, adding the following sentence to the DESCRIPTION: "In this case, the file descriptor associated with the stream need not be closed if the call to `freopen()` succeeds.".

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/41 is applied, adding a mandatory [EBADF] error, and an optional [EBADF] error to the ERRORS section.

## Issue 7

Austin Group Interpretation 1003.1-2001 #043 is applied, clarifying that the `freopen()` function allocates a file descriptor as per `open()`.

Austin Group Interpretation 1003.1-2001 #143 is applied.

Austin Group Interpretation 1003.1-2001 #159 is applied, clarifying requirements for the flags set on the open file description.

SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

SD5-XSH-ERN-150 and SD5-XSH-ERN-219 are applied.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0181 [291,433], XSH/TC1-2008/0182 [146,433], XSH/TC1-2008/0183 [324], and XSH/TC1-2008/0184 [14] are applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0134 [822] is applied.

## Issue 8

Austin Group Defect 293 is applied, adding the [EILSEQ] error.

Austin Group Defect 411 is applied, adding the e and x mode string characters.

Austin Group Defect 1200 is applied, correcting the argument name in the [ELOOP] errors.

Austin Group Defect 1302 is applied, aligning this function with the ISO/IEC 9899:2018 standard.

## 1.52. fscanf

### Synopsis

```
#include <stdio.h>

int fscanf(FILE *restrict stream, const char *restrict format, ...
int scanf(const char *restrict format, ...);
int sscanf(const char *restrict s, const char *restrict format, ...
```

### Description

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `fscanf()` function shall read from the named input stream. The `scanf()` function shall read from the standard input stream `stdin`. The `sscanf()` function shall read from the string `s`. Each function reads bytes, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string format described below, and a set of pointer arguments indicating where the converted input should be stored. The result is undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments shall be evaluated but otherwise ignored.

[CX] Conversions can be applied to the *n*-th argument after the format in the argument list, rather than to the next unused argument. In this case, the conversion specifier character `%` (see below) is replaced by the sequence `"%n$"`, where *n* is a decimal integer in the range `[1,{NL_ARGMAX}]`. This feature provides for the definition of format strings that select arguments in an order appropriate to specific languages. In format strings containing the `"%n$"` form of conversion specifications, it is unspecified whether numbered arguments in the argument list can be referenced from the format string more than once.

The format can contain either form of a conversion specification—that is, `%` or `"%n$"`—but the two forms cannot be mixed within a single format string. The only exception to this is that `%%` or `%*` can be mixed with the `"%n$"` form. 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 pointers.

The `fscanf()` function in all its forms shall allow detection of a language-dependent radix character in the input string. The radix character is defined in the current locale (category LC\_NUMERIC). In the POSIX locale, or in a locale where the radix character is not defined, the radix character shall default to a period ('.').

## Format String

The application shall ensure that the format is a character string, beginning and ending in its initial shift state, if any, composed of zero or more directives. Each directive is composed of one of the following: one or more white-space bytes; an ordinary character (neither '%' nor a white-space byte); or a conversion specification. Each conversion specification is introduced by the character '%' [CX] or the character sequence "%n\$", after which the following appear in sequence:

- An optional assignment-suppressing character '\*'.
- An optional non-zero decimal integer that specifies the maximum field width.
- [CX] An optional assignment-allocation character 'm'.
- An optional length modifier that specifies the size of the receiving object.
- A conversion specifier character that specifies the type of conversion to be applied.

The `fscanf()` functions shall execute each directive of the format in turn. When all directives have been executed, or if a directive fails (as detailed below), the function shall return. Failures are described as input failures (due to the unavailability of input bytes) or matching failures (due to inappropriate input).

## Directive Execution

A directive composed of one or more white-space bytes shall be executed by reading input up to the first non-white-space byte, which shall remain unread, or until no more bytes can be read. The directive shall never fail.

A directive that is an ordinary character shall be executed as follows: the next byte shall be read from the input and compared with the byte that comprises the directive; if the comparison shows that they are not equivalent, the directive shall fail, and the differing and subsequent bytes shall remain unread. Similarly, if end-of-file, an encoding error, or a read error prevents a character from being read, the directive shall fail.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each conversion character. A conversion specification shall be executed in the following steps:

1. Input white-space bytes shall be skipped, unless the conversion specification includes a [, c, C, or n conversion specifier.
2. An item shall be read from the input, unless the conversion specification includes an n conversion specifier. An input item shall be 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 specifier) which is an initial subsequence of a matching sequence. The first byte, if any, after the input item shall remain unread. If the length of the input item is 0, the execution of the conversion specification shall fail; 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.
3. Except in the case of a % conversion specifier, the input item (or, in the case of a %n conversion specification, the count of input bytes) shall be 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 shall be 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 %, [CX] or in the n-th argument if introduced by the character sequence "%n\$". If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

[CX] The c, s, and [ conversion specifiers shall accept an optional assignment-allocation character 'm', which shall cause a memory buffer to be allocated to hold the conversion results. If the conversion specifier is s or [, the allocated buffer shall include space for a terminating null character (or wide character). In such a case, the argument corresponding to the conversion specifier should be a reference to a pointer variable that will receive a pointer to the allocated buffer. The system shall allocate a buffer as if `malloc()` had been called. The application shall be responsible for freeing the memory after usage. If there is insufficient memory to allocate a buffer, the function shall set `errno` to [ENOMEM] and a conversion error shall result. If the function returns EOF, any memory successfully allocated for parameters using assignment-allocation character 'm' by this call shall be freed before the function returns.

## Length Modifiers

The length modifiers and their meanings are:

### **hh**

Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an

argument with type pointer to `signed char` or `unsigned char`.

## **h**

Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to `short` or `unsigned short`.

## **I (ell)**

Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to `long` or `unsigned long`; that a following a, A, e, E, f, F, g, or G conversion specifier applies to an argument with type pointer to `double`; or that a following c, s, or [ conversion specifier applies to an argument with type pointer to `wchar_t`. [CX] If the 'm' assignment-allocation character is specified, the conversion applies to an argument with the type pointer to a pointer to `wchar_t`.

## **II (ell-ell)**

Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to `long long` or `unsigned long long`.

## **j**

Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to `intmax_t` or `uintmax_t`.

## **z**

Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to `size_t` or the corresponding signed integer type.

## **t**

Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to `ptrdiff_t` or the corresponding `unsigned` type.

## **L**

Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to an argument with type pointer to `long double`.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

# **Conversion Specifiers**

The following conversion specifiers are valid:

## **d**

Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of `strtol()` with the value 10 for the base argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to `int`.

i

Matches an optionally signed integer, whose format is the same as expected for the subject sequence of `strtol()` with 0 for the base argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to `int`.

o

Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of `strtoul()` with the value 8 for the base argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to `unsigned`.

u

Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of `strtoul()` with the value 10 for the base argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to `unsigned`.

x

Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of `strtoul()` with the value 16 for the base argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to `unsigned`.

a, e, f, g

Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected for the subject sequence of `strtod()`. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to `float`.

If the `fprintf()` family of functions generates character string representations for infinity and NaN (a symbolic entity encoded in floating-point format) to support IEEE Std 754-1985, the `fscanf()` family of functions shall recognize them as input.

s

Matches a sequence of bytes that are not white-space bytes. If the 'm' assignment-allocation character is not specified, the application shall ensure that the corresponding argument is a pointer to the initial byte of an array of `char`, `signed char`, or `unsigned char` large enough to accept the sequence and a terminating null character code, which shall be added automatically. [CX] Otherwise, the application shall ensure that the corresponding argument is a pointer to a pointer to a `char`.

If an l (ell) qualifier is present, the input is a sequence of characters that begins in the initial shift state. Each character shall be converted to a wide character as if by a call to the `mbrtowc()` function, with the conversion state described by an

`mbstate_t` object initialized to zero before the first character is converted. If the 'm' assignment-allocation character is not specified, the application shall ensure that the corresponding argument is a pointer to an array of `wchar_t` large enough to accept the sequence and the terminating null wide character, which shall be added automatically. [CX] Otherwise, the application shall ensure that the corresponding argument is a pointer to a pointer to a `wchar_t`.

[

Matches a non-empty sequence of bytes from a set of expected bytes (the scanset). The normal skip over white-space bytes shall be suppressed in this case. If the 'm' assignment-allocation character is not specified, the application shall ensure that the corresponding argument is a pointer to the initial byte of an array of `char`, `signed char`, or `unsigned char` large enough to accept the sequence and a terminating null byte, which shall be added automatically. [CX] Otherwise, the application shall ensure that the corresponding argument is a pointer to a pointer to a `char`.

If an l (ell) qualifier is present, the input is a sequence of characters that begins in the initial shift state. Each character in the sequence shall be converted to a wide character as if by a call to the `mbrtowc()` function, with the conversion state described by an `mbstate_t` object initialized to zero before the first character is converted. If the 'm' assignment-allocation character is not specified, the application shall ensure that the corresponding argument is a pointer to an array of `wchar_t` large enough to accept the sequence and the terminating null wide character, which shall be added automatically. [CX] Otherwise, the application shall ensure that the corresponding argument is a pointer to a pointer to a `wchar_t`.

The conversion specification includes all subsequent bytes in the format string up to and including the matching right-square-bracket (']'). The bytes between the square brackets (the scanlist) comprise the scanset, unless the byte after the left-square-bracket is a circumflex (^), in which case the scanset contains all bytes that do not appear in the scanlist between the circumflex and the right-square-bracket. If the conversion specification begins with "[" or "[^]", the right-square-bracket is included in the scanlist and the next right-square-bracket is the matching right-square-bracket that ends the conversion specification; otherwise, the first right-square-bracket is the one that ends the conversion specification. If a '-' is in the scanlist and is not the first character, nor the second where the first character is a '^', nor the last character, the behavior is implementation-defined.

c

Matches a sequence of bytes of the number specified by the field width (1 if no field width is present in the conversion specification). No null byte is added. The normal skip over white-space bytes shall be suppressed in this case. If the 'm' assignment-allocation character is not specified, the application shall ensure that the corresponding argument is a pointer to the initial byte of an array of `char`,

`signed char`, or `unsigned char` large enough to accept the sequence. [CX] Otherwise, the application shall ensure that the corresponding argument is a pointer to a pointer to a `char`.

If an `l` (ell) qualifier is present, the input shall be a sequence of characters that begins in the initial shift state. Each character in the sequence is converted to a wide character as if by a call to the `mbrtowc()` function, with the conversion state described by an `mbstate_t` object initialized to zero before the first character is converted. No null wide character is added. If the '`m`' assignment-allocation character is not specified, the application shall ensure that the corresponding argument is a pointer to an array of `wchar_t` large enough to accept the resulting sequence of wide characters. [CX] Otherwise, the application shall ensure that the corresponding argument is a pointer to a pointer to a `wchar_t`.

### **p**

Matches an implementation-defined set of sequences, which shall be the same as the set of sequences that is produced by the `%p` conversion specification of the corresponding `fprintf()` functions. The application shall ensure that the corresponding argument is a pointer to a pointer to `void`. The interpretation of the input item is implementation-defined. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise, the behavior of the `%p` conversion specification is undefined.

### **n**

No input is consumed. The application shall ensure that the corresponding argument is a pointer to the integer into which shall be written the number of bytes read from the input so far by this call to the `fscanf()` functions. Execution of a `%n` conversion specification shall not increment the assignment count returned at the completion of execution of the function. No argument shall be converted, but one shall be consumed.

### **%**

Matches a single '%' character; no assignment or conversion shall be done. The complete conversion specification shall be `%%`.

## **Return Value**

Upon successful completion, these functions shall return the number of successfully matched and assigned input items; this number can be zero in the event of an early matching failure. If the input ends before the first matching failure or conversion, `EOF` shall be returned. If a read error occurs, the error indicator for the stream shall be set, `EOF` shall be returned, and `errno` shall be set to indicate the error.

## Errors

For the conditions under which the functions shall fail and may fail, see the documentation for the underlying `fgetc()` or `fgetwc()` functions. In addition, the `fscanf()` functions may fail if:

- **[ENOMEM]** - Insufficient storage space is available to hold the converted value when using the 'm' assignment-allocation character.

## Examples

```
#include <stdio.h>

int main(void) {
    int i;
    float f;
    char s[100];

    /* Read integer, float, and string */
    if (fscanf(stdin, "%d %f %99s", &i, &f, s) == 3) {
        printf("Read: %d, %f, %s\n", i, f, s);
    }

    return 0;
}
```

## Application Usage

The `fscanf()` family of functions may be used in applications that need to read and parse formatted input from files, standard input, or strings. The format string provides flexible control over how input is interpreted and converted.

The assignment-allocation character 'm' is particularly useful when the input size is not known in advance, as it allows the implementation to allocate appropriate memory automatically.

## Rationale

The `fscanf()` functions provide a powerful mechanism for input parsing and conversion. The design allows for:

- Precise control over input parsing through format specifications

- Automatic memory allocation for variable-length input when using the 'm' modifier
- Support for both narrow and wide character input
- Numbered argument specifications for internationalized applications

The behavior is carefully specified to ensure consistent results across different implementations while maintaining compatibility with the ISO C standard.

## Future Directions

None.

## See Also

[fgetc\(\)](#) , [fgetwc\(\)](#) , [fprintf\(\)](#) , [fread\(\)](#) , [fseek\(\)](#) , [getchar\(\)](#) ,  
[scanf\(\)](#) , [setlocale\(\)](#) , [sscanf\(\)](#) , [strtod\(\)](#) , [strtol\(\)](#) , [strtoul\(\)](#) ,  
[ungetc\(\)](#)

## 1.53. `fsync`

---

### SYNOPSIS

```
#include <unistd.h>

int fsync(int fildes);
```

### DESCRIPTION

The `fsync()` function shall request that all data for the open file descriptor named by `fildes` is to be transferred to the storage device associated with the file described by `fildes`. The nature of the transfer is implementation-defined. The `fsync()` function shall not return until the system has completed that action or until an error is detected.

If `_POSIX_SYNCHRONIZED_IO` is defined, the `fsync()` function shall force all currently queued I/O operations associated with the file indicated by file descriptor `fildes` to the synchronized I/O completion state. All I/O operations shall be completed as defined for synchronized I/O file integrity completion.

### RETURN VALUE

Upon successful completion, `fsync()` shall return 0. Otherwise, -1 shall be returned and `errno` set to indicate the error. If the `fsync()` function fails, outstanding I/O operations are not guaranteed to have been completed.

### ERRORS

The `fsync()` function shall fail if:

- **[EBADF]**  
The `fildes` argument is not a valid descriptor.
- **[EINTR]**  
The `fsync()` function was interrupted by a signal.
- **[EINVAL]**  
The `fildes` argument does not refer to a file on which this operation is possible.

- **[EIO]**

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

In the event that any of the queued I/O operations fail, `fsync()` shall return the error conditions defined for `read()` and `write()`.

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

The `fsync()` function should be used by programs which require modifications to a file to be completed before continuing; for example, a program which contains a simple transaction facility might use it to ensure that all modifications to a file or files caused by a transaction are recorded.

An application that modifies a directory, for example, by creating a file in the directory, can invoke `fsync()` on the directory to ensure that the directory's entries and file attributes are synchronized. For most applications, synchronizing the directory's entries should not be necessary (see XBD 4.11 File System Cache).

## RATIONALE

The `fsync()` function is intended to force a physical write of data from the buffer cache, and to assure that after a system crash or other failure that all data up to the time of the `fsync()` call is recorded on the disk. Since the concepts of "buffer cache", "system crash", "physical write", and "non-volatile storage" are not defined here, the wording has to be more abstract.

If `_POSIX_SYNCHRONIZED_IO` is not defined, the wording relies heavily on the conformance document to tell the user what can be expected from the system. It is explicitly intended that a null implementation is permitted. This could be valid in the case where the system cannot assure non-volatile storage under any circumstances or when the system is highly fault-tolerant and the functionality is not required. In the middle ground between these extremes, `fsync()` might or might not actually cause data to be written where it is safe from a power failure. The conformance document should identify at least that one configuration exists (and how to obtain that configuration) where this can be assured for at least some files that the user can select to use for critical data. It is not intended that an exhaustive list is required, but rather sufficient information is provided so that if

critical data needs to be saved, the user can determine how the system is to be configured to allow the data to be written to non-volatile storage.

It is reasonable to assert that the key aspects of `fsync()` are unreasonable to test in a test suite. That does not make the function any less valuable, just more difficult to test. A formal conformance test should probably force a system crash (power shutdown) during the test for this condition, but it needs to be done in such a way that automated testing does not require this to be done except when a formal record of the results is being made. It would also not be unreasonable to omit testing for `fsync()`, allowing it to be treated as a quality-of-implementation issue.

## FUTURE DIRECTIONS

None.

## SEE ALSO

`sync()`

XBD `<unistd.h>`

## CHANGE HISTORY

First released in Issue 3.

### Issue 5

Aligned with `fsync()` in the POSIX Realtime Extension. Specifically, the DESCRIPTION and RETURN VALUE sections are much expanded, and the ERRORS section is updated to indicate that `fsync()` can return the error conditions defined for `read()` and `write()`.

### Issue 6

This function is marked as part of the File Synchronization option.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The `[EINVAL]` and `[EIO]` mandatory error conditions are added.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/44 is applied, applying an editorial rewording of the DESCRIPTION. No change in meaning is intended.

## Issue 8

Austin Group Defect 672 is applied, changing the APPLICATION USAGE section.

## 1.54. ftrylockfile

---

### SYNOPSIS

```
#include <stdio.h>

void flockfile(FILE *file);
int ftrylockfile(FILE *file);
void funlockfile(FILE *file);
```

### DESCRIPTION

These functions shall provide for explicit application-level locking of the locks associated with standard I/O streams (see [2.5 Standard I/O Streams](#)). These functions can be used by a thread to delineate a sequence of I/O statements that are executed as a unit.

The `flockfile()` function shall acquire for a thread ownership of a (`FILE *`) object.

The `ftrylockfile()` function shall acquire for a thread ownership of a (`FILE *`) object if the object is available; `ftrylockfile()` is a non-blocking version of `flockfile()`.

The `funlockfile()` function shall relinquish the ownership granted to the thread. The behavior is undefined if a thread other than the current owner calls the `funlockfile()` function.

The functions shall behave as if there is a lock count associated with each (`FILE *`) object. This count is implicitly initialized to zero when the (`FILE *`) object is created. The (`FILE *`) object is unlocked when the count is zero. When the count is positive, a single thread owns the (`FILE *`) object. When the `flockfile()` function is called, if the count is zero or if the count is positive and the caller owns the (`FILE *`) object, the count shall be incremented. Otherwise, the calling thread shall be suspended, waiting for the count to return to zero. Each call to `funlockfile()` shall decrement the count. This allows matching calls to `flockfile()` (or successful calls to `ftrylockfile()`) and `funlockfile()` to be nested.

### RETURN VALUE

None for `flockfile()` and `funlockfile()`.

The `ftrylockfile()` function shall return zero for success and non-zero to indicate that the lock cannot be acquired.

## ERRORS

No errors are defined.

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

Applications using these functions may be subject to priority inversion, as discussed in XBD [3.275 Priority Inversion](#).

A call to `_exit()` can block until locked streams are unlocked because a thread having ownership of a (`FILE *`) object blocks all function calls that reference that (`FILE *`) object (except those with names ending in `_unlocked`) from other threads, including calls to `_exit()`.

Note: a `FILE` lock is not a file lock (see XBD [3.143 File Lock](#)).

## RATIONALE

The `flockfile()` and `funlockfile()` functions provide an orthogonal mutual-exclusion lock for each `FILE`. The `ftrylockfile()` function provides a non-blocking attempt to acquire a `FILE` lock, analogous to `pthread_mutex_trylock()`.

These locks behave as if they are the same as those used internally by `stdio` for thread-safety. This both provides thread-safety of these functions without requiring a second level of internal locking and allows functions in `stdio` to be implemented in terms of other `stdio` functions.

Application developers and implementors should be aware that there are potential deadlock problems on `FILE` objects. For example, the line-buffered flushing semantics of `stdio` (requested via `{_IOLBF}`) require that certain input

operations sometimes cause the buffered contents of implementation-defined line-buffered output streams to be flushed. If two threads each hold the lock on the other's `FILE`, deadlock ensues. This type of deadlock can be avoided by acquiring `FILE` locks in a consistent order. In particular, the line-buffered output stream deadlock can typically be avoided by acquiring locks on input streams before locks on output streams if a thread would be acquiring both.

In summary, threads sharing `stdio` streams with other threads can use `flockfile()` and `funlockfile()` to cause sequences of I/O performed by a single thread to be kept bundled. The only case where the use of `flockfile()` and `funlockfile()` is required is to provide a scope protecting uses of the `*_unlocked` functions/macros. This moves the cost/performance tradeoff to the optimal point.

## FUTURE DIRECTIONS

None.

## SEE ALSO

`_exit()`, `getc_unlocked()`

XBD 3.275 Priority Inversion, `<stdio.h>`

## CHANGE HISTORY

### First released in Issue 5

Included for alignment with the POSIX Threads Extension.

### Issue 6

These functions are marked as part of the Thread-Safe Functions option.

### Issue 7

The `flockfile()`, `ftrylockfile()`, and `funlockfile()` functions are moved from the Thread-Safe Functions option to the Base.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0140 [118] is applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0116 [611] is applied.

## Issue 8

Austin Group Defect 1118 is applied, clarifying that a `FILE` lock is not a file lock.

Austin Group Defect 1302 is applied, replacing parts of the text with a reference to [2.5 Standard I/O Streams](#).

## 1.55. flockfile, ftrylockfile, funlockfile — stdio locking functions

---

### SYNOPSIS

```
#include <stdio.h>

void flockfile(FILE *file);
int ftrylockfile(FILE *file);
void funlockfile(FILE *file);
```

### DESCRIPTION

These functions shall provide for explicit application-level locking of the locks associated with standard I/O streams (see [2.5 Standard I/O Streams](#)). These functions can be used by a thread to delineate a sequence of I/O statements that are executed as a unit.

The `flockfile()` function shall acquire for a thread ownership of a `(FILE *)` object.

The `ftrylockfile()` function shall acquire for a thread ownership of a `(FILE *)` object if the object is available; `ftrylockfile()` is a non-blocking version of `flockfile()`.

The `funlockfile()` function shall relinquish the ownership granted to the thread. The behavior is undefined if a thread other than the current owner calls the `funlockfile()` function.

The functions shall behave as if there is a lock count associated with each `(FILE *)` object. This count is implicitly initialized to zero when the `(FILE *)` object is created. The `(FILE *)` object is unlocked when the count is zero. When the count is positive, a single thread owns the `(FILE *)` object. When the `flockfile()` function is called, if the count is zero or if the count is positive and the caller owns the `(FILE *)` object, the count shall be incremented. Otherwise, the calling thread shall be suspended, waiting for the count to return to zero. Each call to `funlockfile()` shall decrement the count. This allows matching calls to `flockfile()` (or successful calls to `ftrylockfile()`) and `funlockfile()` to be nested.

## RETURN VALUE

None for `flockfile()` and `funlockfile()`.

The `ftrylockfile()` function shall return zero for success and non-zero to indicate that the lock cannot be acquired.

## ERRORS

No errors are defined.

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

Applications using these functions may be subject to priority inversion, as discussed in XBD [3.275 Priority Inversion](#).

A call to `exit()` can block until locked streams are unlocked because a thread having ownership of a (`FILE *`) object blocks all function calls that reference that (`FILE *`) object (except those with names ending in `_unlocked`) from other threads, including calls to `exit()`.

**Note:** a `FILE` lock is not a file lock (see XBD [3.143 File Lock](#)).

## RATIONALE

The `flockfile()` and `funlockfile()` functions provide an orthogonal mutual-exclusion lock for each `FILE`. The `ftrylockfile()` function provides a non-blocking attempt to acquire a `FILE` lock, analogous to `pthread_mutex_trylock()`.

These locks behave as if they are the same as those used internally by `stdio` for thread-safety. This both provides thread-safety of these functions without requiring a second level of internal locking and allows functions in `stdio` to be implemented in terms of other `stdio` functions.

Application developers and implementors should be aware that there are potential deadlock problems on `FILE` objects. For example, the line-buffered flushing

semantics of `stdio` (requested via `{_IOLBF}`) require that certain input operations sometimes cause the buffered contents of implementation-defined line-buffered output streams to be flushed. If two threads each hold the lock on the other's `FILE`, deadlock ensues. This type of deadlock can be avoided by acquiring `FILE` locks in a consistent order. In particular, the line-buffered output stream deadlock can typically be avoided by acquiring locks on input streams before locks on output streams if a thread would be acquiring both.

In summary, threads sharing `stdio` streams with other threads can use `flockfile()` and `funlockfile()` to cause sequences of I/O performed by a single thread to be kept bundled. The only case where the use of `flockfile()` and `funlockfile()` is required is to provide a scope protecting uses of the `*_unlocked` functions/macros. This moves the cost/performance tradeoff to the optimal point.

## FUTURE DIRECTIONS

None.

## SEE ALSO

`exit()`, `getc_unlocked()`

XBD 3.275 Priority Inversion, `<stdio.h>`

## CHANGE HISTORY

### First released in Issue 5

Included for alignment with the POSIX Threads Extension.

### Issue 6

These functions are marked as part of the Thread-Safe Functions option.

### Issue 7

The `flockfile()`, `ftrylockfile()`, and `funlockfile()` functions are moved from the Thread-Safe Functions option to the Base.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0140 [118] is applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0116 [611] is applied.

## Issue 8

Austin Group Defect 1118 is applied, clarifying that a `FILE` lock is not a file lock.

Austin Group Defect 1302 is applied, replacing parts of the text with a reference to [2.5 Standard I/O Streams](#).

## 1.56. `fwrite` — binary output

---

### SYNOPSIS

```
#include <stdio.h>

size_t fwrite(const void *restrict ptr,
              size_t size,
              size_t nitems,
              FILE *restrict stream);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `fwrite()` function shall write, from the array pointed to by `ptr`, up to `nitems` elements whose size is specified by `size`, to the stream pointed to by `stream`. For each object, `size` calls shall be made to the `fputc()` function, taking the values (in order) from an array of **unsigned char** exactly overlaying the object. The file-position indicator for the stream (if defined) shall be advanced by the number of bytes successfully written. If an error occurs, the resulting value of the file-position indicator for the stream is unspecified.

[CX] The last data modification and last file status change timestamps of the file shall be marked for update between the successful execution of `fwrite()` and the next successful completion of a call to `fflush()` or `fclose()` on the same stream, or a call to `exit()` or `abort()`.

### RETURN VALUE

The `fwrite()` function shall return the number of elements successfully written, which shall be less than `nitems` only if a write error is encountered. If `size` or `nitems` is 0, `fwrite()` shall return 0 and the state of the stream remains unchanged. Otherwise, if a write error occurs, the error indicator for the stream shall be set, [CX] and `errno` shall be set to indicate the error.

# ERRORS

Refer to [fputc\(\)](#).

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

Because of possible differences in element length and byte ordering, files written using [fwrite\(\)](#) are application-dependent, and possibly cannot be read using [fread\(\)](#) by a different application or by the same application on a different processor.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- 2.5 Standard I/O Streams
- [ferror\(\)](#)
- [fopen\(\)](#)
- [fprintf\(\)](#)
- [putc\(\)](#)
- [puts\(\)](#)
- [write\(\)](#)
- XBD [`<stdio.h>`](#)

# CHANGE HISTORY

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

## Issue 6

Extensions beyond the ISO C standard are marked.

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

- The `fwrite()` prototype is updated.
- The DESCRIPTION is updated to clarify how the data is written out using `fputc()`.

## Issue 7

Changes are made related to support for finegrained timestamps.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0228 [14] is applied.

## Issue 8

Austin Group Defect 1196 is applied, clarifying the RETURN VALUE section.

## 1.57. `getc` — get a byte from a stream

---

### SYNOPSIS

```
#include <stdio.h>

int getc(FILE *stream);
```

### DESCRIPTION

The `getc()` function shall be equivalent to `fgetc()`, except that if it is implemented as a macro it may evaluate *stream* more than once, so the argument should never be an expression with side-effects.

### RETURN VALUE

Refer to `fgetc()`.

### ERRORS

Refer to `fgetc()`.

---

*The following sections are informative.*

### EXAMPLES

None.

### APPLICATION USAGE

If the integer value returned by `getc()` is stored into a variable of type `char` and then compared against the integer constant `EOF`, the comparison may never succeed, because sign-extension of a variable of type `char` on widening to integer is implementation-defined.

Since it may be implemented as a macro, `getc()` may treat incorrectly a *stream* argument with side-effects. In particular, `getc(*_f_++)` does not necessarily work as expected. Therefore, use of this function should be preceded by "#undef `getc`" in such situations; `fgetc()` could also be used.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- 2.5 Standard I/O Streams
- `fgetc()`
- XBD `<stdio.h>`

## CHANGE HISTORY

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

### Issue 7

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0231 [14] is applied.

---

## 1.58. `getc_unlocked`, `getchar_unlocked`, `putc_unlocked`, `putchar_unlocked` — stdio with explicit client locking

---

### SYNOPSIS

```
#include <stdio.h>

int getc_unlocked(FILE *stream);
int getchar_unlocked(void);
int putc_unlocked(int c, FILE *stream);
int putchar_unlocked(int c);
```

### DESCRIPTION

Versions of the functions `getc()`, `getchar()`, `putc()`, and `putchar()` respectively named `getc_unlocked()`, `getchar_unlocked()`, `putc_unlocked()`, and `putchar_unlocked()` shall be provided which are functionally equivalent to the original versions, with the exception that they are not required to be implemented in a fully thread-safe manner. They shall be thread-safe when used within a scope protected by `flockfile()` (or `ftrylockfile()`) and `funlockfile()`. These functions can safely be used in a multi-threaded program if and only if they are called while the invoking thread owns the `(FILE *)` object, as is the case after a successful call to the `flockfile()` or `ftrylockfile()` functions.

If `getc_unlocked()` or `putc_unlocked()` are implemented as macros they may evaluate `stream` more than once, so the `stream` argument should never be an expression with side-effects.

### RETURN VALUE

See `getc()`, `getchar()`, `putc()`, and `putchar()`.

### ERRORS

See `getc()`, `getchar()`, `putc()`, and `putchar()`.

---

## EXAMPLES

None.

## APPLICATION USAGE

Since they may be implemented as macros, `getc_unlocked()` and `putc_unlocked()` may treat incorrectly a `stream` argument with side-effects. In particular, `getc_unlocked(*f++)` and `putc_unlocked(c,*f++)` do not necessarily work as expected. Therefore, use of these functions in such situations should be preceded by the following statement as appropriate:

```
#undef getc_unlocked
#undef putc_unlocked
```

## RATIONALE

Some I/O functions are typically implemented as macros for performance reasons (for example, `putc()` and `getc()`). For safety, they need to be synchronized, but it is often too expensive to synchronize on every character. Nevertheless, it was felt that the safety concerns were more important; consequently, the `getc()`, `getchar()`, `putc()`, and `putchar()` functions are required to be thread-safe. However, unlocked versions are also provided with names that clearly indicate the unsafe nature of their operation but can be used to exploit their higher performance. These unlocked versions can be safely used only within explicitly locked program regions, using exported locking primitives. In particular, a sequence such as:

```
flockfile(fileptr);
putc_unlocked('1', fileptr);
putc_unlocked('\n', fileptr);
fprintf(fileptr, "Line 2\n");
funlockfile(fileptr);
```

is permissible, and results in the text sequence:

```
1
Line 2
```

being printed without being interspersed with output from other threads.

It would be wrong to have the standard names such as `getc()`, `putc()`, and so on, map to the "faster, but unsafe" rather than the "slower, but safe" versions. In

either case, you would still want to inspect all uses of `getc()`, `putc()`, and so on, by hand when converting existing code. Choosing the safe bindings as the default, at least, results in correct code and maintains the "atomicity at the function" invariant. To do otherwise would introduce gratuitous synchronization errors into converted code. Other routines that modify the **stdio** (**FILE \***) structures or buffers are also safely synchronized.

Note that there is no need for functions of the form `getc_locked()`, `putc_locked()`, and so on, since this is the functionality of `getc()`, `putc()`, et al. It would be inappropriate to use a feature test macro to switch a macro definition of `getc()` between `getc_locked()` and `getc_unlocked()`, since the ISO C standard requires an actual function to exist, a function whose behavior could not be changed by the feature test macro. Also, providing both the `xxx_locked()` and `xxx_unlocked()` forms leads to the confusion of whether the suffix describes the behavior of the function or the circumstances under which it should be used.

Three additional routines, `flockfile()`, `ftrylockfile()`, and `funlockfile()` (which may be macros), are provided to allow the user to delineate a sequence of I/O statements that are executed synchronously.

The `ungetc()` function is infrequently called relative to the other functions/macros so no unlocked variation is needed.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [2.5 Standard I/O Streams](#)
- [flockfile\(\)](#)
- [getc\(\)](#)
- [getchar\(\)](#)
- [putc\(\)](#)
- [putchar\(\)](#)
-

# CHANGE HISTORY

## First released in Issue 5

Included for alignment with the POSIX Threads Extension.

## Issue 6

These functions are marked as part of the Thread-Safe Functions option.

The Open Group Corrigendum U030/2 is applied, adding APPLICATION USAGE describing how applications should be written to avoid the case when the functions are implemented as macros.

## Issue 7

The `getc_unlocked()`, `getchar_unlocked()`, `putc_unlocked()`, and `putchar_unlocked()` functions are moved from the Thread-Safe Functions option to the Base.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0232 [395], XSH/TC1-2008/0233 [395], XSH/TC1-2008/0234 [395], and XSH/TC1-2008/0235 [14] are applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0151 [826] is applied.

---

## 1.59. getchar

---

### SYNOPSIS

```
#include <stdio.h>

int getchar(void);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `getchar()` function shall be equivalent to `getc(stdin)`.

### RETURN VALUE

Refer to `fgetc()`.

### ERRORS

Refer to `fgetc()`.

---

*The following sections are informative.*

### EXAMPLES

None.

### APPLICATION USAGE

If the integer value returned by `getchar()` is stored into a variable of type `char` and then compared against the integer constant EOF, the comparison may

never succeed, because sign-extension of a variable of type `char` on widening to integer is implementation-defined.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [2.5 Standard I/O Streams](#)
- `getc()`
- XBD `<stdio.h>`

## CHANGE HISTORY

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

### Issue 7

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0236 [14] is applied.

---

## 1.60. `getc_unlocked`, `getchar_unlocked`, `putc_unlocked`, `putchar_unlocked` — stdio with explicit client locking

---

### SYNOPSIS

```
[CX] #include <stdio.h>

int getc_unlocked(FILE *stream);
int getchar_unlocked(void);
int putc_unlocked(int c, FILE *stream);
int putchar_unlocked(int c);
```

### DESCRIPTION

Versions of the functions `getc()`, `getchar()`, `putc()`, and `putchar()` respectively named `getc_unlocked()`, `getchar_unlocked()`, `putc_unlocked()`, and `putchar_unlocked()` shall be provided which are functionally equivalent to the original versions, with the exception that they are not required to be implemented in a fully thread-safe manner. They shall be thread-safe when used within a scope protected by `flockfile()` (or `ftrylockfile()`) and `funlockfile()`. These functions can safely be used in a multi-threaded program if and only if they are called while the invoking thread owns the `(FILE *)` object, as is the case after a successful call to the `flockfile()` or `ftrylockfile()` functions.

If `getc_unlocked()` or `putc_unlocked()` are implemented as macros they may evaluate `stream` more than once, so the `stream` argument should never be an expression with side-effects.

### RETURN VALUE

See `getc()`, `getchar()`, `putc()`, and `putchar()`.

### ERRORS

See `getc()`, `getchar()`, `putc()`, and `putchar()`.

---

The following sections are informative.

## EXAMPLES

None.

## APPLICATION USAGE

Since they may be implemented as macros, `getc_unlocked()` and `putc_unlocked()` may treat incorrectly a *stream* argument with side-effects. In particular, `getc_unlocked(*f++)` and `putc_unlocked(c,*f++)` do not necessarily work as expected. Therefore, use of these functions in such situations should be preceded by the following statement as appropriate:

```
#undef getc_unlocked
#undef putc_unlocked
```

## RATIONALE

Some I/O functions are typically implemented as macros for performance reasons (for example, `putc()` and `getc()`). For safety, they need to be synchronized, but it is often too expensive to synchronize on every character. Nevertheless, it was felt that the safety concerns were more important; consequently, the `getc()`, `getchar()`, `putc()`, and `putchar()` functions are required to be thread-safe. However, unlocked versions are also provided with names that clearly indicate the unsafe nature of their operation but can be used to exploit their higher performance. These unlocked versions can be safely used only within explicitly locked program regions, using exported locking primitives. In particular, a sequence such as:

```
flockfile(fileptr);
putc_unlocked('1', fileptr);
putc_unlocked('\n', fileptr);
fprintf(fileptr, "Line 2\n");
funlockfile(fileptr);
```

is permissible, and results in the text sequence:

```
1
Line 2
```

being printed without being interspersed with output from other threads.

It would be wrong to have the standard names such as `getc()`, `putc()`, and so on, map to the "faster, but unsafe" rather than the "slower, but safe" versions. In either case, you would still want to inspect all uses of `getc()`, `putc()`, and so on, by hand when converting existing code. Choosing the safe bindings as the default, at least, results in correct code and maintains the "atomicity at the function" invariant. To do otherwise would introduce gratuitous synchronization errors into converted code. Other routines that modify the `stdio` (`FILE *`) structures or buffers are also safely synchronized.

Note that there is no need for functions of the form `getc_locked()`, `putc_locked()`, and so on, since this is the functionality of `getc()`, `putc()`, *et al.* It would be inappropriate to use a feature test macro to switch a macro definition of `getc()` between `getc_locked()` and `getc_unlocked()`, since the ISO C standard requires an actual function to exist, a function whose behavior could not be changed by the feature test macro. Also, providing both the `xxx_locked()` and `xxx_unlocked()` forms leads to the confusion of whether the suffix describes the behavior of the function or the circumstances under which it should be used.

Three additional routines, `flockfile()`, `ftrylockfile()`, and `funlockfile()` (which may be macros), are provided to allow the user to delineate a sequence of I/O statements that are executed synchronously.

The `ungetc()` function is infrequently called relative to the other functions/macros so no unlocked variation is needed.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [2.5 Standard I/O Streams](#)
- [flockfile\(\)](#)
- [getc\(\)](#)
- [getchar\(\)](#)
- [putc\(\)](#)
- [putchar\(\)](#)

**XBD**

# CHANGE HISTORY

**First released in Issue 5.** Included for alignment with the POSIX Threads Extension.

## Issue 6

These functions are marked as part of the Thread-Safe Functions option.

The Open Group Corrigendum U030/2 is applied, adding APPLICATION USAGE describing how applications should be written to avoid the case when the functions are implemented as macros.

## Issue 7

The `getc_unlocked()`, `getchar_unlocked()`, `putc_unlocked()`, and `putchar_unlocked()` functions are moved from the Thread-Safe Functions option to the Base.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0232 [395], XSH/TC1-2008/0233 [395], XSH/TC1-2008/0234 [395], and XSH/TC1-2008/0235 [14] are applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0151 [826] is applied.

## 1.61. getenv, secure\_getenv — get value of an environment variable

---

### SYNOPSIS

```
#include <stdlib.h>

char *getenv(const char *name);

[CX] char *secure_getenv(const char *name);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `getenv()` function shall search the environment of the calling process (see XBD 8. Environment Variables) for the environment variable *name* if it exists and return a pointer to the value of the environment variable. If the specified environment variable cannot be found, a null pointer shall be returned. The application shall ensure that it does not modify the string pointed to by the `getenv()` function, [CX] unless it is part of a modifiable object previously placed in the environment by assigning a new value to `environ` [XSI] or by using `putenv()`.

[CX] The pointer returned by `getenv()` shall point to a string within the environment data pointed to by `environ`.

#### Note:

*This requirement is an extension to the ISO C standard, which allows `getenv()` to copy the data to an internal buffer.*

The `secure_getenv()` function shall be equivalent to `getenv()`, except that it shall return a null pointer if the calling process does not meet all of the following security criteria:

1. The effective user ID and real user ID of the calling process were equal during program startup.
2. The effective group ID and real group ID of the calling process were equal during program startup.
3. Additional implementation-defined security criteria.

## RETURN VALUE

Upon successful completion, `getenv()` shall return a pointer to a string containing the value for the specified *name*. If the specified *name* cannot be found in the environment of the calling process, a null pointer shall be returned.

[CX] Upon successful completion, `secure_getenv()` shall return a pointer to a string containing the value for the specified *name*. If the specified *name* cannot be found in the environment of the calling process, or the calling process does not meet the security criteria listed in DESCRIPTION, a null pointer shall be returned.

## ERRORS

No errors are defined.

---

## EXAMPLES

### Getting the Value of an Environment Variable

The following example gets the value of the HOME environment variable.

```
#include <stdlib.h>
...
const char *name = "HOME";
char *value;

value = getenv(name);
```

---

## APPLICATION USAGE

None.

---

## RATIONALE

The `clearenv()` function was considered but rejected. The `putenv()` function has now been included for alignment with the Single UNIX Specification.

Some earlier versions of this standard did not require `getenv()` to be thread-safe because it was allowed to return a value pointing to an internal buffer. However, this behavior allowed by the ISO C standard is no longer allowed by POSIX.1. POSIX.1 requires the environment data to be available through `environ[]`, so there is no reason why `getenv()` can't return a pointer to the actual data instead of a copy. Therefore `getenv()` is now required to be thread-safe (except when another thread modifies the environment).

Conforming applications are required not to directly modify the pointers to which `environ` points, but to use only the `setenv()`, `unsetenv()`, and `putenv()` functions, or assignment to `environ` itself, to manipulate the process environment. This constraint allows the implementation to properly manage the memory it allocates. This enables the implementation to free any space it has allocated to strings (and perhaps the pointers to them) stored in `environ` when `unsetenv()` is called. A C runtime start-up procedure (that which invokes `main()` and perhaps initializes `environ`) can also initialize a flag indicating that none of the environment has yet been copied to allocated storage, or that the separate table has not yet been initialized. If the application switches to a complete new environment by assigning a new value to `environ`, this can be detected by `getenv()`, `setenv()`, `unsetenv()`, or `putenv()` and the implementation can at that point reinitialize based on the new environment. (This may include copying the environment strings into a new array and assigning `environ` to point to it.)

In fact, for higher performance of `getenv()`, implementations that do not provide `putenv()` could also maintain a separate copy of the environment in a data structure that could be searched much more quickly (such as an indexed hash table, or a binary tree), and update both it and the linear list at `environ` when `setenv()` or `unsetenv()` is invoked. On implementations that do provide `putenv()`, such a copy might still be worthwhile but would need to allow for the fact that applications can directly modify the content of environment strings added with `putenv()`. For example, if an environment string found by searching the copy is one that was added using `putenv()`, the implementation would need to check that the string in `environ` still has the same name (and value, if the copy includes values), and whenever searching the copy produces no match the implementation would then need to search each environment string in `environ` that was added using `putenv()` in case any of them have changed their names

and now match. Thus, each use of `putenv()` to add to the environment would reduce the speed advantage of having the copy.

Performance of `getenv()` can be important for applications which have large numbers of environment variables. Typically, applications like this use the environment as a resource database of user-configurable parameters. The fact that these variables are in the user's shell environment usually means that any other program that uses environment variables (such as `ls`, which attempts to use `COLUMNS`), or really almost any utility (`LANG`, `LC_ALL`, and so on) is similarly slowed down by the linear search through the variables.

An implementation that maintains separate data structures, or even one that manages the memory it consumes, is not currently required as it was thought it would reduce consensus among implementors who do not want to change their historical implementations.

---

## FUTURE DIRECTIONS

None.

---

## SEE ALSO

- `exec()`
- `putenv()`
- `setenv()`
- `unsetenv()`

XBD 8. Environment Variables, `<stdlib.h>`

---

## CHANGE HISTORY

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

### Issue 5

Normative text previously in the APPLICATION USAGE section is moved to the RETURN VALUE section.

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

## Issue 6

The following changes were made to align with the IEEE P1003.1a draft standard:

- References added to the new `setenv()` and `unsetenv()` functions.

The normative text is updated to avoid use of the term "must" for application requirements.

## Issue 7

Austin Group Interpretation 1003.1-2001 #062 is applied, clarifying that a call to `putenv()` may also cause the string to be overwritten.

Austin Group Interpretation 1003.1-2001 #148 is applied, adding the FUTURE DIRECTIONS.

Austin Group Interpretation 1003.1-2001 #156 is applied.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0238 [75,428], XSH/TC1-2008/0239 [167], and XSH/TC1-2008/0240 [167] are applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0157 [656] is applied.

## Issue 8

Austin Group Defects 188 and 1394 are applied, changing `getenv()` to be thread-safe.

Austin Group Defect 922 is applied, adding the `secure_getenv()` function.

---

## 1.62. gets

---

### SYNOPSIS

```
#include <stdio.h>

char *gets(char *s);
```

### DESCRIPTION

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2017 defers to the ISO C standard.

The `gets()` function shall read bytes from the standard input stream, `stdin`, into the array pointed to by `s`, until a `<newline>` is read or an end-of-file condition is encountered. Any `<newline>` shall be discarded and a null byte shall be placed immediately after the last byte read into the array.

The `gets()` function may mark the last data access timestamp of the file associated with `stream` for update. The last data access timestamp shall be marked for update by the first successful execution of `fgetc()`, `fgets()`, `fread()`, `fscanf()`, `getc()`, `getchar()`, `getdelim()`, `getline()`, `gets()`, or `scanf()` using `stream` that returns data not supplied by a prior call to `ungetc()`.

### RETURN VALUE

Upon successful completion, `gets()` shall return `s`. If the end-of-file indicator for the stream is set, or if the stream is at end-of-file, the end-of-file indicator for the stream shall be set and `gets()` shall return a null pointer. If a read error occurs, the error indicator for the stream shall be set, `gets()` shall return a null pointer, and set `errno` to indicate the error.

### ERRORS

Refer to `fgetc()`.

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

Reading a line that overflows the array pointed to by `s` results in undefined behavior. The use of `fgets()` is recommended.

Since the user cannot specify the length of the buffer passed to `gets()`, use of this function is discouraged. The length of the string read is unlimited. It is possible to overflow this buffer in such a way as to cause applications to fail, or possible system security violations.

Applications should use the `fgets()` function instead of the obsolescent `gets()` function.

## RATIONALE

The standard developers decided to mark the `gets()` function as obsolescent even though it is in the ISO C standard due to the possibility of buffer overflow.

## FUTURE DIRECTIONS

The `gets()` function may be removed in a future version.

## SEE ALSO

- [Standard I/O Streams](#)
- `feof()`
- `ferror()`
- `fgets()`
-

# CHANGE HISTORY

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

## Issue 6

Extensions beyond the ISO C standard are marked.

## Issue 7

Austin Group Interpretation 1003.1-2001 #051 is applied, clarifying the RETURN VALUE section.

The `gets()` function is marked obsolescent.

Changes are made related to support for finegrained timestamps.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0257 [14] is applied.

## 1.63. gmtime, gmtime\_r — convert a time value to a broken-down UTC time

---

### SYNOPSIS

```
#include <time.h>

struct tm *gmtime(const time_t *timer);

[CX] struct tm *gmtime_r(const time_t *restrict timer,
                         struct tm *restrict result);
```

### DESCRIPTION

For `gmtime()`:

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `gmtime()` function shall convert the time in seconds since the Epoch pointed to by `timer` into a broken-down time, expressed as Coordinated Universal Time (UTC).

[CX] The relationship between a time in seconds since the Epoch used as an argument to `gmtime()` and the `tm` structure (defined in the `<time.h>` header) is that the result shall be as specified in the expression given in the definition of seconds since the Epoch (see XBD 4.19 Seconds Since the Epoch), where the names in the structure and in the expression correspond.

The same relationship shall apply for `gmtime_r()`.

The `gmtime()` function need not be thread-safe; however, `gmtime()` shall avoid data races with all functions other than itself, `asctime()`, `ctime()`, and `localtime()`.

The `asctime()`, `ctime()`, `gmtime()`, and `localtime()` functions shall return values in one of two static objects: a broken-down time structure and an array of type `char`. Execution of any of the functions that return a pointer to one of these object types may overwrite the information in any object of the same type pointed to by the value returned from any previous call to any of them.

[CX] The `gmtime_r()` function shall convert the time in seconds since the Epoch pointed to by `timer` into a broken-down time expressed as Coordinated

Universal Time (UTC). The broken-down time is stored in the structure referred to by `result`. The `gmtime_r()` function shall also return the address of the same structure.

## RETURN VALUE

Upon successful completion, the `gmtime()` function shall return a pointer to a `struct tm`. If an error is detected, `gmtime()` shall return a null pointer [CX] and set `errno` to indicate the error.

Upon successful completion, `gmtime_r()` shall return the address of the structure pointed to by the argument `result`. The structure's `tm_zone` member shall be set to a pointer to the string "UTC", which shall have static storage duration. If an error is detected, `gmtime_r()` shall return a null pointer and set `errno` to indicate the error.

## ERRORS

The `gmtime()` [CX] and `gmtime_r()` functions shall fail if:

- **[OVERFLOW]** [CX] The result cannot be represented.

## EXAMPLES

None.

## APPLICATION USAGE

The `gmtime_r()` function is thread-safe and returns values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `asctime()`
- `clock()`
- `ctime()`
- `difftime()`
- `futimens()`
- `localtime()`
- `mktime()`
- `strftime()`
- `strptime()`
- `time()`

XBD 4.19 Seconds Since the Epoch, `<time.h>`

## CHANGE HISTORY

### First released in Issue 1

Derived from Issue 1 of the SVID.

### Issue 5

- A note indicating that the `gmtime()` function need not be reentrant is added to the DESCRIPTION.
- The `gmtime_r()` function is included for alignment with the POSIX Threads Extension.

### Issue 6

- The `gmtime_r()` function is marked as part of the Thread-Safe Functions option.
- Extensions beyond the ISO C standard are marked.
- The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

- The `restrict` keyword is added to the `gmtime_r()` prototype for alignment with the ISO/IEC 9899:1999 standard.
- IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/27 is applied, adding the [EOVERFLOW] error.
- IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/48 is applied, updating the error handling for `gmtime_r()`.

## Issue 7

- Austin Group Interpretation 1003.1-2001 #156 is applied.
- The `gmtime_r()` function is moved from the Thread-Safe Functions option to the Base.

## Issue 8

- Austin Group Defect 1302 is applied, aligning the `gmtime()` function with the ISO/IEC 9899:2018 standard.
  - Austin Group Defect 1376 is applied, removing CX shading from some text derived from the ISO C standard and updating it to match the ISO C standard.
  - Austin Group Defect 1533 is applied, adding `tm_gmtoff` and `tm_zone` to the `tm` structure.
-

## 1.64. gmtime, gmtime\_r — convert a time value to a broken-down UTC time

---

### SYNOPSIS

```
#include <time.h>

struct tm *gmtime(const time_t *timer);

[CX] struct tm *gmtime_r(const time_t *restrict timer,
                         struct tm *restrict result);
```

### DESCRIPTION

For `gmtime()` : [CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `gmtime()` function shall convert the time in seconds since the Epoch pointed to by `timer` into a broken-down time, expressed as Coordinated Universal Time (UTC).

[CX] The relationship between a time in seconds since the Epoch used as an argument to `gmtime()` and the `tm` structure (defined in the `<time.h>` header) is that the result shall be as specified in the expression given in the definition of seconds since the Epoch (see XBD 4.19 Seconds Since the Epoch), where the names in the structure and in the expression correspond.

The same relationship shall apply for `gmtime_r()`.

The `gmtime()` function need not be thread-safe; however, `gmtime()` shall avoid data races with all functions other than itself, `asctime()`, `ctime()`, and `localtime()`.

The `asctime()`, `ctime()`, `gmtime()`, and `localtime()` functions shall return values in one of two static objects: a broken-down time structure and an array of type `char`. Execution of any of the functions that return a pointer to one of these object types may overwrite the information in any object of the same type pointed to by the value returned from any previous call to any of them.

[CX] The `gmtime_r()` function shall convert the time in seconds since the Epoch pointed to by `timer` into a broken-down time expressed as Coordinated Universal Time (UTC). The broken-down time is stored in the structure referred to

by `result`. The `gmtime_r()` function shall also return the address of the same structure.

## RETURN VALUE

Upon successful completion, the `gmtime()` function shall return a pointer to a `struct tm`. If an error is detected, `gmtime()` shall return a null pointer [CX] and set `errno` to indicate the error.

Upon successful completion, `gmtime_r()` shall return the address of the structure pointed to by the argument `result`. The structure's `tm_zone` member shall be set to a pointer to the string "UTC", which shall have static storage duration. If an error is detected, `gmtime_r()` shall return a null pointer and set `errno` to indicate the error.

## ERRORS

The `gmtime()` [CX] and `gmtime_r()` functions shall fail if:

- **[OVERFLOW]** [CX] The result cannot be represented.

## EXAMPLES

None.

## APPLICATION USAGE

The `gmtime_r()` function is thread-safe and returns values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

`asctime()` , `clock()` , `ctime()` , `difftime()` , `futimens()` ,  
`localtime()` , `mktime()` , `strftime()` , `strptime()` , `time()`

XBD 4.19 Seconds Since the Epoch, `<time.h>`

## CHANGE HISTORY

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

### Issue 5

A note indicating that the `gmtime()` function need not be reentrant is added to the DESCRIPTION.

The `gmtime_r()` function is included for alignment with the POSIX Threads Extension.

### Issue 6

The `gmtime_r()` function is marked as part of the Thread-Safe Functions option.

Extensions beyond the ISO C standard are marked.

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

The `restrict` keyword is added to the `gmtime_r()` prototype for alignment with the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/27 is applied, adding the [EOVERFLOW] error.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/48 is applied, updating the error handling for `gmtime_r()`.

### Issue 7

Austin Group Interpretation 1003.1-2001 #156 is applied.

The `gmtime_r()` function is moved from the Thread-Safe Functions option to the Base.

## Issue 8

Austin Group Defect 1302 is applied, aligning the `gmtime()` function with the ISO/IEC 9899:2018 standard.

Austin Group Defect 1376 is applied, removing CX shading from some text derived from the ISO C standard and updating it to match the ISO C standard.

Austin Group Defect 1533 is applied, adding `tm_gmtoff` and `tm_zone` to the `tm` structure.

## 1.65. `imaxabs` — return absolute value

---

### SYNOPSIS

```
#include <inttypes.h>

intmax_t imaxabs(intmax_t j);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `imaxabs()` function shall compute the absolute value of an integer `j`. If the result cannot be represented, the behavior is undefined.

### RETURN VALUE

The `imaxabs()` function shall return the absolute value.

### ERRORS

No errors are defined.

---

### EXAMPLES

None.

### APPLICATION USAGE

Since POSIX.1 requires a two's complement representation of `intmax_t`, the absolute value of the negative integer with the largest magnitude {INTMAX\_MIN} is not representable, thus `imaxabs(INTMAX_MIN)` is undefined.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `imaxdiv()`
- XBD `<inttypes.h>`

## CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

### Issue 8

Austin Group Defect 1108 is applied, changing the APPLICATION USAGE section.

## 1.66. imaxdiv

---

### SYNOPSIS

```
#include <inttypes.h>

imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `imaxdiv()` function shall compute `numer / denom` and `numer % denom` in a single operation.

### RETURN VALUE

The `imaxdiv()` function shall return a structure of type `imaxdiv_t`, comprising both the quotient and the remainder. The structure shall contain (in either order) the members `quot` (the quotient) and `rem` (the remainder), each of which has type `intmax_t`.

If either part of the result cannot be represented, the behavior is undefined.

### ERRORS

No errors are defined.

---

*The following sections are informative.*

### EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `imaxabs()`
- XBD `<inttypes.h>`

## CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

---

## 1.67. isalnum, isalnum\_l — test for an alphanumeric character

---

### SYNOPSIS

```
#include <ctype.h>

int isalnum(int c);

[CX] int isalnum_l(int c, locale_t locale);
```

### DESCRIPTION

For `isalnum()`: [CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `isalnum()` [CX] and `isalnum_l()` functions shall test whether `c` is a character of class **alpha** or **digit** in the current locale, [CX] or in the locale represented by `locale`, respectively; see XBD 7. Locale.

The `c` argument is an `int`, the value of which the application shall ensure is representable as an `unsigned char` or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined.

[CX] The behavior is undefined if the `locale` argument to `isalnum_l()` is the special locale object LC\_GLOBAL\_LOCALE or is not a valid locale object handle.

### RETURN VALUE

The `isalnum()` [CX] and `isalnum_l()` functions shall return non-zero if `c` is an alphanumeric character; otherwise, they shall return 0.

### ERRORS

No errors are defined.

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

To ensure applications portability, especially across natural languages, only these functions and the functions in the reference pages listed in the SEE ALSO section should be used for character classification.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [isalpha\(\)](#)
- [isblank\(\)](#)
- [iscntrl\(\)](#)
- [isdigit\(\)](#)
- [isgraph\(\)](#)
- [islower\(\)](#)
- [isprint\(\)](#)
- [ispunct\(\)](#)
- [isspace\(\)](#)
- [isupper\(\)](#)
- [isxdigit\(\)](#)
- [setlocale\(\)](#)

- `uselocale()`

XBD 7. Locale, `<ctype.h>`, `<stdio.h>`

## CHANGE HISTORY

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

### Issue 6

The normative text is updated to avoid use of the term "must" for application requirements.

### Issue 7

The `isalnum_l()` function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0274 [302], XSH/TC1-2008/0275 [283], and XSH/TC1-2008/0276 [283] are applied.

---

## 1.68. `isalpha`, `isalpha_l` — test for an alphabetic character

---

### SYNOPSIS

```
#include <ctype.h>

int isalpha(int c);

[CX] int isalpha_l(int c, locale_t locale);
```

### DESCRIPTION

For `isalpha()`: [CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `isalpha()` [CX] and `isalpha_l()` functions shall test whether `c` is a character of class **alpha** in the current locale, [CX] or in the locale represented by `locale`, respectively; see XBD 7. Locale.

The `c` argument is an `int`, the value of which the application shall ensure is representable as an `unsigned char` or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined.

[CX] The behavior is undefined if the `locale` argument to `isalpha_l()` is the special locale object LC\_GLOBAL\_LOCALE or is not a valid locale object handle.

### RETURN VALUE

The `isalpha()` [CX] and `isalpha_l()` functions shall return non-zero if `c` is an alphabetic character; otherwise, they shall return 0.

### ERRORS

No errors are defined.

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

To ensure applications portability, especially across natural languages, only these functions and the functions in the reference pages listed in the SEE ALSO section should be used for character classification.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [isalnum\(\)](#)
- [isblank\(\)](#)
- [iscntrl\(\)](#)
- [isdigit\(\)](#)
- [isgraph\(\)](#)
- [islower\(\)](#)
- [isprint\(\)](#)
- [ispunct\(\)](#)
- [isspace\(\)](#)
- [isupper\(\)](#)
- [isxdigit\(\)](#)
- [setlocale\(\)](#)

- `uselocale()`

XBD 7. Locale, `<ctype.h>`, `<locale.h>`, `<stdio.h>`

## CHANGE HISTORY

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

### Issue 6

The normative text is updated to avoid use of the term "must" for application requirements.

### Issue 7

The `isalpha_l()` function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0277 [302], XSH/TC1-2008/0278 [283], and XSH/TC1-2008/0279 [283] are applied.

---

## 1.69. `isblank`, `isblank_l` — test for a blank character

---

### SYNOPSIS

```
#include <ctype.h>

int isblank(int c);

[CX] int isblank_l(int c, locale_t locale);
```

### DESCRIPTION

For `isblank()` : [CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `isblank()` [CX] and `isblank_l()` functions shall test whether `c` is a character of class **blank** in the current locale, [CX] or in the locale represented by `locale`, respectively; see XBD [7. Locale].

The `c` argument is a type `int`, the value of which the application shall ensure is a character representable as an `unsigned char` or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined.

[CX] The behavior is undefined if the `locale` argument to `isblank_l()` is the special locale object LC\_GLOBAL\_LOCALE or is not a valid locale object handle.

### RETURN VALUE

The `isblank()` [CX] and `isblank_l()` functions shall return non-zero if `c` is a ; otherwise, they shall return 0.

### ERRORS

No errors are defined.

### EXAMPLES

None.

## APPLICATION USAGE

To ensure applications portability, especially across natural languages, only these functions and the functions in the reference pages listed in the SEE ALSO section should be used for character classification.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

`isalnum()` ,  
`isalpha()` ,  
`iscntrl()` ,  
`isdigit()` ,  
`isgraph()` ,  
`islower()` ,  
`isprint()` ,  
`ispunct()` ,  
`isspace()` ,  
`isupper()` ,  
`isxdigit()` ,  
`setlocale()` ,  
`use locale()`

XBD [7. Locale], `<ctype.h>`, `<locale.h>`

## CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

### Issue 7

The `isblank_l()` function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0280 [302], XSH/TC1-2008/0281 [283], and XSH/TC1-2008/0282 [283] are applied.

## 1.70. iscntrl, iscntrl\_l - test for a control character

---

### SYNOPSIS

```
#include <ctype.h>

int iscntrl(int c);

[CX] int iscntrl_l(int c, locale_t locale);
```

### DESCRIPTION

For `iscntrl()`: [CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `iscntrl()` [CX] and `iscntrl_l()` functions shall test whether `c` is a character of class `cntrl` in the current locale, [CX] or in the locale represented by `locale`, respectively; see XBD [7. Locale].

The `c` argument is a type `int`, the value of which the application shall ensure is a character representable as an `unsigned char` or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined.

[CX] The behavior is undefined if the `locale` argument to `iscntrl_l()` is the special locale object LC\_GLOBAL\_LOCALE or is not a valid locale object handle.

### RETURN VALUE

The `iscntrl()` [CX] and `iscntrl_l()` functions shall return non-zero if `c` is a control character; otherwise, they shall return 0.

### ERRORS

No errors are defined.

### EXAMPLES

None.

# APPLICATION USAGE

To ensure applications portability, especially across natural languages, only these functions and the functions in the reference pages listed in the SEE ALSO section should be used for character classification.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [isalnum\(\)](#)
- [isalpha\(\)](#)
- [isblank\(\)](#)
- [isdigit\(\)](#)
- [isgraph\(\)](#)
- [islower\(\)](#)
- [isprint\(\)](#)
- [ispunct\(\)](#)
- [isspace\(\)](#)
- [isupper\(\)](#)
- [isxdigit\(\)](#)
- [setlocale\(\)](#)
- [uselocale\(\)](#)

XBD [7. Locale], [<ctype.h>](#), [<locale.h>](#)

## CHANGE HISTORY

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

## Issue 6

The normative text is updated to avoid use of the term "must" for application requirements.

## Issue 7

The `iscntrl_l()` function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0283 [302], XSH/TC1-2008/0284 [283], and XSH/TC1-2008/0285 [283] are applied.

---

## 1.71. isdigit, isdigit\_l — test for a decimal digit

---

### SYNOPSIS

```
#include <ctype.h>

int isdigit(int c);

[CX] int isdigit_l(int c, locale_t locale);
```

### DESCRIPTION

For `isdigit()` :

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `isdigit()` [CX] and `isdigit_l()` functions shall test whether `c` is a character of class **digit** in the current locale, [CX] or in the locale represented by `locale`, respectively; see XBD [7. Locale](#).

The `c` argument is an `int`, the value of which the application shall ensure is a character representable as an `unsigned char` or equal to the value of the macro `EOF`. If the argument has any other value, the behavior is undefined.

[CX] The behavior is undefined if the `locale` argument to `isdigit_l()` is the special locale object `LC_GLOBAL_LOCALE` or is not a valid locale object handle.

### RETURN VALUE

The `isdigit()` [CX] and `isdigit_l()` functions shall return non-zero if `c` is a decimal digit; otherwise, they shall return 0.

### ERRORS

No errors are defined.

*The following sections are informative.*

# EXAMPLES

None.

# APPLICATION USAGE

To ensure applications portability, especially across natural languages, only these functions and the functions in the reference pages listed in the SEE ALSO section should be used for character classification.

# RATIONALE

None.

# FUTURE DIRECTIONS

None.

# SEE ALSO

- [isalnum\(\)](#)
- [isalpha\(\)](#)
- [isblank\(\)](#)
- [iscntrl\(\)](#)
- [isgraph\(\)](#)
- [islower\(\)](#)
- [isprint\(\)](#)
- [ispunct\(\)](#)
- [isspace\(\)](#)
- [isupper\(\)](#)
- [isxdigit\(\)](#)

XBD [7. Locale](#),

# CHANGE HISTORY

## First released in Issue 1.

Derived from Issue 1 of the SVID.

## Issue 6

The normative text is updated to avoid use of the term "must" for application requirements.

## Issue 7

The `isdigit_l()` function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0286 [302], XSH/TC1-2008/0287 [283], and XSH/TC1-2008/0288 [283] are applied.

## 1.72. `isgraph`, `isgraph_l` — test for a visible character

---

### SYNOPSIS

```
#include <ctype.h>

int isgraph(int c);

[CX] int isgraph_l(int c, locale_t locale);
```

### DESCRIPTION

For `isgraph()` : [CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `isgraph()` [CX] and `isgraph_l()` functions shall test whether `c` is a character of class **graph** in the current locale, [CX] or in the locale represented by `locale`, respectively; see XBD [7. Locale](#).

The `c` argument is an **int**, the value of which the application shall ensure is a character representable as an **unsigned char** or equal to the value of the macro **EOF**. If the argument has any other value, the behavior is undefined.

[CX] The behavior is undefined if the `locale` argument to `isgraph_l()` is the special locale object **LC\_GLOBAL\_LOCALE** or is not a valid locale object handle.

### RETURN VALUE

The `isgraph()` [CX] and `isgraph_l()` functions shall return non-zero if `c` is a character with a visible representation; otherwise, they shall return 0.

### ERRORS

No errors are defined.

---

*The following sections are informative.*

# EXAMPLES

None.

# APPLICATION USAGE

To ensure applications portability, especially across natural languages, only these functions and the functions in the reference pages listed in the SEE ALSO section should be used for character classification.

# RATIONALE

None.

# FUTURE DIRECTIONS

None.

# SEE ALSO

- [isalnum\(\)](#)
- [isalpha\(\)](#)
- [isblank\(\)](#)
- [iscntrl\(\)](#)
- [isdigit\(\)](#)
- [islower\(\)](#)
- [isprint\(\)](#)
- [ispunct\(\)](#)
- [isspace\(\)](#)
- [isupper\(\)](#)
- [isxdigit\(\)](#)
- [setlocale\(\)](#)
- [uselocale\(\)](#)

# CHANGE HISTORY

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

## Issue 6

The normative text is updated to avoid use of the term "must" for application requirements.

## Issue 7

The `isgraph_l()` function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0289 [302], XSH/TC1-2008/0290 [283], and XSH/TC1-2008/0291 [283] are applied.

## 1.73. islower, islower\_l - test for a lowercase letter

---

### SYNOPSIS

```
#include <ctype.h>

int islower(int c);

/* XSI extension */
int islower_l(int c, locale_t locale);
```

### DESCRIPTION

For `islower()`: The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `islower()` and `islower_l()` functions shall test whether `c` is a character of class **lower** in the current locale, or in the locale represented by `locale`, respectively; see XBD 7. Locale.

The `c` argument is an `int`, the value of which the application shall ensure is a character representable as an `unsigned char` or equal to the value of the macro `EOF`. If the argument has any other value, the behavior is undefined.

The behavior is undefined if the `locale` argument to `islower_l()` is the special locale object `LC_GLOBAL_LOCALE` or is not a valid locale object handle.

### RETURN VALUE

The `islower()` and `islower_l()` functions shall return non-zero if `c` is a lowercase letter; otherwise, they shall return 0.

### ERRORS

No errors are defined.

# EXAMPLES

## Testing for a Lowercase Letter

Two examples follow, the first using `islower()`, the second using multiple concurrent locales and `islower_l()`.

The examples test whether the value is a lowercase letter, based on the current locale, then use it as part of a key value.

```
/* Example 1 -- using islower() */
#include <ctype.h>
#include <stdlib.h>
#include <locale.h>
...
char *keystr;
int elementlen, len;
unsigned char c;
...
setlocale(LC_ALL, "");
...
len = 0;
while (len < elementlen) {
    c = (unsigned char) (rand() % 256);
    ...
    if (islower(c))
        keystr[len++] = c;
}
...
...
```

```
/* Example 2 -- using islower_l() */
#include <ctype.h>
#include <stdlib.h>
#include <locale.h>
...
char *keystr;
int elementlen, len;
unsigned char c;
...
locale_t loc = newlocale (LC_ALL_MASK, "", (locale_t) 0);
...
len = 0;
while (len < elementlen) {
    c = (unsigned char) (rand() % 256);
    ...
    if (islower_l(c, loc))
        keystr[len++] = c;
}
```

```
 }  
...
```

## APPLICATION USAGE

To ensure applications portability, especially across natural languages, only these functions and the functions in the reference pages listed in the SEE ALSO section should be used for character classification.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [isalnum\(\)](#)
- [isalpha\(\)](#)
- [isblank\(\)](#)
- [iscntrl\(\)](#)
- [isdigit\(\)](#)
- [isgraph\(\)](#)
- [isprint\(\)](#)
- [ispunct\(\)](#)
- [isspace\(\)](#)
- [isupper\(\)](#)
- [isxdigit\(\)](#)
- [setlocale\(\)](#)
- [use locale\(\)](#)

# CHANGE HISTORY

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

## Issue 6

The normative text is updated to avoid use of the term "must" for application requirements and an example is added.

## Issue 7

The `islower_l()` function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0292 [302], XSH/TC1-2008/0293 [283], XSH/TC1-2008/0294 [283], XSH/TC1-2008/0295 [302], and XSH/TC1-2008/0296 [304] are applied.

## 1.74. `isprint`, `isprint_l` — test for a printable character

---

### SYNOPSIS

```
#include <ctype.h>

int isprint(int c);

/* [CX] Extension */
int isprint_l(int c, locale_t locale);
```

### DESCRIPTION

For `isprint()` :

[The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.]

The `isprint()` and `isprint_l()` functions shall test whether `c` is a character of class **print** in the current locale, or in the locale represented by `locale`, respectively; see XBD [7. Locale].

The `c` argument is an `int`, the value of which the application shall ensure is a character representable as an `unsigned char` or equal to the value of the macro `EOF`. If the argument has any other value, the behavior is undefined.

[The behavior is undefined if the `locale` argument to `isprint_l()` is the special locale object `LC_GLOBAL_LOCALE` or is not a valid locale object handle.]

### RETURN VALUE

The `isprint()` and `isprint_l()` functions shall return non-zero if `c` is a printable character; otherwise, they shall return 0.

### ERRORS

No errors are defined.

## APPLICATION USAGE

To ensure applications portability, especially across natural languages, only these functions and the functions in the reference pages listed in the SEE ALSO section should be used for character classification.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [isalnum\(\)](#)
- [isalpha\(\)](#)
- [isblank\(\)](#)
- [iscntrl\(\)](#)
- [isdigit\(\)](#)
- [isgraph\(\)](#)
- [islower\(\)](#)
- [ispunct\(\)](#)
- [isspace\(\)](#)
- [isupper\(\)](#)
- [isxdigit\(\)](#)
- [setlocale\(\)](#)
- [uselocale\(\)](#)

XBD [7. Locale], [<ctype.h>](#), [<locale.h>](#)

## CHANGE HISTORY

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

## Issue 6

The normative text is updated to avoid use of the term "must" for application requirements.

## Issue 7

The `isprint_l()` function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0297 [302], XSH/TC1-2008/0298 [283], and XSH/TC1-2008/0299 [283] are applied.

---

## 1.75. `ispunct`, `ispunct_l` — test for a punctuation character

---

### SYNOPSIS

```
#include <ctype.h>

int ispunct(int c);

[CX] int ispunct_l(int c, locale_t locale);
```

### DESCRIPTION

For `ispunct()`: [CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `ispunct()` [CX] and `ispunct_l()` functions shall test whether `c` is a character of class **punct** in the current locale, [CX] or in the locale represented by `locale`, respectively; see XBD [7. Locale](#).

The `c` argument is an `int`, the value of which the application shall ensure is a character representable as an `unsigned char` or equal to the value of the macro `EOF`. If the argument has any other value, the behavior is undefined.

[CX] The behavior is undefined if the `locale` argument to `ispunct_l()` is the special locale object `LC_GLOBAL_LOCALE` or is not a valid locale object handle.

### RETURN VALUE

The `ispunct()` [CX] and `ispunct_l()` functions shall return non-zero if `c` is a punctuation character; otherwise, they shall return 0.

### ERRORS

No errors are defined.

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

To ensure applications portability, especially across natural languages, only these functions and the functions in the reference pages listed in the SEE ALSO section should be used for character classification.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [isalnum\(\)](#)
- [isalpha\(\)](#)
- [isblank\(\)](#)
- [iscntrl\(\)](#)
- [isdigit\(\)](#)
- [isgraph\(\)](#)
- [islower\(\)](#)
- [isprint\(\)](#)
- [isspace\(\)](#)
- [isupper\(\)](#)
- [isxdigit\(\)](#)
- [setlocale\(\)](#)

- `uselocale()`

XBD 7. Locale, `<ctype.h>`, `<locale.h>`

## CHANGE HISTORY

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

### Issue 6

The normative text is updated to avoid use of the term "must" for application requirements.

### Issue 7

The `ispunct_l()` function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0300 [302], XSH/TC1-2008/0301 [283], and XSH/TC1-2008/0302 [283] are applied.

## 1.76. isspace, isspace\_l — test for a white-space character

---

### SYNOPSIS

```
#include <ctype.h>

int isspace(int c);

[CX] int isspace_l(int c, locale_t locale);
```

### DESCRIPTION

For `isspace()`: [CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `isspace()` [CX] and `isspace_l()` functions shall test whether `c` is a character of class **space** in the current locale, [CX] or in the locale represented by `locale`, respectively; see XBD [7. Locale](#).

The `c` argument is an `int`, the value of which the application shall ensure is a character representable as an `unsigned char` or equal to the value of the macro `EOF`. If the argument has any other value, the behavior is undefined.

[CX] The behavior is undefined if the `locale` argument to `isspace_l()` is the special locale object `LC_GLOBAL_LOCALE` or is not a valid locale object handle.

### RETURN VALUE

The `isspace()` [CX] and `isspace_l()` functions shall return non-zero if `c` is a white-space character; otherwise, they shall return 0.

### ERRORS

No errors are defined.

## EXAMPLES

None.

## APPLICATION USAGE

To ensure applications portability, especially across natural languages, only these functions and the functions in the reference pages listed in the SEE ALSO section should be used for character classification.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

`isalnum()` , `isalpha()` , `iscntrl()` , `isdigit()` , `isgraph()` ,  
`islower()` , `isprint()` , `ispunct()` , `isupper()` , `isxdigit()` ,  
`setlocale()` , `uselocale()`

XBD 7. Locale, `<ctype.h>` , `<locale.h>`

## CHANGE HISTORY

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

### Issue 6

The normative text is updated to avoid use of the term "must" for application requirements.

### Issue 7

The `isspace_l()` function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0303 [302], XSH/TC1-2008/0304 [283], and XSH/TC1-2008/0305 [283] are applied.

---

## 1.77. isupper, isupper\_l — test for an uppercase letter

---

### SYNOPSIS

```
#include <ctype.h>

int isupper(int c);

[CX] int isupper_l(int c, locale_t locale);
```

### DESCRIPTION

For `isupper()` : [CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `isupper()` [CX] and `isupper_l()` functions shall test whether `c` is a character of class **upper** in the current locale, [CX] or in the locale represented by `locale`, respectively; see XBD [7. Locale](#).

The `c` argument is an `int`, the value of which the application shall ensure is a character representable as an `unsigned char` or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined.

[CX] The behavior is undefined if the `locale` argument to `isupper_l()` is the special locale object LC\_GLOBAL\_LOCALE or is not a valid locale object handle.

### RETURN VALUE

The `isupper()` [CX] and `isupper_l()` functions shall return non-zero if `c` is an uppercase letter; otherwise, they shall return 0.

### ERRORS

No errors are defined.

### EXAMPLES

None.

## APPLICATION USAGE

To ensure applications portability, especially across natural languages, only these functions and the functions in the reference pages listed in the SEE ALSO section should be used for character classification.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

`isalnum()` , `isalpha()` , `isblank()` , `iscntrl()` , `isdigit()` ,  
`isgraph()` , `islower()` , `isprint()` , `ispunct()` , `isspace()` ,  
`isxdigit()` , `setlocale()` , `uselocale()`

XBD 7. Locale, `<ctype.h>` , `<locale.h>`

## CHANGE HISTORY

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

### Issue 6

The normative text is updated to avoid use of the term "must" for application requirements.

### Issue 7

The `isupper_l()` function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0306 [302], XSH/TC1-2008/0307 [283], and XSH/TC1-2008/0308 [283] are applied.

---

## 1.78. `isxdigit`, `isxdigit_l` — test for a hexadecimal digit

### SYNOPSIS

```
#include <ctype.h>

int isxdigit(int c);

[CX] int isxdigit_l(int c, locale_t locale);
```

### DESCRIPTION

For `isxdigit()` :

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `isxdigit()` [CX] and `isxdigit_l()` functions shall test whether `c` is a character of class **xdigit** in the current locale, [CX] or in the locale represented by `locale`, respectively; see XBD [7. Locale](#).

The `c` argument is an `int`, the value of which the application shall ensure is a character representable as an `unsigned char` or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined.

[CX] The behavior is undefined if the `locale` argument to `isxdigit_l()` is the special locale object LC\_GLOBAL\_LOCALE or is not a valid locale object handle.

### RETURN VALUE

The `isxdigit()` [CX] and `isxdigit_l()` functions shall return non-zero if `c` is a hexadecimal digit; otherwise, they shall return 0.

### ERRORS

No errors are defined.

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

To ensure applications portability, especially across natural languages, only these functions and the functions in the reference pages listed in the SEE ALSO section should be used for character classification.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

`isalnum()` , `isalpha()` , `isblank()` , `iscntrl()` , `isdigit()` ,  
`isgraph()` , `islower()` , `isprint()` , `ispunct()` , `isspace()` ,  
`isupper()`

XBD 7. Locale, `<ctype.h>`

## CHANGE HISTORY

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

### Issue 6

The normative text is updated to avoid use of the term "must" for application requirements.

### Issue 7

The `isxdigit_l()` function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0347 [302], XSH/TC1-2008/0348 [283], and XSH/TC1-2008/0349 [283] are applied.

## 1.79. kill

---

### SYNOPSIS

```
#include <signal.h>

int kill(pid_t pid, int sig);
```

### DESCRIPTION

The `kill()` function shall send a signal to a process or a group of processes specified by `pid`. The signal to be sent is specified by `sig` and is either one from the list given in `<signal.h>` or 0. If `sig` is 0 (the null signal), error checking is performed but no signal is actually sent. The null signal can be used to check the validity of `pid`.

For a process to have permission to send a signal to a process designated by `pid`, unless the sending process has appropriate privileges, the real or effective user ID of the sending process shall match the real or saved set-user-ID of the receiving process.

If `pid` is greater than 0, `sig` shall be sent to the process whose process ID is equal to `pid`.

If `pid` is 0, `sig` shall be sent to all processes (excluding an unspecified set of system processes) whose process group ID is equal to the process group ID of the sender, and for which the process has permission to send a signal.

If `pid` is -1, `sig` shall be sent to all processes (excluding an unspecified set of system processes) for which the process has permission to send that signal.

If `pid` is negative, but not -1, `sig` shall be sent to all processes (excluding an unspecified set of system processes) whose process group ID is equal to the absolute value of `pid`, and for which the process has permission to send a signal.

If the value of `pid` causes `sig` to be generated for the sending process, and if `sig` is not blocked for the calling thread and if no other thread has `sig` unblocked or is waiting in a `sigwait()` function for `sig`, either `sig` or at least one pending unblocked signal shall be delivered to the sending thread before `kill()` returns.

The user ID tests described above shall not be applied when sending SIGCONT to a process that is a member of the same session as the sending process.

An implementation that provides extended security controls may impose further implementation-defined restrictions on the sending of signals, including the null signal. In particular, the system may deny the existence of some or all of the processes specified by `pid`.

The `kill()` function is successful if the process has permission to send `sig` to any of the processes specified by `pid`. If `kill()` fails, no signal shall be sent.

## RETURN VALUE

Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and `errno` set to indicate the error.

## ERRORS

The `kill()` function shall fail if:

- **[EINVAL]**  
The value of the `sig` argument is an invalid or unsupported signal number.
  - **[EPERM]**  
The process does not have permission to send the signal to any receiving process.
  - **[ESRCH]**  
No process or process group can be found corresponding to that specified by `pid`.
- 

## APPLICATION USAGE

None.

## RATIONALE

The semantics for permission checking for `kill()` differed between System V and most other implementations, such as Version 7 or 4.3 BSD. The semantics chosen for this volume of POSIX.1-2024 agree with System V. Specifically, a set-user-ID process cannot protect itself against signals (or at least not against SIGKILL) unless it changes its real user ID. This choice allows the user who starts an application to send it signals even if it changes its effective user ID. The other

semantics give more power to an application that wants to protect itself from the user who ran it.

Some implementations provide semantic extensions to the `kill()` function when the absolute value of `pid` is greater than some maximum, or otherwise special, value. Negative values are a flag to `kill()`. Since most implementations return [ESRCH] in this case, this behavior is not included in this volume of POSIX.1-2024, although a conforming implementation could provide such an extension.

The unspecified processes to which a signal cannot be sent may include the scheduler or `init`.

There was initially strong sentiment to specify that, if `pid` specifies that a signal be sent to the calling process and that signal is not blocked, that signal would be delivered before `kill()` returns. This would permit a process to call `kill()` and be guaranteed that the call never return. However, historical implementations that provide only the `signal()` function make only the weaker guarantee in this volume of POSIX.1-2024, because they only deliver one signal each time a process enters the kernel. Modifications to such implementations to support the `sigaction()` function generally require entry to the kernel following return from a signal-catching function, in order to restore the signal mask. Such modifications have the effect of satisfying the stronger requirement, at least when `sigaction()` is used, but not necessarily when `signal()` is used. The standard developers considered making the stronger requirement except when `signal()` is used, but felt this would be unnecessarily complex. Implementors are encouraged to meet the stronger requirement whenever possible. In practice, the weaker requirement is the same, except in the rare case when two signals arrive during a very short window. This reasoning also applies to a similar requirement for `sigprocmask()`.

In 4.2 BSD, the SIGCONT signal can be sent to any descendant process regardless of user-ID security checks. This allows a job control shell to continue a job even if processes in the job have altered their user IDs (as in the `su` command). In keeping with the addition of the concept of sessions, similar functionality is provided by allowing the SIGCONT signal to be sent to any process in the same session regardless of user ID security checks. This is less restrictive than BSD in the sense that ancestor processes (in the same session) can now be the recipient. It is more restrictive than BSD in the sense that descendant processes that form new sessions are now subject to the user ID checks. A similar relaxation of security is not necessary for the other job control signals since those signals are typically sent by the terminal driver in recognition of special characters being typed; the terminal driver bypasses all security checks.

In secure implementations, a process may be restricted from sending a signal to a process having a different security label. In order to prevent the existence or nonexistence of a process from being used as a covert channel, such processes

should appear nonexistent to the sender; that is, [ESRCH] should be returned, rather than [EPERM], if `pid` refers only to such processes.

Historical implementations varied on the result of a `kill()` with `pid` indicating a zombie process. Some indicated success on such a call (subject to permission checking), while others gave an error of [ESRCH]. Since the definition of process lifetime in this volume of POSIX.1-2024 covers zombie processes, the [ESRCH] error as described is inappropriate in this case and implementations that give this error do not conform. This means that an application cannot have a parent process check for termination of a particular child by sending it the null signal with `kill()`, but must instead use `waitpid()` or `waitid()`.

There is some belief that the name `kill()` is misleading, since the function is not always intended to cause process termination. However, the name is common to all historical implementations, and any change would be in conflict with the goal of minimal changes to existing application code.

## FUTURE DIRECTIONS

None.

## SEE ALSO

`getpid()` , `raise()` , `setsid()` , `sig2str()` , `sigaction()` ,  
`sigqueue()` , `wait()`

XBD `<signal.h>` , `<sys/types.h>`

## CHANGE HISTORY

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

### Issue 5

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

### Issue 6

In the SYNOPSIS, the optional include of the `<sys/types.h>` header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, the second paragraph is reworded to indicate that the saved set-user-ID of the calling process is checked in place of its effective user ID. This is a FIPS requirement.
- The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- The behavior when `pid` is -1 is now specified. It was previously explicitly unspecified in the POSIX.1-1988 standard.

The normative text is updated to avoid use of the term "must" for application requirements.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/51 is applied, correcting the RATIONALE section.

## Issue 7

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0194 [765] is applied.

## Issue 8

Austin Group Defect 1138 is applied, adding `sig2str()` to the SEE ALSO section.

---

## 1.80. labs, llabs — return a long integer absolute value

---

### SYNOPSIS

```
#include <stdlib.h>

long labs(long i);
long long llabs(long long i);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `labs()` function shall compute the absolute value of the **long** integer operand `i`. The `llabs()` function shall compute the absolute value of the **long long** integer operand `i`. If the result cannot be represented, the behavior is undefined.

### RETURN VALUE

The `labs()` function shall return the absolute value of the **long** integer operand.

The `llabs()` function shall return the absolute value of the **long long** integer operand.

### ERRORS

No errors are defined.

---

*The following sections are informative.*

### EXAMPLES

None.

## APPLICATION USAGE

Since POSIX.1 requires a two's complement representation of `long` and `long long`, the absolute value of the negative integers with the largest magnitude `{LONG_MIN}` and `{LLONG_MIN}` are not representable, thus `labs(LONG_MIN)` and `llabs(LLONG_MIN)` are undefined.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `abs()`
- XBD `<stdlib.h>`

## CHANGE HISTORY

First released in Issue 4. Derived from the ISO C standard.

### Issue 6

The `llabs()` function is added for alignment with the ISO/IEC 9899:1999 standard.

### Issue 7

SD5-XSH-ERN-152 is applied, correcting the RETURN VALUE section.

### Issue 8

Austin Group Defect 1108 is applied, changing the APPLICATION USAGE section.

## 1.81. **ldiv**, **lldiv** — compute quotient and remainder of a long division

---

### SYNOPSIS

```
#include <stdlib.h>

ldiv_t ldiv(long numer, long denom);
lldiv_t lldiv(long long numer, long long denom);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

These functions shall compute the quotient and remainder of the division of the numerator `numer` by the denominator `denom`. If the division is inexact, the resulting quotient is the `long` integer (for the `ldiv()` function) or `long long` integer (for the `lldiv()` function) of lesser magnitude that is the nearest to the algebraic quotient. If the result cannot be represented, the behavior is undefined; otherwise, `quot * denom + rem` shall equal `numer`.

### RETURN VALUE

The `ldiv()` function shall return a structure of type `ldiv_t`, comprising both the quotient and the remainder. The structure shall include the following members, in any order:

```
long quot; /* Quotient */
long rem; /* Remainder */
```

The `lldiv()` function shall return a structure of type `lldiv_t`, comprising both the quotient and the remainder. The structure shall include the following members, in any order:

```
long long quot; /* Quotient */
long long rem; /* Remainder */
```

# ERRORS

No errors are defined.

## SEE ALSO

- `div()`
- `<stdlib.h>`

## CHANGE HISTORY

First released in Issue 4. Derived from the ISO C standard.

### Issue 6

The `lldiv()` function is added for alignment with the ISO/IEC 9899:1999 standard.

---

## 1.82. llabs

---

### SYNOPSIS

```
#include <stdlib.h>

long labs(long i);
long long llabs(long long i);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `labs()` function shall compute the absolute value of the **long** integer operand `i`. The `llabs()` function shall compute the absolute value of the **long long** integer operand `i`. If the result cannot be represented, the behavior is undefined.

### RETURN VALUE

The `labs()` function shall return the absolute value of the **long** integer operand.

The `llabs()` function shall return the absolute value of the **long long** integer operand.

### ERRORS

No errors are defined.

---

*The following sections are informative.*

### EXAMPLES

None.

## APPLICATION USAGE

Since POSIX.1 requires a two's complement representation of `long` and `long long`, the absolute value of the negative integers with the largest magnitude `{LONG_MIN}` and `{LLONG_MIN}` are not representable, thus `labs(LONG_MIN)` and `llabs(LLONG_MIN)` are undefined.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [abs\(\)](#)
- XBD

## CHANGE HISTORY

First released in Issue 4. Derived from the ISO C standard.

### Issue 6

The `llabs()` function is added for alignment with the ISO/IEC 9899:1999 standard.

### Issue 7

SD5-XSH-ERN-152 is applied, correcting the RETURN VALUE section.

### Issue 8

Austin Group Defect 1108 is applied, changing the APPLICATION USAGE section.

## 1.83. lldiv

---

### SYNOPSIS

```
#include <stdlib.h>

ldiv_t ldiv(long numer, long denom);
lldiv_t lldiv(long long numer, long long denom);
```

### DESCRIPTION

These functions shall compute the quotient and remainder of the division of the numerator `numer` by the denominator `denom`. If the division is inexact, the resulting quotient is the `long` integer (for the `ldiv()` function) or `long long` integer (for the `lldiv()` function) of lesser magnitude that is the nearest to the algebraic quotient. If the result cannot be represented, the behavior is undefined; otherwise, `quot` \* `denom` + `rem` shall equal `numer`.

### RETURN VALUE

The `ldiv()` function shall return a structure of type `ldiv_t`, comprising both the quotient and the remainder. The structure shall include the following members, in any order:

```
long quot; /* Quotient */
long rem; /* Remainder */
```

The `lldiv()` function shall return a structure of type `lldiv_t`, comprising both the quotient and the remainder. The structure shall include the following members, in any order:

```
long long quot; /* Quotient */
long long rem; /* Remainder */
```

### ERRORS

No errors are defined.

---

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `div()`

XBD `<stdlib.h>`

## CHANGE HISTORY

First released in Issue 4. Derived from the ISO C standard.

### Issue 6

The `lldiv()` function is added for alignment with the ISO/IEC 9899:1999 standard.

---

## 1.84. `localeconv` — return locale-specific information

---

### SYNOPSIS

```
#include <locale.h>

struct lconv *localeconv(void);
```

### DESCRIPTION

The `localeconv()` function shall set the components of an object with the type `struct lconv` with the values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale.

The members of the structure with type `char *` are pointers to strings, any of which (except `decimal_point`) can point to "", to indicate that the value is not available in the current locale or is of zero length. The members with type `char` are non-negative numbers, any of which can be `{CHAR_MAX}` to indicate that the value is not available in the current locale.

#### Structure Members

##### `char *decimal_point`

- The radix character used to format non-monetary quantities.

##### `char *thousands_sep`

- The character used to separate groups of digits before the decimal-point character in formatted non-monetary quantities.

##### `char *grouping`

- A string whose elements taken as one-byte integer values indicate the size of each group of digits in formatted non-monetary quantities.

##### `char *int_curr_symbol`

- The international currency symbol applicable to the current locale. The first three characters contain the alphabetic international currency symbol in accordance with those specified in the ISO 4217:2015 standard. The fourth character (immediately preceding the null byte) is the character used to separate the international currency symbol from the monetary quantity.

##### `char *currency_symbol`

- The local currency symbol applicable to the current locale.

#### **char \*mon\_decimal\_point**

- The radix character used to format monetary quantities.

#### **char \*mon\_thousands\_sep**

- The separator for groups of digits before the decimal-point in formatted monetary quantities.

#### **char \*mon\_grouping**

- A string whose elements taken as one-byte integer values indicate the size of each group of digits in formatted monetary quantities.

#### **char \*positive\_sign**

- The string used to indicate a non-negative valued formatted monetary quantity.

#### **char \*negative\_sign**

- The string used to indicate a negative valued formatted monetary quantity.

#### **char int\_frac\_digits**

- The number of fractional digits (those after the decimal-point) to be displayed in an internationally formatted monetary quantity.

#### **char frac\_digits**

- The number of fractional digits (those after the decimal-point) to be displayed in a formatted monetary quantity.

#### **char p\_cs\_precedes**

- Set to 1 if the `currency_symbol` precedes the value for a non-negative formatted monetary quantity. Set to 0 if the symbol succeeds the value.

#### **char p\_sep\_by\_space**

- Set to a value indicating the separation of the `currency_symbol`, the sign string, and the value for a non-negative formatted monetary quantity.

#### **char n\_cs\_precedes**

- Set to 1 if the `currency_symbol` precedes the value for a negative formatted monetary quantity. Set to 0 if the symbol succeeds the value.

#### **char n\_sep\_by\_space**

- Set to a value indicating the separation of the `currency_symbol`, the sign string, and the value for a negative formatted monetary quantity.

### char p\_sign\_posn

- Set to a value indicating the positioning of the `positive_sign` for a non-negative formatted monetary quantity.

### char n\_sign\_posn

- Set to a value indicating the positioning of the `negative_sign` for a negative formatted monetary quantity.

### char int\_p\_cs\_precedes

- Set to 1 or 0 if the `int_curr_symbol` respectively precedes or succeeds the value for a non-negative internationally formatted monetary quantity.

### char int\_n\_cs\_precedes

- Set to 1 or 0 if the `int_curr_symbol` respectively precedes or succeeds the value for a negative internationally formatted monetary quantity.

### char int\_p\_sep\_by\_space

- Set to a value indicating the separation of the `int_curr_symbol`, the sign string, and the value for a non-negative internationally formatted monetary quantity.

### char int\_n\_sep\_by\_space

- Set to a value indicating the separation of the `int_curr_symbol`, the sign string, and the value for a negative internationally formatted monetary quantity.

### char int\_p\_sign\_posn

- Set to a value indicating the positioning of the `positive_sign` for a non-negative internationally formatted monetary quantity.

### char int\_n\_sign\_posn

- Set to a value indicating the positioning of the `negative_sign` for a negative internationally formatted monetary quantity.

## Grouping Interpretation

The elements of `grouping` and `mon_grouping` are interpreted according to the following:

- **{CHAR\_MAX}**: No further grouping is to be performed.
- **0**: The previous element is to be repeatedly used for the remainder of the digits.

- **other**: The integer value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits before the current group.

## Space Separation Values

The values of `p_sep_by_space`, `n_sep_by_space`, `int_p_sep_by_space`, and `int_n_sep_by_space` are interpreted according to the following:

- **0**: No space separates the currency symbol and value.
- **1**: If the currency symbol and sign string are adjacent, a space separates them from the value; otherwise, a space separates the currency symbol from the value.
- **2**: If the currency symbol and sign string are adjacent, a space separates them; otherwise, a space separates the sign string from the value.

For `int_p_sep_by_space` and `int_n_sep_by_space`, the fourth character of `int_curr_symbol` is used instead of a space.

## Sign Positioning Values

The values of `p_sign_posn`, `n_sign_posn`, `int_p_sign_posn`, and `int_n_sign_posn` are interpreted according to the following:

- **0**: Parentheses surround the quantity and `currency_symbol` or `int_curr_symbol`.
- **1**: The sign string precedes the quantity and `currency_symbol` or `int_curr_symbol`.
- **2**: The sign string succeeds the quantity and `currency_symbol` or `int_curr_symbol`.
- **3**: The sign string immediately precedes the `currency_symbol` or `int_curr_symbol`.
- **4**: The sign string immediately succeeds the `currency_symbol` or `int_curr_symbol`.

The implementation shall behave as if no function in this volume of POSIX.1-2024 calls `localeconv()`.

The `localeconv()` function need not be thread-safe; however, `localeconv()` shall avoid data races with all other functions.

## RETURN VALUE

The `localeconv()` function shall return a pointer to the filled-in object. The application shall not modify the structure to which the return value points, nor any storage areas pointed to by pointers within the structure. The returned pointer, and pointers within the structure, might be invalidated or the structure or the storage areas might be overwritten by a subsequent call to `localeconv()`. In addition, the returned pointer, and pointers within the structure, might be invalidated or the structure or the storage areas might be overwritten by subsequent calls to `setlocale()` with the categories `LC_ALL`, `LC_MONETARY`, or `LC_NUMERIC`, or by calls to `use locale()` which change the categories `LC_MONETARY` or `LC_NUMERIC`. The returned pointer, pointers within the structure, the structure, and the storage areas might also be invalidated if the calling thread is terminated.

## ERRORS

No errors are defined.

## EXAMPLES

None.

## APPLICATION USAGE

The following table illustrates the rules which may be used by four countries to format monetary quantities:

Country	Positive Format	Negative Format	International Format
Italy	€.1.230	-€.1.230	EUR.1.230
Netherlands	€ 1.234,56	€ -1.234,56	EUR 1.234,56
Norway	kr1.234,56	kr1.234,56-	NOK 1.234,56
Switzerland	SFr.1,234.56	SFr.1,234.56C	CHF 1,234.56

For these four countries, the respective values for the monetary members of the structure returned by `localeconv()` are:

Member	Italy	Netherlands	Norway	Switzerland
<b>int_curr_symbol</b>	"EUR."	"EUR "	"NOK "	"CHF "
<b>currency_symbol</b>	"€."	"€"	"kr"	"SFrs."
<b>mon_decimal_point</b>	"."	"."	"."	"."
<b>mon_thousands_sep</b>	"."	"."	"."	"."
<b>mon_grouping</b>	"\3"	"\3"	"\3"	"\3"
<b>positive_sign</b>	"+"	"+"	"+"	"+"
<b>negative_sign</b>	"_"	"_"	"_"	"C"
<b>int_frac_digits</b>	0	2	2	2
<b>frac_digits</b>	0	2	2	2
<b>p_cs_precedes</b>	1	1	1	1
<b>p_sep_by_space</b>	0	1	0	0
<b>n_cs_precedes</b>	1	1	1	1
<b>n_sep_by_space</b>	0	1	0	0
<b>p_sign_posn</b>	1	1	1	1
<b>n_sign_posn</b>	1	4	2	2
<b>int_p_cs_precedes</b>	1	1	1	1
<b>int_n_cs_precedes</b>	1	1	1	1
<b>int_p_sep_by_space</b>	0	0	0	0
<b>int_n_sep_by_space</b>	0	0	0	0
<b>int_p_sign_posn</b>	1	1	1	1
<b>int_n_sign_posn</b>	1	4	4	2

# RATIONALE

None.

# FUTURE DIRECTIONS

None.

# SEE ALSO

- `fprintf()`
- `fscanf()`
- `isalpha()`
- `nl_langinfo()`
- `setlocale()`
- `strcat()`
- `strchr()`
- `strcmp()`
- `strcoll()`
- `strcpy()`
- `strftime()`
- `strlen()`
- `strpbrk()`
- `strspn()`
- `strtok()`
- `strxfrm()`
- `strtod()`
- `uselocale()`
- XBD `<langinfo.h>` , `<locale.h>`

# CHANGE HISTORY

First released in Issue 4. Derived from the ANSI C standard.

## Issue 6

- A note indicating that this function need not be reentrant is added to the DESCRIPTION.
- The RETURN VALUE section is rewritten to avoid use of the term "must".
- This reference page is updated for alignment with the ISO/IEC 9899:1999 standard.
- ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.
- IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/31 is applied, removing references to `int_curr_symbol` and updating the descriptions of `p_sep_by_space` and `n_sep_by_space`. These changes are for alignment with the ISO C standard.

## Issue 7

- Austin Group Interpretation 1003.1-2001 #156 is applied.
- The definitions of `int_curr_symbol` and `currency_symbol` are updated.
- The examples in the APPLICATION USAGE section are updated.
- POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0362 [75] is applied.
- POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0200 [656] is applied.

## Issue 8

- Austin Group Defect 1302 is applied, aligning this function with the ISO/IEC 9899:2018 standard.

## 1.85. `localtime`, `localtime_r` — convert a time value to a broken-down local time

### SYNOPSIS

```
#include <time.h>

struct tm *localtime(const time_t *timer);

[CX] struct tm *localtime_r(const time_t *restrict timer,
                           struct tm *restrict result);
```

### DESCRIPTION

For `localtime()`:

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `localtime()` function shall convert the time in seconds since the Epoch pointed to by `timer` into a broken-down time, expressed as a local time. The function corrects for the timezone and any seasonal time adjustments. [CX] Local timezone information shall be set as though `localtime()` calls `tzset()`.

The relationship between a time in seconds since the Epoch used as an argument to `localtime()` and the `tm` structure (defined in the `<time.h>` header) is that the result shall be as specified in the expression given in the definition of seconds since the Epoch (see XBD 4.19 Seconds Since the Epoch) corrected for timezone and any seasonal time adjustments, where the names in the structure and in the expression correspond.

The same relationship shall apply for `localtime_r()`.

The `localtime()` function need not be thread-safe; however, `localtime()` shall avoid data races with all functions other than itself, `asctime()`, `ctime()`, and `gmtime()`.

The `asctime()`, `ctime()`, `gmtime()`, and `localtime()` functions shall return values in one of two static objects: a broken-down time structure and an array of type `char`. Execution of any of the functions that return a pointer to one of these object types may overwrite the information in any object of the same type pointed to by the value returned from any previous call to any of them.

[CX] The `localtime_r()` function shall convert the time in seconds since the Epoch pointed to by `timer` into a broken-down time stored in the structure to which `result` points. The `localtime_r()` function shall also return a pointer to that same structure.

Unlike `localtime()`, the `localtime_r()` function is not required to set `tzname`. If `localtime_r()` sets `tzname`, it shall also set `daylight` and `timezone`. If `localtime_r()` does not set `tzname`, it shall not set `daylight` and shall not set `timezone`. If the `tm` structure member `tm_zone` is accessed after the value of `TZ` is subsequently modified, the behaviour is undefined.

## RETURN VALUE

Upon successful completion, the `localtime()` function shall return a pointer to the broken-down time structure. If an error is detected, `localtime()` shall return a null pointer [CX] and set `errno` to indicate the error.

Upon successful completion, `localtime_r()` shall return a pointer to the structure pointed to by the argument `result`. If an error is detected, `localtime_r()` shall return a null pointer and set `errno` to indicate the error.

## ERRORS

The `localtime()` [CX] and `localtime_r()` functions shall fail if:

- **[OVERFLOW]:** [CX] The result cannot be represented.

## EXAMPLES

### Getting the Local Date and Time

The following example uses the `time()` function to calculate the time elapsed, in seconds, since January 1, 1970 0:00 UTC (the Epoch), `localtime()` to convert that value to a broken-down time, and `asctime()` to convert the broken-down time values into a printable string.

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

int main(void)
{
    time_t result;
```

```
    result = time(NULL);
    printf("%s%ju secs since the Epoch\n",
           asctime(localtime(&result)),
           (uintmax_t)result);
    return(0);
}
```

This example writes the current time to `stdout` in a form like this:

```
Wed Jun 26 10:32:15 1996
835810335 secs since the Epoch
```

## Getting the Modification Time for a File

The following example prints the last data modification timestamp in the local timezone for a given file.

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

int
print_file_time(const char *pathname)
{
    struct stat statbuf;
    struct tm *tm;
    char timestr[BUFSIZ];

    if(stat(pathname, &statbuf) == -1)
        return -1;
    if((tm = localtime(&statbuf.st_mtime)) == NULL)
        return -1;
    if(strftime(timestr, sizeof(timestr), "%Y-%m-%d %H:%M:%S", tm)
        return -1;
    printf("%s: %s.%09ld\n", pathname, timestr, statbuf.st_mtim.tv_sec);
    return 0;
}
```

## Timing an Event

The following example gets the current time, converts it to a string using `localtime()` and `asctime()`, and prints it to standard output using `fputs()`. It then prints the number of minutes to an event being timed.

```
#include <time.h>
#include <stdio.h>
...
time_t now;
int minutes_to_event;
...
time(&now);
printf("The time is ");
fputs(asctime(localtime(&now)), stdout);
printf("There are still %d minutes to the event.\n",
       minutes_to_event);
...
```

## APPLICATION USAGE

The `localtime_r()` function is thread-safe and returns values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

`asctime()`, `clock()`, `ctime()`, `difftime()`, `futimens()`, `getdate()`,  
`gmtime()`, `mktime()`, `strftime()`, `strptime()`, `time()`, `tzset()`

XBD 4.19 Seconds Since the Epoch, `<time.h>`

## CHANGE HISTORY

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

## Issue 5

A note indicating that the `localtime()` function need not be reentrant is added to the DESCRIPTION.

The `localtime_r()` function is included for alignment with the POSIX Threads Extension.

## Issue 6

The `localtime_r()` function is marked as part of the Thread-Safe Functions option.

Extensions beyond the ISO C standard are marked.

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

The `restrict` keyword is added to the `localtime_r()` prototype for alignment with the ISO/IEC 9899:1999 standard.

Examples are added.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/32 is applied, adding the [EOVERFLOW] error.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/55 is applied, updating the error handling for `localtime_r()`.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/56 is applied, adding a requirement that if `localtime_r()` does not set the `tzname` variable, it shall not set the `daylight` or `timezone` variables. On systems supporting XSI, the `daylight`, `timezone`, and `tzname` variables should all be set to provide information for the same timezone. This updates the description of `localtime_r()` to mention `daylight` and `timezone` as well as `tzname`.

The SEE ALSO section is updated.

## Issue 7

Austin Group Interpretation 1003.1-2001 #156 is applied.

The `localtime_r()` function is moved from the Thread-Safe Functions option to the Base.

Changes are made to the EXAMPLES section related to support for finegrained timestamps.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0363 [291] is applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0201 [664] is applied.

## Issue 8

Austin Group Defect 1125 is applied, changing "Local timezone information is used" to "Local timezone information shall be set".

Austin Group Defect 1302 is applied, aligning the `localtime()` function with the ISO/IEC 9899:2018 standard.

Austin Group Defect 1376 is applied, removing CX shading from some text derived from the ISO C standard and updating it to match the ISO C standard.

Austin Group Defect 1533 is applied, adding `tm_gmtoff` and `tm_zone` to the `tm` structure.

Austin Group Defect 1570 is applied, removing extra spacing in "==".

---

## 1.86. localtime, localtime\_r - convert a time value to a broken-down local time

---

### SYNOPSIS

```
#include <time.h>

struct tm *localtime(const time_t *timer);

[CX] struct tm *localtime_r(const time_t *restrict timer,
                           struct tm *restrict result);
```

### DESCRIPTION

For `localtime()`:

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `localtime()` function shall convert the time in seconds since the Epoch pointed to by `timer` into a broken-down time, expressed as a local time. The function corrects for the timezone and any seasonal time adjustments. [CX] Local timezone information shall be set as though `localtime()` calls `tzset()`.

The relationship between a time in seconds since the Epoch used as an argument to `localtime()` and the `tm` structure (defined in the `<time.h>` header) is that the result shall be as specified in the expression given in the definition of seconds since the Epoch (see XBD 4.19 Seconds Since the Epoch) corrected for timezone and any seasonal time adjustments, where the names in the structure and in the expression correspond.

The same relationship shall apply for `localtime_r()`.

The `localtime()` function need not be thread-safe; however, `localtime()` shall avoid data races with all functions other than itself, `asctime()`, `ctime()`, and `gmtime()`.

The `asctime()`, `ctime()`, `gmtime()`, and `localtime()` functions shall return values in one of two static objects: a broken-down time structure and an array of type `char`. Execution of any of the functions that return a pointer to one of these object types may overwrite the information in any object of the same type pointed to by the value returned from any previous call to any of them.

[CX] The `localtime_r()` function shall convert the time in seconds since the Epoch pointed to by `timer` into a broken-down time stored in the structure to which `result` points. The `localtime_r()` function shall also return a pointer to that same structure.

Unlike `localtime()`, the `localtime_r()` function is not required to set `tzname`. If `localtime_r()` sets `tzname`, it shall also set `daylight` and `timezone`. If `localtime_r()` does not set `tzname`, it shall not set `daylight` and shall not set `timezone`. If the `tm` structure member `tm_zone` is accessed after the value of `TZ` is subsequently modified, the behaviour is undefined.

## RETURN VALUE

Upon successful completion, the `localtime()` function shall return a pointer to the broken-down time structure. If an error is detected, `localtime()` shall return a null pointer [CX] and set `errno` to indicate the error.

Upon successful completion, `localtime_r()` shall return a pointer to the structure pointed to by the argument `result`. If an error is detected, `localtime_r()` shall return a null pointer and set `errno` to indicate the error.

## ERRORS

The `localtime()` [CX] and `localtime_r()` functions shall fail if:

- **OVERFLOW** [CX] The result cannot be represented.

## EXAMPLES

### Getting the Local Date and Time

The following example uses the `time()` function to calculate the time elapsed, in seconds, since January 1, 1970 0:00 UTC (the Epoch), `localtime()` to convert that value to a broken-down time, and `asctime()` to convert the broken-down time values into a printable string.

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

int main(void)
{
    time_t result;
```

```
    result = time(NULL);
    printf("%s%ju secs since the Epoch\n",
           asctime(localtime(&result)),
           (uintmax_t)result);
    return(0);
}
```

This example writes the current time to `stdout` in a form like this:

```
Wed Jun 26 10:32:15 1996
835810335 secs since the Epoch
```

## Getting the Modification Time for a File

The following example prints the last data modification timestamp in the local timezone for a given file.

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

int
print_file_time(const char *pathname)
{
    struct stat statbuf;
    struct tm *tm;
    char timestr[BUFSIZ];

    if(stat(pathname, &statbuf) == -1)
        return -1;
    if((tm = localtime(&statbuf.st_mtime)) == NULL)
        return -1;
    if(strftime(timestr, sizeof(timestr), "%Y-%m-%d %H:%M:%S", tm)
        return -1;
    printf("%s: %s.%09ld\n", pathname, timestr, statbuf.st_mtim.tv_sec);
    return 0;
}
```

## Timing an Event

The following example gets the current time, converts it to a string using `localtime()` and `asctime()`, and prints it to standard output using `fputs()`. It then prints the number of minutes to an event being timed.

```
#include <time.h>
#include <stdio.h>
...
time_t now;
int minutes_to_event;
...
time(&now);
printf("The time is ");
fputs(asctime(localtime(&now)), stdout);
printf("There are still %d minutes to the event.\n",
       minutes_to_event);
...
```

## APPLICATION USAGE

The `localtime_r()` function is thread-safe and returns values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `asctime()`
- `clock()`
- `ctime()`
- `difftime()`
- `futimens()`
- `getdate()`
- `gmtime()`
- `mktime()`

- `strftime()`
- `strptime()`
- `time()`
- `tzset()`

XBD 4.19 Seconds Since the Epoch, `<time.h>`

## CHANGE HISTORY

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

### Issue 5

A note indicating that the `localtime()` function need not be reentrant is added to the DESCRIPTION.

The `localtime_r()` function is included for alignment with the POSIX Threads Extension.

### Issue 6

The `localtime_r()` function is marked as part of the Thread-Safe Functions option.

Extensions beyond the ISO C standard are marked.

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

The `restrict` keyword is added to the `localtime_r()` prototype for alignment with the ISO/IEC 9899:1999 standard.

Examples are added.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/32 is applied, adding the [EOVERFLOW] error.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/55 is applied, updating the error handling for `localtime_r()`.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/56 is applied, adding a requirement that if `localtime_r()` does not set the `tzname` variable, it shall not set the `daylight` or `timezone` variables. On systems supporting XSI, the `daylight`, `timezone`, and `tzname` variables should all be set to provide information for the same timezone. This updates the description of

`localtime_r()` to mention `daylight` and `timezone` as well as `tzname`. The SEE ALSO section is updated.

## Issue 7

Austin Group Interpretation 1003.1-2001 #156 is applied.

The `localtime_r()` function is moved from the Thread-Safe Functions option to the Base.

Changes are made to the EXAMPLES section related to support for fine-grained timestamps.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0363 [291] is applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0201 [664] is applied.

## Issue 8

Austin Group Defect 1125 is applied, changing "Local timezone information is used" to "Local timezone information shall be set".

Austin Group Defect 1302 is applied, aligning the `localtime()` function with the ISO/IEC 9899:2018 standard.

Austin Group Defect 1376 is applied, removing CX shading from some text derived from the ISO C standard and updating it to match the ISO C standard.

Austin Group Defect 1533 is applied, adding `tm_gmtoff` and `tm_zone` to the `tm` structure.

Austin Group Defect 1570 is applied, removing extra spacing in "==".

## 1.87. longjmp — non-local goto

---

### SYNOPSIS

```
#include <setjmp.h>

_Noreturn void longjmp(jmp_buf env, int val);
```

### DESCRIPTION

The `longjmp()` function shall restore the environment saved by the most recent invocation of `setjmp()` in the same process, with the corresponding `jmp_buf` argument. If the most recent invocation of `setjmp()` with the corresponding `jmp_buf` occurred in another thread, or if there is no such invocation, or if the function containing the invocation of `setjmp()` has terminated execution in the interim, or if the invocation of `setjmp()` was within the scope of an identifier with variably modified type and execution has left that scope in the interim, the behavior is undefined. It is unspecified whether `longjmp()` restores the signal mask, leaves the signal mask unchanged, or restores it to its value at the time `setjmp()` was called.

All accessible objects have values, and all other components of the abstract machine have state (for example, floating-point status flags and open files), as of the time `longjmp()` was called, except that the values of objects of automatic storage duration are unspecified if they meet all the following conditions:

- They are local to the function containing the corresponding `setjmp()` invocation.
- They do not have volatile-qualified type.
- They are changed between the `setjmp()` invocation and `longjmp()` call.

Although `longjmp()` is an async-signal-safe function, if it is invoked from a signal handler which interrupted a non-async-signal-safe function or equivalent (such as the processing equivalent to `exit()` performed after a return from the initial call to `main()`), the behavior of any subsequent call to a non-async-signal-safe function or equivalent is undefined.

The effect of a call to `longjmp()` where initialization of the `jmp_buf` structure was not performed in the calling thread is undefined.

## RETURN VALUE

After `longjmp()` is completed, thread execution shall continue as if the corresponding invocation of `setjmp()` had just returned the value specified by `val`. The `longjmp()` function shall not cause `setjmp()` to return 0; if `val` is 0, `setjmp()` shall return 1.

## ERRORS

No errors are defined.

## APPLICATION USAGE

Applications whose behavior depends on the value of the signal mask should not use `longjmp()` and `setjmp()`, since their effect on the signal mask is unspecified, but should instead use the `siglongjmp()` and `sigsetjmp()` functions (which can save and restore the signal mask under application control).

It is recommended that applications do not call `longjmp()` or `siglongjmp()` from signal handlers. To avoid undefined behavior when calling these functions from a signal handler, the application needs to ensure one of the following two things:

1. After the call to `longjmp()` or `siglongjmp()` the process only calls async-signal-safe functions and does not return from the initial call to `main()`.
2. Any signal whose handler calls `longjmp()` or `siglongjmp()` is blocked during *every* call to a non-async-signal-safe function, and no such calls are made after returning from the initial call to `main()`.

## SEE ALSO

- `setjmp()`
- `sigaction()`
- `siglongjmp()`
- `sigsetjmp()`
- `<setjmp.h>`

# CHANGE HISTORY

## First released in Issue 1

Derived from Issue 1 of the SVID.

## Issue 5

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

## Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION now explicitly makes `longjmp()`'s effect on the signal mask unspecified.

The DESCRIPTION is updated for alignment with the ISO/IEC 9899:1999 standard.

## Issue 7

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0365 [394] is applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0202 [516] is applied.

## Issue 8

Austin Group Defect 1302 is applied, aligning this function with the ISO/IEC 9899:2018 standard.

---

## 1.88. malloc — a memory allocator

---

### SYNOPSIS

```
#include <stdlib.h>

void *malloc(size_t size);
```

### DESCRIPTION

The `malloc()` function shall allocate unused space for an object whose size in bytes is specified by `size` and whose value is unspecified.

The order and contiguity of storage allocated by successive calls to `malloc()` is unspecified. The pointer returned if the allocation succeeds shall be suitably aligned so that it may be assigned to a pointer to any type of object with a fundamental alignment requirement and then used to access such an object in the space allocated (until the space is explicitly freed or reallocated). Each such allocation shall yield a pointer to an object disjoint from any other object. The pointer returned points to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer shall be returned. If the size of the space requested is 0, the behavior is implementation-defined: either a null pointer shall be returned, or the behavior shall be as if the size were some non-zero value, except that the behavior is undefined if the returned pointer is used to access an object.

For purposes of determining the existence of a data race, `malloc()` shall behave as though it accessed only memory locations accessible through its argument and not other static duration storage. The function may, however, visibly modify the storage that it allocates. Calls to `aligned_alloc()`, `calloc()`, `free()`, `malloc()`, `posix_memalign()`, `reallocarray()`, and `realloc()` that allocate or deallocate a particular region of memory shall occur in a single total order (see [4.15.1 Memory Ordering](#)), and each such deallocation call shall synchronize with the next allocation (if any) in this order.

### RETURN VALUE

Upon successful completion, `malloc()` shall return a pointer to the allocated space; if `size` is 0, the application shall ensure that the pointer is not used to access an object.

Otherwise, it shall return a null pointer and set `errno` to indicate the error.

## ERRORS

The `malloc()` function shall fail if:

- **[ENOMEM]** Insufficient storage space is available.

The `malloc()` function may fail if:

- **[EINVAL]** `size` is 0 and the implementation does not support 0 sized allocations.

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

Some implementations set `errno` to [EAGAIN] to signal memory allocation failures that might succeed if retried and [ENOMEM] for failures that are unlikely to ever succeed, for example due to configured limits. [2.3 Error Numbers](#) permits this behavior; when multiple error conditions are simultaneously true there is no precedence between them.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `aligned_alloc()`
- `calloc()`
- `free()`

- `getrlimit()`
- `posix_memalign()`
- `realloc()`

XBD `<stdlib.h>`

## CHANGE HISTORY

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

### Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE section, the requirement to set `errno` to indicate an error is added.
- The [ENOMEM] error condition is added.

### Issue 7

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0203 [526] is applied.

### Issue 8

Austin Group Defect 374 is applied, changing the RETURN VALUE and ERRORS sections in relation to 0 sized allocations.

Austin Group Defect 1302 is applied, aligning this function with the ISO/IEC 9899:2018 standard.

Austin Group Defects 1387 and 1489 are applied, changing the RATIONALE section.

---

## 1.89. memchr

---

### SYNOPSIS

```
#include <string.h>

void *memchr(const void *s, int c, size_t n);
```

### DESCRIPTION

The `memchr()` function shall locate the first occurrence of `c` (converted to an `unsigned char`) in the initial `n` bytes (each interpreted as `unsigned char`) pointed to by `s`.

The implementation shall behave as if it reads the bytes sequentially and stops as soon as a matching byte is found.

The `memchr()` function shall not change the setting of `errno` on valid input.

### RETURN VALUE

The `memchr()` function shall return a pointer to the located byte, or a null pointer if the byte is not found.

### ERRORS

No errors are defined.

---

### EXAMPLES

None.

### APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

`<string.h>`

## CHANGE HISTORY

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

### Issue 7

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0374 [110] is applied.

### Issue 8

Austin Group Defect 448 is applied, adding a requirement that `memchr()` does not change the setting of `errno` on valid input.

Austin Group Defect 1302 is applied, aligning this function with the ISO/IEC 9899:2018 standard.

---

## 1.90. memcmp — compare bytes in memory

---

### SYNOPSIS

```
#include <string.h>

int memcmp(const void *s1, const void *s2, size_t n);
```

### DESCRIPTION

The `memcmp()` function shall compare the first `n` bytes (each interpreted as `unsigned char`) of the object pointed to by `s1` to the first `n` bytes of the object pointed to by `s2`.

The sign of a non-zero return value shall be determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type `unsigned char`) that differ in the objects being compared.

The `memcmp()` function shall not change the setting of `errno` on valid input.

### RETURN VALUE

The `memcmp()` function shall return an integer greater than, equal to, or less than 0, if the object pointed to by `s1` is greater than, equal to, or less than the object pointed to by `s2`, respectively.

### ERRORS

No errors are defined.

### EXAMPLES

None.

### APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `<string.h>`

## CHANGE HISTORY

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

### Issue 8

Austin Group Defect 448 is applied, adding a requirement that `memcmp()` does not change the setting of `errno` on valid input.

---

## 1.91. `memcpy` — copy bytes in memory

---

### SYNOPSIS

```
#include <string.h>

void *memcpy(void *restrict s1, const void *restrict s2, size_t n)
```



### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `memcpy()` function shall copy `n` bytes from the object pointed to by `s2` into the object pointed to by `s1`. If copying takes place between objects that overlap, the behavior is undefined.

[CX] The `memcpy()` function shall not change the setting of `errno` on valid input.

### RETURN VALUE

The `memcpy()` function shall return `s1`; no return value is reserved to indicate an error.

### ERRORS

No errors are defined.

---

*The following sections are informative.*

### EXAMPLES

None.

# APPLICATION USAGE

The `memcpy()` function does not check for the overflow of the receiving memory area.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

XBD `<string.h>`

## CHANGE HISTORY

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

### Issue 6

The `memcpy()` prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

### Issue 8

Austin Group Defect 448 is applied, adding a requirement that `memcpy()` does not change the setting of `errno` on valid input.

---

## 1.92. memmove

---

### SYNOPSIS

```
#include <string.h>

void *memmove(void *s1, const void *s2, size_t n);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `memmove()` function shall copy `n` bytes from the object pointed to by `s2` into the object pointed to by `s1`. Copying takes place as if the `n` bytes from the object pointed to by `s2` are first copied into a temporary array of `n` bytes that does not overlap the objects pointed to by `s1` and `s2`, and then the `n` bytes from the temporary array are copied into the object pointed to by `s1`.

[CX] The `memmove()` function shall not change the setting of `errno` on valid input.

### RETURN VALUE

The `memmove()` function shall return `s1`; no return value is reserved to indicate an error.

### ERRORS

No errors are defined.

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

XBD [`<string.h>`](#)

## CHANGE HISTORY

First released in Issue 4. Derived from the ANSI C standard.

### Issue 8

Austin Group Defect 448 is applied, adding a requirement that `memmove()` does not change the setting of `errno` on valid input.

---

## 1.93. `memset`

---

### SYNOPSIS

```
#include <string.h>

void *memset(void *s, int c, size_t n);
```

### DESCRIPTION

[Option Start] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard. [Option End]

The `memset()` function shall copy `c` (converted to an `unsigned char`) into each of the first `n` bytes of the object pointed to by `s`.

[Option Start] The `memset()` function shall not change the setting of `errno` on valid input. [Option End]

### RETURN VALUE

The `memset()` function shall return `s`; no return value is reserved to indicate an error.

### ERRORS

No errors are defined.

---

*The following sections are informative.*

### EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

XBD [`<string.h>`](#)

## CHANGE HISTORY

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

### Issue 8

Austin Group Defect 448 is applied, adding a requirement that `memset()` does not change the setting of `errno` on valid input.

## 1.94. `mktimes` — convert broken-down time into time since the Epoch

---

### SYNOPSIS

```
#include <time.h>

time_t mktimes(struct tm *timeptr);
```

### DESCRIPTION

The `mktimes()` function shall convert the broken-down time, expressed as local time, in some members of 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 `mktimes()` function shall make use of only the `tm_year`, `tm_mon`, `tm_mday`, `tm_hour`, `tm_min`, `tm_sec`, and `tm_isdst` members of the structure pointed to by `timeptr`; the values of these members shall not be restricted to the ranges described in `<time.h>`.

Local timezone information shall be set as though `mktimes()` called `tzset()`.

The `mktimes()` function shall calculate the time in seconds since the Epoch to be returned as if by manipulating the members of the `tm` structure according to the following steps:

1. The `tm_sec` member may, but should not, be brought into the range 0 to 60, inclusive. For each 60 seconds added to or subtracted from `tm_sec`, a decrement or increment, respectively, of 1 minute shall be saved for later application.
2. The `tm_min` member shall be brought into the range 0 to 59, inclusive, and any saved decrement or increment of minutes shall then be applied, repeating the range adjustment afterwards if necessary. For each 60 minutes added to or subtracted from `tm_min`, a decrement or increment, respectively, of 1 hour shall be saved for later application.
3. The `tm_hour` member shall be brought into the range 0 to 23, inclusive, and any saved decrement or increment of hours shall then be applied, repeating the range adjustment afterwards if necessary. For each 24 hours added to or subtracted from `tm_hour`, a decrement or increment, respectively, of 1 day shall be saved for later application.

4. The `tm_mon` member shall be brought into the range 0 to 11, inclusive. For each 12 months added to or subtracted from `tm_mon`, a decrement or increment, respectively, of 1 year shall be saved for later use.
5. The `tm_mday` member shall be brought into the range 1 to 31, inclusive, and any saved decrement or increment of days shall then be applied, repeating the range adjustment afterwards if necessary. Adjustments downwards shall be applied by subtracting the number of days (according to the Gregorian calendar) in month `tm_mon` +1 of the year obtained by adding/subtracting any saved increment/decrement of years to the value `tm_year` +1900, and then incrementing `tm_mon` by 1, repeated as necessary. Adjustments upwards shall be applied by adding the number of days in the month before month `tm_mon` +1 of the year obtained by adding/subtracting any saved increment/decrement of years to the value `tm_year` +1900, and then decrementing `tm_mon` by 1, repeated as necessary. During these adjustments, the `tm_mon` value shall be kept within the range 0 to 11, inclusive, by applying step 4 as necessary.
6. If the `tm_mday` member is greater than the number of days in month `tm_mon` +1 of the year obtained by adding/subtracting any saved increment/decrement of years to the value `tm_year` +1900, that number of days shall be subtracted from `tm_mday`, and `tm_mon` shall be incremented by 1. If this results in `tm_mon` having the value 12, step 4 shall be applied.
7. The number of seconds since the Epoch in Coordinated Universal Time shall be calculated from the range-corrected values of the relevant `tm` structure members (or the original value where a member was not range corrected) as specified in the expression given in the definition of seconds since the Epoch (see XBD 4.19 Seconds Since the Epoch), where the names other than `tm_year` and `tm_yday` in the structure and in the expression correspond, the `tm_year` value used in the expression is the `tm_year` in the structure plus/minus any saved increment/decrement of years, and the `tm_yday` value used in the expression is the day of the year from 0 to 365 inclusive, calculated from the `tm_mon` and `tm_mday` members of the `tm` structure, for that year.
8. The time since the Epoch shall be corrected for the offset of the local timezone's standard time from Coordinated Universal Time.
9. The time since the Epoch shall be further corrected (if applicable—see below) for Daylight Saving Time.

If the timezone is one that includes Daylight Saving Time (DST) adjustments, the value of `tm_isdst` in the `tm` structure controls whether or not `mktime()`

adjusts the calculated seconds since the Epoch value by the DST offset (after it has made the timezone adjustment), as follows:

- If `tm_isdst` is zero, `mkttime()` shall not further adjust the seconds since the Epoch by the DST offset.
- If `tm_isdst` is positive, `mkttime()` shall further adjust the seconds since the Epoch by the DST offset.
- If `tm_isdst` is negative, `mkttime()` shall attempt to determine whether DST is in effect for the specified time; if it determines that DST is in effect it shall produce the same result as an equivalent call with a positive `tm_isdst` value, otherwise it shall produce the same result as an equivalent call with a `tm_isdst` value of zero. If the broken-down time specifies a time that is either skipped over or repeated when a transition to or from DST occurs, it is unspecified whether `mkttime()` produces the same result as an equivalent call with a positive `tm_isdst` value or as an equivalent call with a `tm_isdst` value of zero.

If the `TZ` environment variable specifies a geographical timezone for which the implementation's timezone database includes historical or future changes to the offset from Coordinated Universal Time of the timezone's standard time, and the broken-down time corresponds to a time that was (or will be) skipped over or repeated due to the occurrence of such a change, `mkttime()` shall calculate the time since the Epoch value using either the offset in effect before the change or the offset in effect after the change.

Upon successful completion, the members of the structure shall be set to the values that would be returned by a call to `localtime()` with the calculated time since the Epoch as its argument.

## RETURN VALUE

The `mkttime()` function shall return the calculated time since the Epoch encoded as a value of type `time_t`. If the time since the Epoch cannot be represented as a `time_t` or the value to be returned in the `tm_year` member of the structure pointed to by `timeptr` cannot be represented as an `int`, the function shall return the value `(time_t)-1` and set `errno` to `[EOVERFLOW]`, and shall not change the value of the `tm_wday` component of the structure.

Since `(time_t)-1` is a valid return value for a successful call to `mkttime()`, an application wishing to check for error situations should set `tm_wday` to a value less than 0 or greater than 6 before calling `mkttime()`. On return, if `tm_wday` has not changed an error has occurred.

# ERRORS

The `mkttime()` function shall fail if:

## [EOVERFLOW]

The result cannot be represented.

# EXAMPLES

What day of the week is July 4, 2001?

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

struct tm time_str;

char daybuf[20];

int main(void)
{
    time_str.tm_year = 2001 - 1900;
    time_str.tm_mon = 7 - 1;
    time_str.tm_mday = 4;
    time_str.tm_hour = 0;
    time_str.tm_min = 0;
    time_str.tm_sec = 1;
    time_str.tm_isdst = -1;
    time_str.tm_wday = -1;
    if (mktime(&time_str) == (time_t)-1 && time_str.tm_wday == -1)
        (void)puts("-unknown-");
    else {
        (void)strftime(daybuf, sizeof(daybuf), "%A", &time_str);
        (void)puts(daybuf);
    }
    return 0;
}
```

# APPLICATION USAGE

When using `mkttime()` to add or subtract a fixed time period (one that always corresponds to a fixed number of seconds) to or from a broken-down time in the local timezone, reliable results for arbitrary `TZ` can only be assured by using `mkttime()` to convert the original broken-down time to a time since the Epoch, adding or subtracting the desired number of seconds to that value, and then calling

`localtime()` with the result. The alternative of adjusting the broken-down time before calling `mkttime()` may produce unexpected results if the original and updated times are on different sides of a geographical timezone change. On implementations that follow the recommendation of not range-correcting `tm_sec` (see step 1 in the DESCRIPTION), reliable results can also be assured by adding or subtracting the desired number of seconds to `tm_sec` (and not modifying any other members of the `tm` structure). In applications needing to be portable to non-POSIX systems where the `time_t` encoding is not a count of seconds, it is recommended that conditional compilation is used such that the adjustment is performed on the `mkttime()` return value when possible, and otherwise on the `tm_sec` member. For timezones that are known not to have geographical timezone changes, such as TZ=UTC0, adjustments using just `mkttime()` do not have this problem.

The way the `mkttime()` function interprets out-of-range `tm` structure fields might not produce the expected result when multiple adjustments are made at the same time. For example, if an application tries to go back one day first and then one year by calling `localtime()`, decrementing `tm_mday` and `tm_year`, and then calling `mkttime()` this would not produce the expected result if it was called on 2021-03-01 because `mkttime()` would see the supplied year as 2020 (a leap year) and correct Mar 0 to Feb 29, whereas the intended result was Feb 28. Such issues can be avoided by doing multiple adjustments one at a time when the order in which they are done matters.

Examples of how `mkttime()` handles some adjustments are:

- If given Feb 29 in a non-leap year it treats that as the day after Feb 28 and gives back Mar 1.
- If given Feb 0 it treats that as the day before Feb 1 and gives back Jan 31.
- If given 21:65 it treats that as 6 minutes after 21:59 and gives back 22:05.
- If given `tm_isdst` =0 for a time when DST is in effect, it gives back a positive `tm_isdst` and alters the other fields appropriately.
- If there is a DST transition where 02:00 standard time becomes 03:00 DST and `mkttime()` is given 02:30 (with negative `tm_isdst`), it treats that as either 30 minutes after 02:00 standard time or 30 minutes before 03:00 DST and gives back a zero or positive `tm_isdst`, respectively, with the `tm_hour` field altered appropriately.
- If a geographical timezone changes its UTC offset such that "old 00:00" becomes "new 00:30" and `mkttime()` is given 00:20, it treats that as either 20 minutes after "old 00:00" or 10 minutes before "new 00:30", and gives back appropriately altered **struct tm** fields.

If an application wants to check whether a given broken-down time is one that is skipped over, it can do so by seeing whether the `tm_mday`, `tm_hour`, and `tm_min` values it gets back from `mktimed()` are the same ones it fed in. Just checking `tm_hour` and `tm_min` might appear at first sight to suffice, but `tm_mday` could also change—without `tm_hour` and `tm_min` changing—if, for example, `TZ` is set to "ABC12XYZ-12" (which might be used in a torture test) or if a geographical timezone changes the offset from Coordinated Universal Time of its standard time by 24 hours.

## RATIONALE

In order to allow applications to distinguish between a successful return of `(time_t)-1` and an [EOVERFLOW] error, `mktimed()` is required not to change `tm_wday` on error. This mechanism is used rather than the convention used for other functions whereby the application sets `errno` to zero before the call and the call does not change `errno` on error because the ISO C standard does not require `mktimed()` to set `errno` on error. The next revision of the ISO C standard is expected to require that `mktimed()` does not change `tm_wday` when returning `(time_t)-1` to indicate an error, and that this return convention is used both for the case where the value to be returned by the function cannot be represented as a `time_t` and the case where the value to be returned in the `tm_year` member of the `tm` structure cannot be represented as an `int`.

The DESCRIPTION section says that `mktimed()` converts the specified broken-down time into a time since the Epoch value. The use of the indefinite article here is necessary because, when `tm_isdst` is negative and the timezone has Daylight Saving Time transitions, there is not a one-to-one correspondence between broken-down times and time since the Epoch values.

The description of how the value of `tm_isdst` affects the behavior of `mktimed()` is shaded CX because the requirements in the ISO C standard are unclear. The next revision of the ISO C standard is expected to state the requirements using wording equivalent to the wording in this standard.

Implementations are encouraged not to range-correct `tm_sec` (see step 1 in the DESCRIPTION) in order for the results of making an adjustment to `tm_sec` always to be equivalent to making the same adjustment to the value returned by `mktimed()`, even when the original and updated times are on different sides of a geographical timezone change. This provides a way for applications to do reliable fixed-period adjustment using only `mktimed()`, as described in APPLICATION USAGE.

The described method for range-correcting the `tm` structure members uses separate variables to hold adjustment values to be applied later to other members,

or (for the year adjustment) used in later calculations, because this is one way of avoiding intermediate member values that are not representable as an `int`. Implementations may use other methods; all that is required is that `tm_year` is the only member for which an [OVERFLOW] error can occur.

The described method for range-correcting `tm_mday` would, if implemented that way, be highly inefficient for very large values. The efficiency can be improved by observing that any period of 400 years always has the same number of days, so the month-by-month correction method need only be applied for a maximum of 4800 months.

## FUTURE DIRECTIONS

A future version of this standard may require that `mkttime()` does not perform the optional range correction of the `tm_sec` member of the `tm` structure described at step 1 in the DESCRIPTION.

A future version of this standard is expected to add a `timegm()` function that is similar to `mkttime()`, except that the `tm` structure pointed to by `timeptr` contains a broken-down time in Coordinated Universal Time (rather than the local timezone), where references to `localtime()` are replaced by references to `gmtime()`, and where there are no timezone offset or Daylight Saving Time adjustments. A combination of `gmtime()` and `timegm()` will be the expected way to perform arithmetic upon a `time_t` value and remain compatible with the ISO C standard (where the internal structure of a `time_t` is not specified), since attempting such manipulations using `localtime()` and `mkttime()` can lead to unexpected results.

## SEE ALSO

`asctime()`, `clock()`, `ctime()`, `difftime()`, `futimens()`, `gmtime()`,  
`localtime()`, `strftime()`, `strptime()`, `time()`, `tzset()`

XBD 4.19 Seconds Since the Epoch, `<time.h>`

## CHANGE HISTORY

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard and the ANSI C standard.

## Issue 6

Extensions beyond the ISO C standard are marked.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/58 is applied, updating the RETURN VALUE and ERRORS sections to add the optional [EOVERFLOW] error as a CX extension.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/59 is applied, adding the `tzset()` function to the SEE ALSO section.

## Issue 7

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0393 [104] is applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0228 [724] is applied.

## Issue 8

Austin Group Defect 1253 is applied, changing "Daylight Savings" to "Daylight Saving".

Austin Group Defect 1613 is applied, changing the way the **tm** structure members used by `mkttime()` are specified and clarifying that a successful call sets the members to the same values that would be returned by `localtime()`.

Austin Group Defect 1614 is applied, clarifying how `tm_isdst` is handled and the conditions under which `(time_t)-1` is returned.

Austin Group Defect 1627 is applied, clarifying how `mkttime()` calculates the time in seconds since the Epoch from the members of the **tm** structure.

## 1.95. `mlock`, `munlock` — lock or unlock a range of process address space (REALTIME)

---

### SYNOPSIS

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

int mlock(const void *addr, size_t len);
int munlock(const void *addr, size_t len);
```

### DESCRIPTION

The `mlock()` function shall cause those whole pages containing any part of the address space of the process starting at address `addr` and continuing for `len` bytes to be memory-resident until unlocked or until the process exits or `exec`s another process image. The implementation may require that `addr` be a multiple of {PAGESIZE}.

The `munlock()` function shall unlock those whole pages containing any part of the address space of the process starting at address `addr` and continuing for `len` bytes, regardless of how many times `mlock()` has been called by the process for any of the pages in the specified range. The implementation may require that `addr` be a multiple of {PAGESIZE}.

If any of the pages in the range specified to a call to `munlock()` are also mapped into the address spaces of other processes, any locks established on those pages by another process are unaffected by the call of this process to `munlock()`. If any of the pages in the range specified by a call to `munlock()` are also mapped into other portions of the address space of the calling process outside the range specified, any locks established on those pages via the other mappings are also unaffected by this call.

Upon successful return from `mlock()`, pages in the specified range shall be locked and memory-resident. Upon successful return from `munlock()`, pages in the specified range shall be unlocked with respect to the address space of the process. Memory residency of unlocked pages is unspecified.

Appropriate privileges are required to lock process memory with `mlock()`.

## RETURN VALUE

Upon successful completion, the `mlock()` and `munlock()` functions shall return a value of zero. Otherwise, no change is made to any locks in the address space of the process, and the function shall return a value of -1 and set `errno` to indicate the error.

## ERRORS

The `mlock()` and `munlock()` functions shall fail if:

- **[ENOMEM]**

Some or all of the address range specified by the `addr` and `len` arguments does not correspond to valid mapped pages in the address space of the process.

The `mlock()` function shall fail if:

- **[EAGAIN]**

Some or all of the memory identified by the operation could not be locked when the call was made.

The `mlock()` and `munlock()` functions may fail if:

- **[EINVAL]**

The `addr` argument is not a multiple of {PAGESIZE}.

The `mlock()` function may fail if:

- **[ENOMEM]**

Locking the pages mapped by the specified range would exceed an implementation-defined limit on the amount of memory that the process may lock.

- **[EPERM]**

The calling process does not have appropriate privileges to perform the requested operation.

## EXAMPLES

None.

# APPLICATION USAGE

None.

# RATIONALE

None.

# FUTURE DIRECTIONS

None.

# SEE ALSO

- `exec`
- `_exit()`
- `fork()`
- `mlockall()`
- `munmap()`
- `<sys/mman.h>`

# CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

## Issue 6

The `mlock()` and `munlock()` functions are marked as part of the Range Memory Locking option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Range Memory Locking option.

---

## 1.96. mlockall

---

### SYNOPSIS

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

int mlockall(int flags);
int munlockall(void);
```

### DESCRIPTION

The `mlockall()` function shall cause all of the pages mapped by the address space of a process to be memory-resident until unlocked or until the process exits or `exec`s another process image. The `flags` argument determines whether the pages to be locked are those currently mapped by the address space of the process, those that are mapped in the future, or both. The `flags` argument is constructed from the bitwise-inclusive OR of one or more of the following symbolic constants, defined in `<sys/mman.h>`:

- **MCL\_CURRENT**

Lock all of the pages currently mapped into the address space of the process.

- **MCL\_FUTURE**

Lock all of the pages that become mapped into the address space of the process in the future, when those mappings are established.

If `MCL_FUTURE` is specified, and the automatic locking of future mappings eventually causes the amount of locked memory to exceed the amount of available physical memory or any other implementation-defined limit, the behavior is implementation-defined. The manner in which the implementation informs the application of these situations is also implementation-defined.

The `munlockall()` function shall unlock all currently mapped pages of the address space of the process. Any pages that become mapped into the address space of the process after a call to `munlockall()` shall not be locked, unless there is an intervening call to `mlockall()` specifying `MCL_FUTURE` or a subsequent call to `mlockall()` specifying `MCL_CURRENT`. If pages mapped into the address space of the process are also mapped into the address spaces of other processes and are locked by those processes, the locks established by the other processes shall be unaffected by a call by this process to `munlockall()`.

Upon successful return from the `mlockall()` function that specifies `MCL_CURRENT`, all currently mapped pages of the address space of the process

shall be memory-resident and locked. Upon return from the `munlockall()` function, all currently mapped pages of the address space of the process shall be unlocked with respect to the address space of the process. The memory residency of unlocked pages is unspecified.

Appropriate privileges are required to lock process memory with `mlockall()`.

## RETURN VALUE

Upon successful completion, the `mlockall()` function shall return a value of zero. Otherwise, no additional memory shall be locked, and the function shall return a value of -1 and set `errno` to indicate the error. The effect of failure of `mlockall()` on previously existing locks in the address space is unspecified.

If it is supported by the implementation, the `munlockall()` function shall always return a value of zero. Otherwise, the function shall return a value of -1 and set `errno` to indicate the error.

## ERRORS

The `mlockall()` function shall fail if:

- **[EAGAIN]**

Some or all of the memory identified by the operation could not be locked when the call was made.

- **[EINVAL]**

The `flags` argument is zero, or includes unimplemented flags.

The `mlockall()` function may fail if:

- **[ENOMEM]**

Locking all of the pages currently mapped into the address space of the process would exceed an implementation-defined limit on the amount of memory that the process may lock.

- **[EPERM]**

The calling process does not have appropriate privileges to perform the requested operation.

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

`exec` , `exit()` , `fork()` , `mlock()` , `munmap()`

XBD `<sys/mman.h>`

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

### Issue 6

The `mlockall()` and `munlockall()` functions are marked as part of the Process Memory Locking option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Process Memory Locking option.

---

## 1.97. mmap

---

### SYNOPSIS

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

void *mmap(void *addr, size_t len, int prot, int flags,
           int fildes, off_t off);
```

### DESCRIPTION

The `mmap()` function shall establish a mapping between an address space of a process and a memory object.

The `mmap()` function shall be supported for the following memory objects:

- Regular files
- Anonymous memory objects
- [SHM] Shared memory objects
- [TYM] Typed memory objects

Support for any other type of file is unspecified.

The format of the call is as follows:

```
pa = mmap(addr, len, prot, flags, fildes, off);
```

The `mmap()` function shall establish a mapping between the address space of the process at an address `pa` for `len` bytes to the memory object represented by the file descriptor `fildes` at offset `off` for `len` bytes, or to an anonymous memory object of `len` bytes. The value of `pa` is an implementation-defined function of the parameter `addr` and the values of `flags`, further described below. A successful `mmap()` call shall return `pa` as its result. The address range starting at `pa` and continuing for `len` bytes shall be legitimate for the possible (not necessarily current) address space of the process. The range of bytes starting at `off` and continuing for `len` bytes shall be legitimate for the possible (not necessarily current) offsets in the memory object represented by `fildes`.

[TYM] If `fildes` represents a typed memory object opened with either the `POSIX_TYPED_MEM_ALLOCATE` flag or the `POSIX_TYPED_MEM_ALLOCATE_CONTIG` flag, the memory object to be mapped shall be that portion of the typed memory object allocated by the

implementation as specified below. In this case, if `off` is non-zero, the behavior of `mmap()` is undefined. If `fildes` refers to a valid typed memory object that is not accessible from the calling process, `mmap()` shall fail.

The mapping established by `mmap()` shall replace any previous mappings for those whole pages containing any part of the address space of the process starting at `pa` and continuing for `len` bytes.

If the size of the mapped file changes after the call to `mmap()` as a result of some other operation on the mapped file, the effect of references to portions of the mapped region that correspond to added or removed portions of the file is unspecified.

If `len` is zero, `mmap()` shall fail and no mapping shall be established.

The parameter `prot` determines whether read, write, execute, or some combination of accesses are permitted to the data being mapped. The `prot` shall be either `PROT_NONE` or the bitwise-inclusive OR of one or more of the other flags in the following table, defined in the `<sys/mman.h>` header.

Symbolic Constant	Description
<code>PROT_READ</code>	Data can be read.
<code>PROT_WRITE</code>	Data can be written.
<code>PROT_EXEC</code>	Data can be executed.
<code>PROT_NONE</code>	Data cannot be accessed.

If an implementation cannot support the combination of access types specified by `prot`, the call to `mmap()` shall fail.

An implementation may permit accesses other than those specified by `prot`; however, the implementation shall not permit a write to succeed where `PROT_WRITE` has not been set and shall not permit any access where `PROT_NONE` alone has been set. The implementation shall support at least the following values of `prot`: `PROT_NONE`, `PROT_READ`, `PROT_WRITE`, and the bitwise-inclusive OR of `PROT_READ` and `PROT_WRITE`. The file descriptor `fildes` shall have been opened with read permission, regardless of the protection options specified. If `PROT_WRITE` is specified, the application shall ensure that it has opened the file descriptor `fildes` with write permission unless `MAP_PRIVATE` is specified in the `flags` parameter as described below.

The parameter `flags` provides other information about the handling of the mapped data. The value of `flags` is the bitwise-inclusive OR of these options,

defined in `<sys/mman.h>` :

Symbolic Constant	Description
MAP_ANON	Synonym for MAP_ANONYMOUS.
MAP_ANONYMOUS	Map anonymous memory.
MAP_SHARED	Changes are shared.
MAP_PRIVATE	Changes are private.
MAP_FIXED	Interpret addr exactly.

It is implementation-defined whether MAP\_FIXED shall be supported. [XSI] MAP\_FIXED shall be supported on XSI-conformant systems.

MAP\_SHARED and MAP\_PRIVATE describe the disposition of write references to the memory object. If MAP\_SHARED is specified, write references shall change the underlying object. If MAP\_PRIVATE is specified, modifications to the mapped data by the calling process shall be visible only to the calling process and shall not change the underlying object. It is unspecified whether modifications to the underlying object done after the MAP\_PRIVATE mapping is established are visible through the MAP\_PRIVATE mapping. Either MAP\_SHARED or MAP\_PRIVATE can be specified, but not both. The mapping type is retained across `fork()`.

The state of synchronization objects such as mutexes, semaphores, barriers, and conditional variables placed in shared memory mapped with MAP\_SHARED becomes undefined when the last region in any process containing the synchronization object is unmapped.

[TYM] When `fd` represents a typed memory object opened with either the `POSIX_TYPED_MEM_ALLOCATE` flag or the `POSIX_TYPED_MEM_ALLOCATE_CONTIG` flag, `mmap()` shall, if there are enough resources available, map `len` bytes allocated from the corresponding typed memory object which were not previously allocated to any process in any processor that may access that typed memory object. If there are not enough resources available, the function shall fail. If `fd` represents a typed memory object opened with the `POSIX_TYPED_MEM_ALLOCATE_CONTIG` flag, these allocated bytes shall be contiguous within the typed memory object. If `fd` represents a typed memory object opened with the `POSIX_TYPED_MEM_ALLOCATE` flag, these allocated bytes may be composed of non-contiguous fragments within the typed memory object. If `fd` represents a typed memory object opened with neither the

POSIX\_TYPED\_MEM\_ALLOCATE\_CONTIG flag nor the POSIX\_TYPED\_MEM\_ALLOCATE flag, `len` bytes starting at offset `off` within the typed memory object are mapped, exactly as when mapping a file or shared memory object. In this case, if two processes map an area of typed memory using the same `off` and `len` values and using file descriptors that refer to the same memory pool (either from the same port or from a different port), both processes shall map the same region of storage.

When `MAP_FIXED` is set in the `flags` argument, the implementation is informed that the value of `pa` shall be `addr`, exactly. If `MAP_FIXED` is set, `mmap()` may return `MAP_FAILED` and set `errno` to `[EINVAL]`. If a `MAP_FIXED` request is successful, then any previous mappings [ML|MLR] or memory locks for those whole pages containing any part of the address range [ `pa` , `pa` + `len` ) shall be removed, as if by an appropriate call to `munmap()`, before the new mapping is established.

When `MAP_FIXED` is not set, the implementation uses `addr` in an implementation-defined manner to arrive at `pa`. The `pa` so chosen shall be an area of the address space that the implementation deems suitable for a mapping of `len` bytes to the file. All implementations interpret an `addr` value of 0 as granting the implementation complete freedom in selecting `pa`, subject to constraints described below. A non-zero value of `addr` is taken to be a suggestion of a process address near which the mapping should be placed. When the implementation selects a value for `pa`, it never places a mapping at address 0, nor does it replace any extant mapping.

If `MAP_FIXED` is specified and `addr` is non-zero, it shall have the same remainder as the `off` parameter, modulo the page size as returned by `sysconf()` when passed `_SC_PAGESIZE` or `_SC_PAGE_SIZE`. The implementation may require that `off` is a multiple of the page size. If `MAP_FIXED` is specified, the implementation may require that `addr` is a multiple of the page size. The system performs mapping operations over whole pages. Thus, while the parameter `len` need not meet a size or alignment constraint, the system shall include, in any mapping operation, any partial page specified by the address range starting at `pa` and continuing for `len` bytes.

If `MAP_ANONYMOUS` (or its synonym `MAP_ANON`) is specified, `fildes` is -1, and `off` is 0, then `mmap()` shall ignore `fildes` and instead establish a mapping to a new anonymous memory object of size `len`. The effect of specifying `MAP_ANONYMOUS` (or `MAP_ANON`) with other values of `fildes` or `off` is unspecified. Anonymous memory objects shall be initialized to all bits zero.

The system shall always zero-fill any partial page at the end of an object. Further, the system shall never write out any modified portions of the last page of an object which are beyond its end. References within the address range starting at `pa`

and continuing for `len` bytes to whole pages following the end of an object shall result in delivery of a SIGBUS signal.

An implementation may generate SIGBUS signals when a reference would cause an error in the mapped object, such as out-of-space condition.

The `mmap()` function shall add an extra reference to the file associated with the file descriptor `fd` which is not removed by a subsequent `close()` on that file descriptor. This reference shall be removed when there are no more mappings to the file.

The last data access timestamp of the mapped file may be marked for update at any time between the `mmap()` call and the corresponding `munmap()` call. The initial read or write reference to a mapped region shall cause the file's last data access timestamp to be marked for update if it has not already been marked for update.

The last data modification and last file status change timestamps of a file that is mapped with `MAP_SHARED` and `PROT_WRITE` shall be marked for update at some point in the interval between a write reference to the mapped region and the next call to `msync()` with `MS_ASYNC` or `MS_SYNC` for that portion of the file by any process. If there is no such call and if the underlying file is modified as a result of a write reference, then these timestamps shall be marked for update at some time after the write reference.

There may be implementation-defined limits on the number of memory regions that can be mapped (per process or per system).

[XSI] If such a limit is imposed, whether the number of memory regions that can be mapped by a process is decreased by the use of `shmat()` is implementation-defined.

If `mmap()` fails for reasons other than [EBADF], [EINVAL], or [ENOTSUP], some of the mappings in the address range starting at `addr` and continuing for `len` bytes may have been unmapped.

## RETURN VALUE

Upon successful completion, the `mmap()` function shall return the address at which the mapping was placed (`pa`); otherwise, it shall return a value of `MAP_FAILED` and set `errno` to indicate the error. The symbol `MAP_FAILED` is defined in the `<sys/mman.h>` header. No successful return from `mmap()` shall return the value `MAP_FAILED`.

# ERRORS

The `mmap()` function shall fail if:

[EAGAIN]

[ML] The mapping could not be locked in memory, if required by `mlockall()`, due to a lack of resources.

[EINVAL]

The value of `len` is zero.

[EINVAL]

The value of `flags` is invalid (neither MAP\_PRIVATE nor MAP\_SHARED is set).

[EMFILE]

The number of mapped regions would exceed an implementation-defined limit (per process or per system).

[ENOMEM]

MAP\_FIXED was specified, and the range [`addr`, `addr` + `len`) exceeds that allowed for the address space of a process; or, if MAP\_FIXED was not specified and there is insufficient room in the address space to effect the mapping.

[ENOMEM]

[ML] The mapping could not be locked in memory, if required by `mlockall()`, because it would require more space than the system is able to supply.

[ENOMEM]

[TYM] Not enough unallocated memory resources remain in the typed memory object designated by `fdides` to allocate `len` bytes.

[ENOTSUP]

MAP\_FIXED or MAP\_PRIVATE was specified in the `flags` argument and the implementation does not support this functionality.

The implementation does not support the combination of accesses requested in the `prot` argument.

[ENXIO]

MAP\_FIXED was specified in `flags` and the combination of `addr`, `len`, and `off` is invalid for the specified object.

[ENXIO]

[TYM] The `fdides` argument refers to a typed memory object that is not accessible from the calling process.

In addition, if MAP\_ANONYMOUS (or MAP\_ANON) is not set in `flags`, the `mmap()` function shall fail if:

#### [EACCES]

The `fildes` argument is not open for read, regardless of the protection specified, or `fildes` is not open for write and `PROT_WRITE` was specified for a `MAP_SHARED` type mapping.

#### [EBADF]

The `fildes` argument is not a valid open file descriptor.

#### [ENODEV]

The `fildes` argument refers to a file whose type is not supported by `mmap()`.

#### [EOVERFLOW]

The file is a regular file and the value of `off` plus `len` exceeds the offset maximum established in the open file description associated with `fildes`.

#### [ENXIO]

Addresses in the range  $[\text{off}, \text{off} + \text{len}]$  are invalid for the object specified by `fildes`.

The `mmap()` function may fail if:

#### [EINVAL]

The `addr` argument (if `MAP_FIXED` was specified) or `off` is not a multiple of the page size as returned by `sysconf()`, or is considered invalid by the implementation.

---

## EXAMPLES

None.

## APPLICATION USAGE

Use of `mmap()` may reduce the amount of memory available to other memory allocation functions.

Use of `MAP_FIXED` may result in unspecified behavior in further use of `malloc()` and `shmat()`. The use of `MAP_FIXED` is discouraged, as it may prevent an implementation from making the most effective use of resources. Most implementations require that `off` and `addr` are multiples of the page size as returned by `sysconf()`.

The application must ensure correct synchronization when using `mmap()` in conjunction with any other file access method, such as `read()` and `write()`, standard input/output, and `shmat()`.

The `mmap()` function allows access to resources via address space manipulations, instead of `read()` / `write()`. Once a file is mapped, all a process has to do to access it is use the data at the address to which the file was mapped. So, using pseudo-code to illustrate the way in which an existing program might be changed to use `mmap()`, the following:

```
fildes = open(...)  
lseek(fildes, some_offset)  
read(fildes, buf, len)  
/* Use data in buf. */
```

becomes:

```
fildes = open(...)  
address = mmap(0, len, PROT_READ, MAP_PRIVATE, fildes, some_offset)  
/* Use data at address. */
```

## RATIONALE

After considering several other alternatives, it was decided to adopt the `mmap()` definition found in SVR4 for mapping memory objects into process address spaces. The SVR4 definition is minimal, in that it describes only what has been built, and what appears to be necessary for a general and portable mapping facility.

Note that while `mmap()` was first designed for mapping files, it is actually a general-purpose mapping facility. It can be used to map any appropriate object, such as memory, files, devices, and so on, into the address space of a process.

When a mapping is established, it is possible that the implementation may need to map more than is requested into the address space of the process because of hardware requirements. An application, however, cannot count on this behavior. Implementations that do not use a paged architecture may simply allocate a common memory region and return the address of it; such implementations probably do not allocate any more than is necessary. References past the end of the requested area are unspecified.

If an application requests a mapping that overlaps existing mappings in the process, it might be desirable that an implementation detect this and inform the application. However, if the program specifies a fixed address mapping (which requires some implementation knowledge to determine a suitable address, if the function is supported at all), then the program is presumed to be successfully managing its own address space and should be trusted when it asks to map over existing data structures. Furthermore, it is also desirable to make as few system

calls as possible, and it might be considered onerous to require an `munmap()` before an `mmap()` to the same address range. This volume of POSIX.1-2024 specifies that the new mapping replaces any existing mappings (implying an automatic `munmap()` on the address range), following existing practice in this regard. The standard developers also considered whether there should be a way for new mappings to overlay existing mappings, but found no existing practice for this.

It is not expected that all hardware implementations are able to support all combinations of permissions at all addresses. Implementations are required to disallow write access to mappings without write permission and to disallow access to mappings without any access permission. Other than these restrictions, implementations may allow access types other than those requested by the application. For example, if the application requests only `PROT_WRITE`, the implementation may also allow read access. A call to `mmap()` fails if the implementation cannot support allowing all the access requested by the application. For example, some implementations cannot support a request for both write access and execute access simultaneously. All implementations must support requests for no access, read access, write access, and both read and write access. Strictly conforming code must only rely on the required checks. These restrictions allow for portability across a wide range of hardware.

The `MAP_FIXED` address treatment is likely to fail for non-page-aligned values and for certain architecture-dependent address ranges. Conforming implementations cannot count on being able to choose address values for `MAP_FIXED` without utilizing non-portable, implementation-defined knowledge. Nonetheless, `MAP_FIXED` is provided as a standard interface conforming to existing practice for utilizing such knowledge when it is available.

Similarly, in order to allow implementations that do not support virtual addresses, support for directly specifying any mapping addresses via `MAP_FIXED` is not required and thus a conforming application may not count on it.

The `MAP_PRIVATE` function can be implemented efficiently when memory protection hardware is available. When such hardware is not available, implementations can implement such "mappings" by simply making a real copy of the relevant data into process private memory, though this tends to behave similarly to `read()`.

The function has been defined to allow for many different models of using shared memory. However, all uses are not equally portable across all machine architectures. In particular, the `mmap()` function allows the system as well as the application to specify the address at which to map a specific region of a memory object. The most portable way to use the function is always to let the system choose the address, specifying `NULL` as the value for the argument `addr` and not to specify `MAP_FIXED`.

If it is intended that a particular region of a memory object be mapped at the same address in a group of processes (on machines where this is even possible), then MAP\_FIXED can be used to pass in the desired mapping address. The system can still be used to choose the desired address if the first such mapping is made without specifying MAP\_FIXED, and then the resulting mapping address can be passed to subsequent processes for them to pass in via MAP\_FIXED. The availability of a specific address range cannot be guaranteed, in general.

The `mmap()` function can be used to map a region of memory that is larger than the current size of the object. Memory access within the mapping but beyond the current end of the underlying objects may result in SIGBUS signals being sent to the process. The reason for this is that the size of the object can be manipulated by other processes and can change at any moment. The implementation should tell the application that a memory reference is outside the object where this can be detected; otherwise, written data may be lost and read data may not reflect actual data in the object.

Note that references beyond the end of the object do not extend the object as the new end cannot be determined precisely by most virtual memory hardware. Instead, the size can be directly manipulated by `ftruncate()`.

Process memory locking does apply to shared memory regions, and the MCL\_FUTURE argument to `mlockall()` can be relied upon to cause new shared memory regions to be automatically locked.

Existing implementations of `mmap()` return the value -1 when unsuccessful. Since the casting of this value to type `void *` cannot be guaranteed by the ISO C standard to be distinct from a successful value, this volume of POSIX.1-2024 defines the symbol MAP\_FAILED, which a conforming implementation does not return as the result of a successful call.

Some historical implementations only supported MAP\_ANON, some only supported MAP\_ANONYMOUS, and some supported both spellings. This standard includes both spellings partly for application compatibility and partly because neither spelling was clearly more popular than the other at the time this feature was considered for standardization.

## FUTURE DIRECTIONS

None.

## SEE ALSO

`exec` , `fcntl()` , `fork()` , `lockf()` , `msync()` , `munmap()` , `mprotect()` ,  
`posix_typed_mem_open()` , `shmat()` , `sysconf()`

XBD `<sys/mman.h>`

## CHANGE HISTORY

First released in Issue 4, Version 2.

### Issue 5

Moved from X/OPEN UNIX extension to BASE.

Aligned with `mmap()` in the POSIX Realtime Extension as follows:

- The DESCRIPTION is extensively reworded.
- The [EAGAIN] and [ENOTSUP] mandatory error conditions are added.
- New cases of [ENOMEM] and [ENXIO] are added as mandatory error conditions.
- The value returned on failure is the value of the constant MAP\_FAILED; this was previously defined as -1.

Large File Summit extensions are added.

### Issue 6

The `mmap()` function is marked as part of the Memory Mapped Files option.

The Open Group Corrigendum U028/6 is applied, changing `(void *) -1` to `MAP_FAILED`.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION is updated to describe the use of MAP\_FIXED.
- The DESCRIPTION is updated to describe the addition of an extra reference to the file associated with the file descriptor passed to `mmap()`.
- The DESCRIPTION is updated to state that there may be implementation-defined limits on the number of memory regions that can be mapped.
- The DESCRIPTION is updated to describe constraints on the alignment and size of the `off` argument.

- The [EINVAL] and [EMFILE] error conditions are added.
- The [EOVERFLOW] error condition is added. This change is to support large files.

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

- The DESCRIPTION is updated to describe the cases when MAP\_PRIVATE and MAP\_FIXED need not be supported.

The following changes are made for alignment with IEEE Std 1003.1j-2000:

- Semantics for typed memory objects are added to the DESCRIPTION.
- New [ENOMEM] and [ENXIO] errors are added to the ERRORS section.
- The `posix_typed_mem_open()` function is added to the SEE ALSO section.

The normative text is updated to avoid use of the term "must" for application requirements.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/34 is applied, changing the margin code in the SYNOPSIS from MF|SHM to MC3 (notation for MF|SHM|TYM).

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/60 is applied, updating the DESCRIPTION and ERRORS sections to add the [EINVAL] error when `len` is zero.

## Issue 7

Austin Group Interpretations 1003.1-2001 #078 and #079 are applied, clarifying page alignment requirements and adding a note about the state of synchronization objects becoming undefined when a shared region is unmapped.

Functionality relating to the Memory Protection and Memory Mapped Files options is moved to the Base.

Changes are made related to support for finegrained timestamps.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0229 [852] is applied.

## Issue 8

Austin Group Defect 850 is applied, adding anonymous memory objects.

---

## 1.98. munlock

---

### SYNOPSIS

```
[MLR]
#include <sys/mman.h>

int mlock(const void *addr, size_t len);
int munlock(const void *addr, size_t len);
```

### DESCRIPTION

The `mlock()` function shall cause those whole pages containing any part of the address space of the process starting at address `addr` and continuing for `len` bytes to be memory-resident until unlocked or until the process exits or `exec`'s another process image. The implementation may require that `addr` be a multiple of {PAGESIZE}.

The `munlock()` function shall unlock those whole pages containing any part of the address space of the process starting at address `addr` and continuing for `len` bytes, regardless of how many times `mlock()` has been called by the process for any of the pages in the specified range. The implementation may require that `addr` be a multiple of {PAGESIZE}.

If any of the pages in the range specified to a call to `munlock()` are also mapped into the address spaces of other processes, any locks established on those pages by another process are unaffected by the call of this process to `munlock()`. If any of the pages in the range specified by a call to `munlock()` are also mapped into other portions of the address space of the calling process outside the range specified, any locks established on those pages via the other mappings are also unaffected by this call.

Upon successful return from `mlock()`, pages in the specified range shall be locked and memory-resident. Upon successful return from `munlock()`, pages in the specified range shall be unlocked with respect to the address space of the process. Memory residency of unlocked pages is unspecified.

Appropriate privileges are required to lock process memory with `mlock()`.

## RETURN VALUE

Upon successful completion, the `mlock()` and `munlock()` functions shall return a value of zero. Otherwise, no change is made to any locks in the address space of the process, and the function shall return a value of -1 and set `errno` to indicate the error.

## ERRORS

The `mlock()` and `munlock()` functions shall fail if:

- **[ENOMEM]**

Some or all of the address range specified by the `addr` and `len` arguments does not correspond to valid mapped pages in the address space of the process.

The `mlock()` function shall fail if:

- **[EAGAIN]**

Some or all of the memory identified by the operation could not be locked when the call was made.

The `mlock()` and `munlock()` functions may fail if:

- **[EINVAL]**

The `addr` argument is not a multiple of {PAGESIZE}.

The `mlock()` function may fail if:

- **[ENOMEM]**

Locking the pages mapped by the specified range would exceed an implementation-defined limit on the amount of memory that the process may lock.

- **[EPERM]**

The calling process does not have appropriate privileges to perform the requested operation.

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

`exec()` , `_exit()` , `fork()` , `mlockall()` , `munmap()`

XBD `<sys/mman.h>`

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

### Issue 6

The `mlock()` and `munlock()` functions are marked as part of the Range Memory Locking option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Range Memory Locking option.

## 1.99. munmap — unmap pages of memory

---

### SYNOPSIS

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

int munmap(void *addr, size_t len);
```

### DESCRIPTION

The `munmap()` function shall remove any mappings for those entire pages containing any part of the address space of the process starting at `addr` and continuing for `len` bytes. Further references to these pages shall result in the generation of a SIGSEGV signal to the process. If there are no mappings in the specified address range, then `munmap()` has no effect.

The implementation may require that `addr` be a multiple of the page size as returned by `sysconf()`.

If a mapping to be removed was private, any modifications made in this address range shall be discarded.

[ML|MLR] Any memory locks (see `mlock()` and `mlockall()`) associated with this address range shall be removed, as if by an appropriate call to `munlock()`.

[TYM] If a mapping removed from a typed memory object causes the corresponding address range of the memory pool to be inaccessible by any process in the system except through allocatable mappings (that is, mappings of typed memory objects opened with the `POSIX_TYPED_MEM_MAP_ALLOCATABLE` flag), then that range of the memory pool shall become deallocated and may become available to satisfy future typed memory allocation requests.

A mapping removed from a typed memory object opened with the `POSIX_TYPED_MEM_MAP_ALLOCATABLE` flag shall not affect in any way the availability of that typed memory for allocation.

The behavior of this function is unspecified if the mapping was not established by a call to `mmap()`.

## RETURN VALUE

Upon successful completion, `munmap()` shall return 0; otherwise, it shall return -1 and set `errno` to indicate the error.

## ERRORS

The `munmap()` function shall fail if:

- **[EINVAL]** Addresses in the range [`addr`, `addr` + `len`) are outside the valid range for the address space of a process.
- **[EINVAL]** The `len` argument is 0.

The `munmap()` function may fail if:

- **[EINVAL]** The `addr` argument is not a multiple of the page size as returned by `sysconf()`.

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

The `munmap()` function corresponds to SVR4, just as the `mmap()` function does.

It is possible that an application has applied process memory locking to a region that contains shared memory. If this has occurred, the `munmap()` call ignores those locks and, if necessary, causes those locks to be removed.

Most implementations require that `addr` is a multiple of the page size as returned by `sysconf()`.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `mlock()`
- `mlockall()`
- `mmap()`
- `posix_typed_mem_open()`
- `sysconf()`
- XBD `<sys/mman.h>`

## CHANGE HISTORY

First released in Issue 4, Version 2.

### Issue 5

Moved from X/OPEN UNIX extension to BASE.

Aligned with `munmap()` in the POSIX Realtime Extension as follows:

- The DESCRIPTION is extensively reworded.
- The SIGBUS error is no longer permitted to be generated.

### Issue 6

The `munmap()` function is marked as part of the Memory Mapped Files and Shared Memory Objects option.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION is updated to state that implementations require `addr` to be a multiple of the page size.
- The [EINVAL] error conditions are added.

The following changes are made for alignment with IEEE Std 1003.1j-2000:

- Semantics for typed memory objects are added to the DESCRIPTION.
- The `posix_typed_mem_open()` function is added to the SEE ALSO section.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/36 is applied, changing the margin code in the SYNOPSIS from MF|SHM to MC3 (notation for MF|SHM|TYM).

## Issue 7

Austin Group Interpretation 1003.1-2001 #078 is applied, clarifying page alignment requirements.

The `munmap()` function is moved from the Memory Mapped Files option to the Base.

---

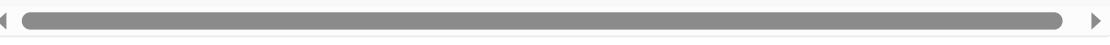
## 1.100. nanosleep — high resolution sleep

---

### SYNOPSIS

```
#include <time.h>

int nanosleep(const struct timespec *rqtp, struct timespec *rmtp);
```



### DESCRIPTION

The `nanosleep()` function shall cause the current thread to be suspended from execution until either the time interval specified by the `rqtp` argument has elapsed or a signal is delivered to the calling thread, and its action is to invoke a signal-catching function or to terminate the process. The suspension time may be longer than requested because the argument value is rounded up to an integer multiple of the sleep resolution or because of the scheduling of other activity by the system. But, except for the case of being interrupted by a signal, the suspension time shall not be less than the time specified by `rqtp`, as measured by the system clock `CLOCK_REALTIME`.

The use of the `nanosleep()` function has no effect on the action or blockage of any signal.

### RETURN VALUE

If the `nanosleep()` function returns because the requested time has elapsed, its return value shall be zero.

If the `nanosleep()` function returns because it has been interrupted by a signal, it shall return a value of -1 and set `errno` to indicate the interruption. If the `rmtp` argument is non-NULL, the `timespec` structure referenced by it is updated to contain the amount of time remaining in the interval (the requested time minus the time actually slept). The `rqtp` and `rmtp` arguments can point to the same object. If the `rmtp` argument is NULL, the remaining time is not returned.

If `nanosleep()` fails, it shall return a value of -1 and set `errno` to indicate the error.

# ERRORS

The `nanosleep()` function shall fail if:

- **[EINTR]**

The `nanosleep()` function was interrupted by a signal.

- **[EINVAL]**

The `rqtp` argument specified a nanosecond value less than zero or greater than or equal to 1000 million.

# EXAMPLES

None.

# APPLICATION USAGE

None.

# RATIONALE

It is common to suspend execution of a thread for an interval in order to poll the status of a non-interrupting function. A large number of actual needs can be met with a simple extension to `sleep()` that provides finer resolution.

In the POSIX.1-1990 standard and SVR4, it is possible to implement such a routine, but the frequency of wakeup is limited by the resolution of the `alarm()` and `sleep()` functions. In 4.3 BSD, it is possible to write such a routine using no static storage and reserving no system facilities. Although it is possible to write a function with similar functionality to `sleep()` using the remainder of the `timer_*` functions, such a function requires the use of signals and the reservation of some signal number. This volume of POSIX.1-2024 requires that `nanosleep()` be non-intrusive of the signals function.

The `nanosleep()` function shall return a value of 0 on success and -1 on failure or if interrupted. This latter case is different from `sleep()`. This was done because the remaining time is returned via an argument structure pointer, `rmtp`, instead of as the return value.

# FUTURE DIRECTIONS

None.

## SEE ALSO

- `clock_nanosleep()`
- `sleep()`
- `<time.h>`

## CHANGE HISTORY

### First released in Issue 5

Included for alignment with the POSIX Realtime Extension.

### Issue 6

- The `nanosleep()` function is marked as part of the Timers option.
- The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Timers option.
- IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/37 is applied, updating the SEE ALSO section to include the `clock_nanosleep()` function.
- IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/63 is applied, correcting text in the RATIONALE section.

### Issue 7

- SD5-XBD-ERN-33 is applied.
- The `nanosleep()` function is moved from the Timers option to the Base.
- POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0231 [909] is applied.

---

## 1.101. open, openat — open file

---

### SYNOPSIS

```
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *path, int oflag, ...);
int openat(int fd, const char *path, int oflag, ...);
```

### DESCRIPTION

The `open()` function shall establish the connection between a file and a file descriptor. It shall create an open file description that refers to a file and a file descriptor that refers to that open file description. The file descriptor is used by other I/O functions to refer to that file. The `path` argument points to a pathname naming the file.

The `open()` function shall return a file descriptor for the named file, allocated as described in [2.6 File Descriptor Allocation](#). The open file description is new, and therefore the file descriptor shall not share it with any other process in the system.

The `FD_CLOEXEC` file descriptor flag associated with the new file descriptor shall be cleared unless the `O_CLOEXEC` flag is set in `oflag`. The `FD_CLOFORK` file descriptor flag associated with the new file descriptor shall be cleared unless the `O_CLOFORK` flag is set in `oflag`.

The file offset used to mark the current position within the file shall be set to the beginning of the file.

The file status flags and file access modes of the open file description shall be set according to the value of `oflag`.

Values for `oflag` are constructed by a bitwise-inclusive OR of flags from the following list, defined in `<fcntl.h>`. Applications shall specify exactly one of the first five values (file access modes) below in the value of `oflag`:

#### File Access Modes

- `O_EXEC`: Open for execute only (non-directory files). If `path` names a directory and `O_EXEC` is not the same value as `O_SEARCH`, `open()` shall fail.
- `O_RDONLY`: Open for reading only.

- **O\_RDWR**: Open for reading and writing. If `path` names a FIFO, and the implementation does not support opening a FIFO for simultaneous read and write, then `open()` shall fail.
- **O\_SEARCH**: Open directory for search only. If `path` names a non-directory file and `O_SEARCH` is not the same value as `O_EXEC`, `open()` shall fail.
- **O\_WRONLY**: Open for writing only.

## Additional Flags

Any combination of the following may be used:

- **O\_APPEND**: If set, the file offset shall be set to the end of the file prior to each write.
- **O\_CLOEXEC**: If set, the `FD_CLOEXEC` flag for the new file descriptor shall be set.
- **O\_CLOFORK**: If set, the `FD_CLOFORK` flag for the new file descriptor shall be set.
- **O\_CREAT**: If the file exists, this flag has no effect except as noted under `O_EXCL` below. Otherwise, if `O_DIRECTORY` is not set the file shall be created as a regular file; the user ID of the file shall be set to the effective user ID of the process; the group ID of the file shall be set to the group ID of the file's parent directory or to the effective group ID of the process; and the access permission bits (see `<sys/stat.h>`) of the file mode shall be set to the value of the argument following the `oflag` argument taken as type `mode_t` modified as follows: a bitwise AND is performed on the file-mode bits and the corresponding bits in the complement of the process' file mode creation mask.
- **O\_DIRECTORY**: If `path` resolves to a non-directory file, fail and set `errno` to `[ENOTDIR]`.
- **O\_DSYNC** [SIO]: Write I/O operations on the file descriptor shall complete as defined by synchronized I/O data integrity completion.
- **O\_EXCL**: If `O_CREAT` and `O_EXCL` are set, `open()` shall fail if the file exists. The check for the existence of the file and the creation of the file if it does not exist shall be atomic with respect to other threads executing `open()` naming the same filename in the same directory with `O_EXCL` and `O_CREAT` set. If `O_EXCL` and `O_CREAT` are set, and `path` names a symbolic link, `open()` shall fail and set `errno` to `[EEXIST]`, regardless of the contents of the symbolic link.

- **O\_NOCTTY**: If set and `path` identifies a terminal device, `open()` shall not cause the terminal device to become the controlling terminal for the process.
- **O\_NOFOLLOW**: If `path` names a symbolic link, fail and set `errno` to `[ELOOP]`.
- **O\_NONBLOCK**: When opening a FIFO with `O_RDONLY` or `O_WRONLY` set:
  - If `O_NONBLOCK` is set, an `open()` for reading-only shall return without delay. An `open()` for writing-only shall return an error if no process currently has the file open for reading.
  - If `O_NONBLOCK` is clear, an `open()` for reading-only shall block the calling thread until a thread opens the file for writing. An `open()` for writing-only shall block the calling thread until a thread opens the file for reading.

When opening a block special or character special file that supports non-blocking opens:

- If `O_NONBLOCK` is set, the `open()` function shall return without blocking for the device to be ready or available.
- If `O_NONBLOCK` is clear, the `open()` function shall block the calling thread until the device is ready or available before returning.
- **O\_RSYNC** [SIO]: Read I/O operations on the file descriptor shall complete at the same level of integrity as specified by the `O_DSYNC` and `O_SYNC` flags.
- **O\_SYNC** [XSI|SIO]: Write I/O operations on the file descriptor shall complete as defined by synchronized I/O file integrity completion. The `O_SYNC` flag shall be supported for regular files, even if the Synchronized Input and Output option is not supported.
- **O\_TRUNC**: If the file exists and is a regular file, and the file is successfully opened `O_RDWR` or `O_WRONLY`, its length shall be truncated to 0, and the mode and owner shall be unchanged.
- **O\_TTY\_INIT**: If `path` identifies a terminal device other than a pseudo-terminal, the device is not already open in any process, and either `O_TTY_INIT` is set in `oflag` or `O_TTY_INIT` has the value zero, `open()` shall set any non-standard `termios` structure terminal parameters to a state that provides conforming behavior and initialize the `winsize` structure associated with the terminal to appropriate default settings.

The `openat()` function shall be equivalent to the `open()` function except in the case where `path` specifies a relative path. In this case the file to be opened is determined relative to the directory associated with the file descriptor `fd` instead of the current working directory.

If `openat()` is passed the special value `AT_FDCWD` in the `fd` parameter, the current working directory shall be used and the behavior shall be identical to a call to `open()`.

## RETURN VALUE

Upon successful completion, these functions shall open the file and return a non-negative integer representing the file descriptor. Otherwise, these functions shall return -1 and set `errno` to indicate the error. If -1 is returned, no files shall be created or modified.

## ERRORS

These functions shall fail if:

- **[EACCES]**: Search permission is denied on a component of the path prefix, or the file exists and the permissions specified by `oflag` are denied, or the file does not exist and write permission is denied for the parent directory of the file to be created, or `O_TRUNC` is specified and write permission is denied.
- **[EEXIST]**: `O_CREAT` and `O_EXCL` are set, and the named file exists.
- **[EILSEQ]**: `O_CREAT` was specified, the file did not exist, and the last pathname component of `path` is not a portable filename and cannot be created in the target directory.
- **[EINTR]**: A signal was caught during `open()`.
- **[EINVAL]**: The `path` argument names a FIFO, `O_RDWR` was specified, and the implementation considers this an error; or synchronized I/O flags were specified and the implementation does not support synchronized I/O for the file.
- **[EISDIR]**: The named file is a directory and `oflag` includes `O_WRONLY` or `O_RDWR`, or includes `O_CREAT` without `O_DIRECTORY`, or includes `O_EXEC` when `O_EXEC` is not the same value as `O_SEARCH`.
- **[ELOOP]**: A loop exists in symbolic links encountered during resolution of the `path` argument, or `O_NOFOLLOW` was specified and the `path` argument names a symbolic link.
- **[EMFILE]**: All file descriptors available to the process are currently open.
- **[ENAMETOOLONG]**: The length of a component of a pathname is longer than `{NAME_MAX}`.

- **[ENFILE]**: The maximum allowable number of files is currently open in the system.
- **[ENOENT]**: `O_CREAT` is not set and a component of `path` does not name an existing file, or `O_CREAT` is set and a component of the path prefix of `path` does not name an existing file, or `path` points to an empty string.
- **[ENOENT]** or **[ENOTDIR]**: `O_CREAT` is set, and the `path` argument contains at least one non-`<slash>` character and ends with one or more trailing `<slash>` characters.
- **[ENOSPC]**: The directory or file system that would contain the new file cannot be expanded, the file does not exist, and `O_CREAT` is specified.
- **[ENOTDIR]**: A component of the path prefix names an existing file that is neither a directory nor a symbolic link to a directory; or `O_CREAT` and `O_EXCL` are not specified, the `path` argument contains at least one non-`<slash>` character and ends with one or more trailing `<slash>` characters; or `O_DIRECTORY` was specified and the `path` argument names a non-directory file.
- **[ENXIO]**: `O_NONBLOCK` is set, the named file is a FIFO, `O_WRONLY` is set, and no process has the file open for reading.
- **[ENXIO]**: The named file is a character special or block special file, and the device associated with this special file does not exist.
- **[EOVERFLOW]**: The named file is a regular file and the size of the file cannot be represented correctly in an object of type `off_t`.
- **[EROFS]**: The named file resides on a read-only file system and either `O_WRONLY`, `O_RDWR`, `O_CREAT` (if the file does not exist), or `O_TRUNC` is set in the `oflag` argument.

The `openat()` function shall fail if:

- **[EACCES]**: The access mode of the open file description associated with `fd` is not `O_SEARCH` and the permissions of the directory underlying `fd` do not permit directory searches.
- **[EBADF]**: The `path` argument does not specify an absolute path and the `fd` argument is neither `AT_FDCWD` nor a valid file descriptor open for reading or searching.
- **[ENOTDIR]**: The `path` argument is not an absolute path and `fd` is a file descriptor associated with a non-directory file.

These functions may fail if:

- **[EAGAIN]** [XSI]: The `path` argument names the subsidiary side of a pseudo-terminal device that is locked.
- **[EINVAL]**: The value of the `oflag` argument is not valid.
- **[ELOOP]**: More than `{SYMLOOP_MAX}` symbolic links were encountered during resolution of the `path` argument.
- **[ENAMETOOLONG]**: The length of a pathname exceeds `{PATH_MAX}`, or pathname resolution of a symbolic link produced an intermediate result with a length that exceeds `{PATH_MAX}`.
- **[EOPNOTSUPP]**: The `path` argument names a socket.
- **[ETXTBSY]**: The file is a pure procedure (shared text) file that is being executed and `oflag` is `O_WRONLY` or `O_RDWR`.

## EXAMPLES

### Opening a File for Writing by the Owner

```
#include <fcntl.h>
...
int fd;
mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
char *pathname = "/tmp/file";
...
fd = open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);
...
```

### Opening a File Using an Existence Check

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

#define LOCKFILE "/etc/ptmp"
...
int pfd; /* Integer for file descriptor returned by open() call. */
...
if ((pfd = open(LOCKFILE, O_WRONLY | O_CREAT | O_EXCL,
    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
{
    fprintf(stderr, "Cannot open /etc/ptmp. Try again later.\n");
    exit(1);
```

```
}
```

```
...
```

## Opening a File for Writing

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

#define LOCKFILE "/etc/ptmp"

...
int pfd;
char pathname[PATH_MAX+1];
...

if ((pfd = open(pathname, O_WRONLY | O_CREAT | O_TRUNC,
    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
{
    perror("Cannot open output file\n");
    exit(1);
}
...
```

## APPLICATION USAGE

POSIX.1-2024 does not require that terminal parameters be automatically set to any state on first open, nor that they be reset after the last close. To ensure that the device is set to a conforming initial state, applications which perform a first open of a terminal (other than a pseudo-terminal) should do so using the `O_TTY_INIT` flag.

Except as specified in this volume of POSIX.1-2024, the flags allowed in `oflag` are not mutually-exclusive and any number of them may be used simultaneously.

The `O_CLOEXEC` and `O_CLOFORK` flags of `open()` are necessary to avoid a data race in multi-threaded applications.

## SEE ALSO

`chmod()`, `close()`, `creat()`, `dirfd()`, `dup()`, `exec`, `fcntl()`,  
`fdopendir()`, `link()`, `lseek()`, `mkdtemp()`, `mknod()`, `read()`,  
`symlink()`, `umask()`, `unlockpt()`, `write()`

XBD 11. General Terminal Interface, `<fcntl.h>`, `<sys/stat.h>`,  
`<sys/types.h>`



## 1.102. pause

---

### SYNOPSIS

```
#include <unistd.h>

int pause(void);
```

### DESCRIPTION

The `pause()` function shall suspend the calling thread until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process.

If the action is to terminate the process, `pause()` shall not return.

If the action is to execute a signal-catching function, `pause()` shall return after the signal-catching function returns.

### RETURN VALUE

Since `pause()` suspends thread execution indefinitely unless interrupted by a signal, there is no successful completion return value. A value of -1 shall be returned and `errno` set to indicate the error.

### ERRORS

The `pause()` function shall fail if:

- **[EINTR]**

A signal is caught by the calling process and control is returned from the signal-catching function.

*The following sections are informative.*

### EXAMPLES

None.

## APPLICATION USAGE

Many common uses of `pause()` have timing windows. The scenario involves checking a condition related to a signal and, if the signal has not occurred, calling `pause()`. When the signal occurs between the check and the call to `pause()`, the process often blocks indefinitely. The `sigprocmask()` and `sigsuspend()` functions can be used to avoid this type of problem.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `sigsuspend()`
- XBD `<unistd.h>`

## CHANGE HISTORY

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

### Issue 5

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

### Issue 6

The APPLICATION USAGE section is added.

## 1.103. perror

---

### SYNOPSIS

```
#include <stdio.h>

void perror(const char *s);
```

### DESCRIPTION

The `perror()` function shall map the error number accessed through the symbol `errno` to a language-dependent error message, which shall be written to the standard error stream as follows:

- First (if `s` is not a null pointer and the character pointed to by `s` is not the null byte), the string pointed to by `s` followed by a `<colon>` and a `<space>`.
- Then an error message string followed by a `<newline>`.

The contents of the error message strings shall be the same as those returned by `strerror()` with argument `errno`.

The `perror()` function shall mark for update the last data modification and last file status change timestamps of the file associated with the standard error stream at some time between its successful completion and `exit()`, `abort()`, or the completion of `fflush()` or `fclose()` on `stderr`.

The `perror()` function shall not change the orientation of the standard error stream.

On error, `perror()` shall set the error indicator for the stream to which `stderr` points, and shall set `errno` to indicate the error.

Since no value is returned, an application wishing to check for error situations should call `clearerr(stderr)` before calling `perror()`, then if `ferror(stderr)` returns non-zero, the value of `errno` indicates which error occurred.

### RETURN VALUE

The `perror()` function shall not return a value.

# ERRORS

Refer to [fputc\(\)](#).

---

## EXAMPLES

### Printing an Error Message for a Function

The following example replaces `bufptr` with a buffer that is the necessary size. If an error occurs, the [perror\(\)](#) function prints a message and the program exits.

```
#include <stdio.h>
#include <stdlib.h>
...
char *bufptr;
size_t szbuf;
...
if ((bufptr = malloc(szbuf)) == NULL) {
    perror("malloc");
    exit(2);
}
...
```

## APPLICATION USAGE

Application writers may prefer to use alternative interfaces instead of [perror\(\)](#), such as [strerror\\_r\(\)](#) in combination with [fprintf\(\)](#).

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [fprintf\(\)](#)

- `fputc()`
- `psiginfo()`
- `strerror()`
- `<stdio.h>`

## CHANGE HISTORY

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

**Issue 5:** A paragraph is added to the DESCRIPTION indicating that `perror()` does not change the orientation of the standard error stream.

**Issue 6:** Extensions beyond the ISO C standard are marked.

**Issue 7:** SD5-XSH-ERN-95 is applied. Changes are made related to support for finegrained timestamps. POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0429 [389,401], XSH/TC1-2008/0430 [389], and XSH/TC1-2008/0431 [389,401] are applied.

---

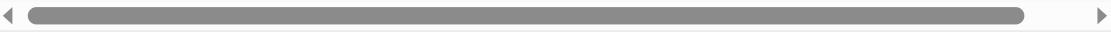
## 1.104. printf

---

### SYNOPSIS

```
#include <stdio.h>

int asprintf(char **restrict ptr, const char *restrict format, ...
int dprintf(int fildes, const char *restrict format, ...);
int fprintf(FILE *restrict stream, const char *restrict format, ...
int printf(const char *restrict format, ...);
int snprintf(char *restrict s, size_t n,
             const char *restrict format, ...);
int sprintf(char *restrict s, const char *restrict format, ...);
```



### DESCRIPTION

Except for asprintf(), dprintf(), and the behavior of the %lc conversion when passed a null wide character, the functionality described on this reference page is aligned with the ISO C standard. Any other conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard for all fprintf(), printf(), snprintf(), and sprintf() functionality except in relation to the %lc conversion when passed a null wide character.

The fprintf() function shall place output on the named output stream. The printf() function shall place output on the standard output stream stdout. The sprintf() function shall place output followed by the null byte, '\0', in consecutive bytes starting at \*s; it is the user's responsibility to ensure that enough space is available.

The asprintf() function shall be equivalent to sprintf(), except that the output string shall be written to dynamically allocated memory, allocated as if by a call to malloc(), of sufficient length to hold the resulting string, including a terminating null byte. If the call to asprintf() is successful, the address of this dynamically allocated string shall be stored in the location referenced by ptr.

The dprintf() function shall be equivalent to the fprintf() function, except that dprintf() shall write output to the file associated with the file descriptor specified by the fildes argument rather than place output on a stream.

The snprintf() function shall be equivalent to sprintf(), with the addition of the n argument which limits the number of bytes written to the buffer referred to by s. If n is zero, nothing shall be written and s may be a null pointer. Otherwise, output bytes beyond the n-1st shall be discarded instead of being written to the array, and a null byte is written at the end of the bytes actually written into the array.

If copying takes place between objects that overlap as a result of a call to `sprintf()` or `snprintf()`, the results are undefined.

Each of these functions converts, formats, and prints its arguments under control of the format. The application shall ensure that the format is a character string, beginning and ending in its initial shift state, if any. The format is composed of zero or more directives: ordinary characters, which are simply copied to the output stream, and conversion specifications, each of which shall result in the fetching of zero or more arguments. The results are undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments shall be evaluated but are otherwise ignored.

Conversions can be applied to the *n*th argument after the format in the argument list, rather than to the next unused argument. In this case, the conversion specifier character % (see below) is replaced by the sequence "%*n*\$", where *n* is a decimal integer in the range [1,{NL\_ARGMAX}], giving the position of the argument in the argument list. This feature provides for the definition of format strings that select arguments in an order appropriate to specific languages (see the EXAMPLES section).

The format can contain either numbered argument conversion specifications (that is, those introduced by "%*n*\$" and optionally containing the "\**m*\$" forms of field width and precision), or unnumbered argument conversion specifications (that is, those introduced by the % character and optionally containing the \* form of field width and precision), but not both. The only exception to this is that %% can be mixed with the "%*n*\$" form. The results of mixing numbered and unnumbered argument specifications in a format string are undefined. When numbered argument specifications are used, specifying the *N*th argument requires that all the leading arguments, from the first to the (*N*-1)th, are specified in the format string.

In format strings containing the "%*n*\$" form of conversion specification, numbered arguments in the argument list can be referenced from the format string as many times as required.

In format strings containing the % form of conversion specification, each conversion specification uses the first unused argument in the argument list.

All forms of the `fprintf()` functions allow for the insertion of a language-dependent radix character in the output string. The radix character is defined in the current locale (category LC\_NUMERIC). In the POSIX locale, or in a locale where the radix character is not defined, the radix character shall default to a period ('.') .

Each conversion specification is introduced by the '%' character or by the character sequence "%*n*\$", after which the following appear in sequence:

1. Zero or more flags (in any order), which modify the meaning of the conversion specification.

2. An optional minimum field width. If the converted value has fewer bytes than the field width, it shall be padded with space characters by default on the left; it shall be padded on the right if the left-adjustment flag ('-'), described below, is given to the field width. The field width takes the form of an asterisk ('\*'), *or in conversion specifications introduced by "%n\$"* the "m\$" string, described below, or a decimal integer.
3. An optional precision that gives the minimum number of digits to appear for the d, i, o, u, x, and X conversion specifiers; the number of digits to appear after the radix character for the a, A, e, E, f, and F conversion specifiers; the maximum number of significant digits for the g and G conversion specifiers; or the maximum number of bytes to be printed from a string in the s and S conversion specifiers. The precision takes the form of a period ('.') followed either by an asterisk ('\*'), *or in conversion specifications introduced by "%n\$"* the "m\$" string, described below, or an optional decimal digit string, where a null digit string is treated as zero. If a precision appears with any other conversion specifier, the behavior is undefined.
4. An optional length modifier that specifies the size of the argument.
5. A conversion specifier character that indicates the type of conversion to be applied.

A field width, or precision, or both, may be indicated by an asterisk ('\*'). *In this case an argument of type int supplies the field width or precision. Applications shall ensure that arguments specifying field width, or precision, or both appear in that order before the argument, if any, to be converted. A negative field width is taken as a '-' flag followed by a positive field width. A negative precision is taken as if the precision were omitted. In format strings containing conversion specifications introduced by "%n\$", in addition to being indicated by the decimal digit string, a field width may be indicated by the sequence "m\$" and precision by the sequence ".\*m\$"*, where m is a decimal integer in the range [1,{NL\_ARGMAX}] giving the position in the argument list (after the format argument) of an integer argument containing the field width or precision, for example:

```
printf("%1$d:%2$.*3$d:%4$.*3$d\n", hour, min, precision, sec);
```

## Flag Characters

The flag characters and their meanings are:

- ' (apostrophe) - The integer portion of the result of a decimal conversion (%i, %d, %u, %f, %F, %g, or %G) shall be formatted with thousands' grouping characters.

For other conversions the behavior is undefined. The non-monetary grouping character is used.

**-** - The result of the conversion shall be left-justified within the field. The conversion is right-justified if this flag is not specified.

**+** - The result of a signed conversion shall always begin with a sign ('+' or '-'). The conversion shall begin with a sign only when a negative value is converted if this flag is not specified.

**(space)** - If the first character of a signed conversion is not a sign or if a signed conversion results in no characters, a space shall be prefixed to the result. This means that if the space and '+' flags both appear, the space flag shall be ignored.

**#** - Specifies that the value is to be converted to an alternative form. For o conversion, it shall increase the precision, if and only if necessary, to force the first digit of the result to be a zero (if the value and precision are both 0, a single 0 is printed). For x or X conversion specifiers, a non-zero result shall have 0x (or 0X) prefixed to it. For a, A, e, E, f, F, g, and G conversion specifiers, the result shall always contain a radix character, even if no digits follow the radix character. Without this flag, a radix character appears in the result of these conversions only if a digit follows it. For g and G conversion specifiers, trailing zeros shall not be removed from the result as they normally are. For other conversion specifiers, the behavior is undefined.

**0** - For d, i, o, u, x, X, a, A, e, E, f, F, g, and G conversion specifiers, leading zeros (following any indication of sign or base) are used to pad to the field width rather than performing space padding, except when converting an infinity or NaN. If the '0' and '-' flags both appear, the '0' flag is ignored. For d, i, o, u, x, and X conversion specifiers, if a precision is specified, the '0' flag shall be ignored. If the '0' and apostrophe flags both appear, the grouping characters are inserted before zero padding. For other conversions, the behavior is undefined.

## Length Modifiers

The length modifiers and their meanings are:

**hh** - Specifies that a following d, i, o, u, x, or X conversion specifier applies to a signed char or unsigned char argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to signed char or unsigned char before printing); or that a following n conversion specifier applies to a pointer to a signed char argument.

**h** - Specifies that a following d, i, o, u, x, or X conversion specifier applies to a short or unsigned short argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to short or

unsigned short before printing); or that a following n conversion specifier applies to a pointer to a short argument.

**I** (ell) - Specifies that a following d, i, o, u, x, or X conversion specifier applies to a long or unsigned long argument; that a following n conversion specifier applies to a pointer to a long argument; that a following c conversion specifier applies to a wint\_t argument; that a following s conversion specifier applies to a pointer to a wchar\_t argument; or has no effect on a following a, A, e, E, f, F, g, or G conversion specifier.

**II** (ell-ell) - Specifies that a following d, i, o, u, x, or X conversion specifier applies to a long long or unsigned long long argument; or that a following n conversion specifier applies to a pointer to a long long argument.

**j** - Specifies that a following d, i, o, u, x, or X conversion specifier applies to an intmax\_t or uintmax\_t argument; or that a following n conversion specifier applies to a pointer to an intmax\_t argument.

**z** - Specifies that a following d, i, o, u, x, or X conversion specifier applies to a size\_t or the corresponding signed integer type argument; or that a following n conversion specifier applies to a pointer to a signed integer type corresponding to a size\_t argument.

**t** - Specifies that a following d, i, o, u, x, or X conversion specifier applies to a ptrdiff\_t or the corresponding unsigned type argument; or that a following n conversion specifier applies to a pointer to a ptrdiff\_t argument.

**L** - Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to a long double argument.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

## Conversion Specifiers

The conversion specifiers and their meanings are:

**d, i** - The int argument shall be converted to a signed decimal in the style "[-]dddd". The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1. The result of converting zero with an explicit precision of zero shall be no characters.

**o** - The unsigned argument shall be converted to unsigned octal format in the style "dddd". The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with

leading zeros. The default precision is 1. The result of converting zero with an explicit precision of zero shall be no characters.

**u** - The unsigned argument shall be converted to unsigned decimal format in the style "dddd". The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1. The result of converting zero with an explicit precision of zero shall be no characters.

**x** - The unsigned argument shall be converted to unsigned hexadecimal format in the style "dddd"; the letters "abcdef" are used. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1. The result of converting zero with an explicit precision of zero shall be no characters.

**X** - Equivalent to the x conversion specifier, except that letters "ABCDEF" are used instead of "abcdef".

**f, F** - The double argument shall be converted to decimal notation in the style "[-]ddd.ddd", where the number of digits after the radix character is equal to the precision specification. If the precision is missing, it shall be taken as 6; if the precision is explicitly zero and no '#' flag is present, no radix character shall appear. If a radix character appears, at least one digit appears before it. The low-order digit shall be rounded in an implementation-defined manner.

A double argument representing an infinity shall be converted in one of the styles "[-]inf" or "[-]infinity"; which style is implementation-defined. A double argument representing a NaN shall be converted in one of the styles "[-]nan(n-char-sequence)" or "[-]nan"; which style, and the meaning of any n-char-sequence, is implementation-defined. The F conversion specifier produces "INF", "INFINITY", or "NAN" instead of "inf", "infinity", or "nan", respectively.

**e, E** - The double argument shall be converted in the style "[-]d.ddd\_e±dd", where there is one digit before the radix character (which is non-zero if the argument is non-zero) and the number of digits after it is equal to the precision; if the precision is missing, it shall be taken as 6; if the precision is zero and no '#' flag is present, no radix character shall appear. The low-order digit shall be rounded in an implementation-defined manner. The E conversion specifier shall produce a number with 'E' instead of 'e' introducing the exponent. The exponent shall always contain at least two digits. If the value is zero, the exponent shall be zero.

A double argument representing an infinity or NaN shall be converted in the style of an f or F conversion specifier.

**g, G** - The double argument representing a floating-point number shall be converted in the style f or e (or in the style F or E in the case of a G conversion specifier), depending on the value converted and the precision. Let P equal the

precision if non-zero, 6 if the precision is omitted, or 1 if the precision is zero. Then, if a conversion with style E would have an exponent of X:

- If  $P > X \geq -4$ , the conversion shall be with style f (or F) and precision  $P-(X+1)$ .
- Otherwise, the conversion shall be with style e (or E) and precision P-1.

Finally, unless the '#' flag is used, any trailing zeros shall be removed from the fractional portion of the result and the decimal-point character shall be removed if there is no fractional portion remaining.

A double argument representing an infinity or NaN shall be converted in the style of an f or F conversion specifier.

**a, A** - A double argument representing a floating-point number shall be converted in the style "[-]0x\_h\_.hhhh\_p±d", where there is one hexadecimal digit (which shall be non-zero if the argument is a normalized floating-point number and is otherwise unspecified) before the decimal-point character and the number of hexadecimal digits after it is equal to the precision; if the precision is missing and FLT\_RADIX is a power of 2, then the precision shall be sufficient for an exact representation of the value; if the precision is missing and FLT\_RADIX is not a power of 2, then the precision shall be sufficient to distinguish values of type double, except that trailing zeros may be omitted; if the precision is zero and the '#' flag is not specified, no decimal-point character shall appear. The letters "abcdef" shall be used for a conversion and the letters "ABCDEF" for A conversion. The A conversion specifier produces a number with 'X' and 'P' instead of 'x' and 'p'. The exponent shall always contain at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent shall be zero.

A double argument representing an infinity or NaN shall be converted in the style of an f or F conversion specifier.

**c** - The int argument shall be converted to an unsigned char, and the resulting byte shall be written.

If an I (ell) qualifier is present, the wint\_t argument shall be converted to a multi-byte sequence as if by a call to wcrtomb() with a pointer to storage of at least MB\_CUR\_MAX bytes, the wint\_t argument converted to wchar\_t, and an initial shift state, and the resulting byte(s) written.

**s** - The argument shall be a pointer to an array of char. Bytes from the array shall be written up to (but not including) any terminating null byte. If the precision is specified, no more than that many bytes shall be written. If the precision is not specified or is greater than the size of the array, the application shall ensure that the array contains a null byte.

If an I (ell) qualifier is present, the argument shall be a pointer to an array of type wchar\_t. Wide characters from the array shall be converted to characters (each as if by a call to the wcrtomb() function, with the conversion state described by an

mbstate\_t object initialized to zero before the first wide character is converted) up to and including a terminating null wide character. The resulting characters shall be written up to (but not including) the terminating null character (byte). If no precision is specified, the application shall ensure that the array contains a null wide character. If a precision is specified, no more than that many characters (bytes) shall be written (including shift sequences, if any), and the array shall contain a null wide character if, to equal the character sequence length given by the precision, the function would need to access a wide character one past the end of the array. In no case shall a partial character be written.

**p** - The argument shall be a pointer to void. The value of the pointer is converted to a sequence of printable characters, in an implementation-defined manner.

**n** - The argument shall be a pointer to an integer into which is written the number of bytes written to the output so far by this call to one of the fprintf() functions. No argument is converted.

**C** - Equivalent to lc.

**S** - Equivalent to ls.

**%** - Write a '%' character; no argument shall be converted. The application shall ensure that the complete conversion specification is %%.

If a conversion specification does not match one of the above forms, the behavior is undefined. If any argument is not the correct type for the corresponding conversion specification, the behavior is undefined.

In no case shall a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field shall be expanded to contain the conversion result. Characters generated by fprintf() and printf() are printed as if fputc() had been called.

For the a and A conversion specifiers, if FLT\_RADIX is a power of 2, the value shall be correctly rounded to a hexadecimal floating number with the given precision.

For a and A conversions, if FLT\_RADIX is not a power of 2 and the result is not exactly representable in the given precision, the result should be one of the two adjacent numbers in hexadecimal floating style with the given precision, with the extra stipulation that the error should have a correct sign for the current rounding direction.

For the e, E, f, F, g, and G conversion specifiers, if the number of significant decimal digits is at most DECIMAL\_DIG, then the result should be correctly rounded. If the number of significant decimal digits is more than DECIMAL\_DIG but the source value is exactly representable with DECIMAL\_DIG digits, then the result should be an exact representation with trailing zeros. Otherwise, the source value is bounded by two adjacent decimal strings L < U, both having

DECIMAL\_DIG significant digits; the value of the resultant decimal string D should satisfy L <= D <= U, with the extra stipulation that the error should have a correct sign for the current rounding direction.

The last data modification and last file status change timestamps of the file shall be marked for update:

1. Between the call to a successful execution of fprintf() or printf() and the next successful completion of a call to fflush() or fclose() on the same stream or a call to exit() or abort()
2. Upon successful completion of a call to dprintf()

## RETURN VALUE

Upon successful completion, the dprintf(), fprintf(), and printf() functions shall return the number of bytes transmitted.

Upon successful completion, the asprintf() function shall return the number of bytes written to the allocated string stored in the location referenced by ptr, excluding the terminating null byte.

Upon successful completion, the sprintf() function shall return the number of bytes written to s, excluding the terminating null byte.

Upon successful completion, the snprintf() function shall return the number of bytes that would be written to s had n been sufficiently large excluding the terminating null byte.

If an error was encountered, these functions shall return a negative value and set errno to indicate the error. For asprintf(), if memory allocation was not possible, or if some other error occurs, the function shall return a negative value, and the contents of the location referenced by ptr are undefined, but shall not refer to allocated memory.

If the value of n is zero on a call to snprintf(), nothing shall be written, the number of bytes that would have been written had n been sufficiently large excluding the terminating null shall be returned, and s may be a null pointer.

## ERRORS

For the conditions under which dprintf(), fprintf(), and printf() fail and may fail, refer to fputc() or fputwc().

In addition, all forms of fprintf() shall fail if:

- **[EILSEQ]** - A wide-character code that does not correspond to a valid character has been detected.
- **[OVERFLOW]** - The value to be returned is greater than {INT\_MAX}.

The asprintf() function shall fail if:

- **[ENOMEM]** - Insufficient storage space is available.

The dprintf() function may fail if:

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

The dprintf(), fprintf(), and printf() functions may fail if:

- **[ENOMEM]** - Insufficient storage space is available.

## EXAMPLES

### Printing Language-Independent Date and Time

The following statement can be used to print date and time using a language-independent format:

```
printf(format, weekday, month, day, hour, min);
```

For American usage, format could be a pointer to the following string:

```
"%s, %s %d, %d:%.2d\n"
```

This example would produce the following message:

```
Sunday, July 3, 10:02
```

For German usage, format could be a pointer to the following string:

```
"%1$s, %3$d. %2$s, %4$d:%5$.2d\n"
```

This definition of format would produce the following message:

```
Sonntag, 3. Juli, 10:02
```

### Printing File Information

The following example prints information about the type, permissions, and number of links of a specific file in a directory.

```

#include <stdio.h>
#include <sys/types.h>
#include <pwd.h>
#include <grp.h>

char *strperm (mode_t);
...
struct stat statbuf;
struct passwd *pwd;
struct group *grp;
...
printf("%10.10s", strperm (statbuf.st_mode));
printf("%4d", statbuf.st_nlink);

if ((pwd = getpwuid(statbuf.st_uid)) != NULL)
    printf(" %-8.8s", pwd->pw_name);
else
    printf(" %-8ld", (long) statbuf.st_uid);

if ((grp = getgrgid(statbuf.st_gid)) != NULL)
    printf(" %-8.8s", grp->gr_name);
else
    printf(" %-8ld", (long) statbuf.st_gid);

printf("%9jd", (intmax_t) statbuf.st_size);

```

## Printing a Localized Date String

```

#include <stdio.h>
#include <time.h>
#include <langinfo.h>
...
struct dirent *dp;
struct tm *tm;
char datestring[256];
...
strftime(datestring, sizeof(datestring), nl_langinfo (D_T_FMT), tm
printf(" %s %s\n", datestring, dp->d_name);

```

## Printing Error Information

```

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

```

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

#define LOCKFILE "/etc/ptmp"
#define PASSWDFILE "/etc/passwd"
...
int pfd;
...
if ((pfd = open(LOCKFILE, O_WRONLY | O_CREAT | O_EXCL,
    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
{
    fprintf(stderr, "Cannot open /etc/ptmp. Try again later.\n");
    exit(1);
}
...
if (link(LOCKFILE,PASSWDFILE) == -1) {
    fprintf(stderr, "Link error: %s\n", strerror(errno));
    exit(1);
}
```

## Printing Usage Information

```
#include <stdio.h>
#include <stdlib.h>
...
char *Options = "hdbtl";
...
if (argc < 2) {
    fprintf(stderr, "Usage: %s -%s <file\n", argv[0], Options); exit(1);
}
```

## Formatting a Decimal String

```
#include <stdio.h>
...
long i;
char *keystr;
int elementlen, len;
...
while (len < elementlen) {
...
    printf("%s Element%0*ld\n", keystr, elementlen, i);
```

```
...  
}
```

## Creating a Pathname

```
#include <stdint.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <sys/types.h>  
#include <unistd.h>  
...  
char *pathname;  
struct passwd *pw;  
size_t len;  
...  
// digits required for pid_t is number of bits times  
// log2(10) = approx 10/33  
len = strlen(pw->pw_dir) + 1 + 1+(sizeof(pid_t)*80+32)/33 +  
    sizeof ".out";  
pathname = malloc(len);  
if (pathname != NULL)  
{  
    snprintf(pathname, len, "%s/%jd.out", pw->pw_dir,  
            (intmax_t)getpid());  
    ...  
}
```

## Reporting an Event

```
#include <stdio.h>  
#include <unistd.h>  
#include <string.h>  
#include <errno.h>  
...  
while (!event_complete) {  
    ...  
    if (pause() != -1 || errno != EINTR)  
        fprintf(stderr, "pause: unknown error: %s\n", strerror(errno));  
}
```

## Printing Monetary Information

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

```

...
struct tblfmt {
    char *format;
    char *description;
};

struct tblfmt table[] = {
    { "%n", "default formatting" },
    { "%11n", "right align within an 11 character field" },
    { "%#5n", "aligned columns for values up to 99999" },
    { "%=*#5n", "specify a fill character" },
    { "%=0#5n", "fill characters do not use grouping" },
    { "%^#5n", "disable the grouping separator" },
    { "%^#5.0n", "round off to whole units" },
    { "%^#5.4n", "increase the precision" },
    { "%(#5n", "use an alternative pos/neg style" },
    { "%!(#5n", "disable the currency symbol" },
};

...
float input[3];
int i, j;
char convbuf[100];
...

strfmon(convbuf, sizeof(convbuf), table[i].format, input[j]);

if (j == 0) {
    printf("%s %s      %s\n", table[i].format,
           convbuf, table[i].description);
}
else {
    printf("      %s\n", convbuf);
}

```

## Printing Wide Characters

The following example prints a series of wide characters. Suppose that "L@@" expands to three bytes:

```

wchar_t wz [3] = L"@@";           // Zero-terminated
wchar_t wn [3] = L"@@";           // Untermminated

fprintf (stdout,"%ls", wz);      // Outputs 6 bytes
fprintf (stdout,"%ls", wn);      // Undefined because wn has no terminator
fprintf (stdout,"%4ls", wz);     // Outputs 3 bytes
fprintf (stdout,"%4ls", wn);     // Outputs 3 bytes; no terminator needed
fprintf (stdout,"%9ls", wz);     // Outputs 6 bytes
fprintf (stdout,"%9ls", wn);     // Outputs 9 bytes; no terminator needed

```

```
fprintf (stdout,"%10ls", wz); // Outputs 6 bytes
fprintf (stdout,"%10ls", wn); // Undefined because wn has no termini
```

In the last line of the example, after processing three characters, nine bytes have been output. The fourth character must then be examined to determine whether it converts to one byte or more. If it converts to more than one byte, the output is only nine bytes. Since there is no fourth character in the array, the behavior is undefined.

## APPLICATION USAGE

If the application calling `fprintf()` has any objects of type `wint_t` or `wchar_t`, it must also include the header to have these objects defined.

The space allocated by a successful call to `asprintf()` should be subsequently freed by a call to `free()`.

## RATIONALE

If an implementation detects that there are insufficient arguments for the format, it is recommended that the function should fail and report an `[EINVAL]` error.

The behavior specified for the `%lc` conversion differs slightly from the specification in the ISO C standard, in that printing the null wide character produces a null byte instead of 0 bytes of output as would be required by a strict reading of the ISO C standard's direction to behave as if applying the `%ls` specifier to a `wchar_t` array whose first element is the null wide character. Requiring a multi-byte output for every possible wide character, including the null character, matches historical practice, and provides consistency with `%c` in `fprintf()` and with both `%c` and `%lc` in `fwprintf()`. It is anticipated that a future edition of the ISO C standard will change to match the behavior specified here.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- 2.5 Standard I/O Streams
- `fputc()`

- `fscanf()`
- `setlocale()`
- `strfmon()`
- `strlcat()`
- `wcrtomb()`
- `wcslcat()`

XBD:

- 7. Locale
- 
- 
- 

## CHANGE HISTORY

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

### Issue 5

Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the `l` (ell) qualifier can now be used with `c` and `s` conversion specifiers.

The `snprintf()` function is new in Issue 5.

### Issue 6

Extensions beyond the ISO C standard are marked.

The normative text is updated to avoid use of the term "must" for application requirements.

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

- The prototypes for `fprintf()`, `printf()`, `snprintf()`, and `sprintf()` are updated, and the XSI shading is removed from `snprintf()`.
- The description of `snprintf()` is aligned with the ISO C standard. Note that this supersedes the `snprintf()` description in The Open Group Base Resolution bwg98-006, which changed the behavior from Issue 5.
- The DESCRIPTION is updated.

The DESCRIPTION is updated to use the terms "conversion specifier" and "conversion specification" consistently.

ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

An example of printing wide characters is added.

## Issue 7

Austin Group Interpretation 1003.1-2001 #161 is applied, updating the DESCRIPTION of the 0 flag.

Austin Group Interpretation 1003.1-2001 #170 is applied.

ISO/IEC 9899:1999 standard, Technical Corrigendum 2 #68 (SD5-XSH-ERN-70) is applied, revising the description of g and G.

SD5-XSH-ERN-174 is applied.

The dprintf() function is added from The Open Group Technical Standard, 2006, Extended API Set Part 1.

Functionality relating to the %n\$ form of conversion specification and the apostrophe flag is moved from the XSI option to the Base.

Changes are made related to support for finegrained timestamps.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0163 [302], XSH/TC1-2008/0164 [316], XSH/TC1-2008/0165 [316], XSH/TC1-2008/0166 [451,291], and XSH/TC1-2008/0167 [14] are applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0126 [894], XSH/TC2-2008/0127 [557], and XSH/TC2-2008/0128 [936] are applied.

## Issue 8

Austin Group Defect 986 is applied, adding strlcat() and wcslcat() to the SEE ALSO section.

Austin Group Defect 1020 is applied, clarifying that the snprintf() argument n limits the number of bytes written to s; it is not necessarily the same as the size of s.

Austin Group Defect 1021 is applied, changing "output error" to "error" in the RETURN VALUE section.

Austin Group Defect 1137 is applied, clarifying the use of "%n\$" and "\*m\$" in conversion specifications.

Austin Group Defect 1205 is applied, changing the description of the % conversion specifier.

Austin Group Defect 1219 is applied, removing the `snprintf()`-specific [EOVERFLOW] error.

Austin Group Defect 1496 is applied, adding the `asprintf()` function.

Austin Group Defect 1562 is applied, clarifying that it is the application's responsibility to ensure that the format is a character string, beginning and ending in its initial shift state, if any.

Austin Group Defect 1647 is applied, changing the description of the `c` conversion specifier and updating the statement that this volume of POSIX.1-2024 defers to the ISO C standard so that it excludes the `%lc` conversion when passed a null wide character.

## 1.105. pthread\_atfork

---

### Synopsis

```
#include <pthread.h>

int pthread_atfork(void (*prepare)(void), void (*parent)(void), vo:
```



### Description

The following sections are informative.

#### Rationale

There are at least two serious problems with the semantics of `fork()` in a multi-threaded program. One problem has to do with state (for example, memory) covered by mutexes. Consider the case where one thread has a mutex locked and the state covered by that mutex is inconsistent while another thread calls `fork()`. In the child, the mutex is in the locked state (locked by a nonexistent thread and thus can never be unlocked). Having the child simply reinitialize the mutex is unsatisfactory since this approach does not resolve the question about how to correct or otherwise deal with the inconsistent state in the child.

It is suggested that programs that use `fork()` call an `exec` function very soon afterwards in the child process, thus resetting all states. In the meantime, only a short list of async-signal-safe library routines are promised to be available.

Unfortunately, this solution does not address the needs of multi-threaded libraries. Application programs may not be aware that a multi-threaded library is in use, and they feel free to call any number of library routines between the `fork()` and `exec` calls, just as they always have. Indeed, they may be extant single-threaded programs and cannot, therefore, be expected to obey new restrictions imposed by the threads library.

On the other hand, the multi-threaded library needs a way to protect its internal state during `fork()` in case it is re-entered later in the child process. The problem arises especially in multi-threaded I/O libraries, which are almost sure to be invoked between the `fork()` and `exec` calls to effect I/O redirection. The solution may require locking mutex variables during `fork()`, or it may entail simply resetting the state in the child after the `fork()` processing completes.

The `pthread_atfork()` function was intended to provide multi-threaded libraries with a means to protect themselves from innocent application programs that call `fork()`, and to provide multi-threaded application programs with a standard mechanism for protecting themselves from `fork()` calls in a library routine or the application itself.

The expected usage was that the prepare handler would acquire all mutex locks and the other two fork handlers would release them.

For example, an application could have supplied a prepare routine that acquires the necessary mutexes the library maintains and supplied child and parent routines that release those mutexes, thus ensuring that the child would have got a consistent snapshot of the state of the library (and that no mutexes would have been left stranded). This is good in theory, but in reality not practical. Each and every mutex and lock in the process must be located and locked. Every component of a program including third-party components must participate and they must agree who is responsible for which mutex or lock. This is especially problematic for mutexes and locks in dynamically allocated memory. All mutexes and locks internal to the implementation must be locked, too. This possibly delays the thread calling `fork()` for a long time or even indefinitely since uses of these synchronization objects may not be under control of the application. A final problem to mention here is the problem of locking streams. At least the streams under control of the system (like `stdin`, `stdout`, `stderr`) must be protected by locking the stream with `flockfile()`. But the application itself could have done that, possibly in the same thread calling `fork()`. In this case, the process will deadlock.

Alternatively, some libraries might have been able to supply just a `child` routine that reinitializes the mutexes in the library and all associated states to some known value (for example, what it was when the image was originally executed). This approach is not possible, though, because implementations are allowed to fail `*_init_()` and `*_destroy_()` calls for mutexes and locks if the mutex or lock is still locked. In this case, the `child` routine is not able to reinitialize the mutexes and locks.

When `fork()` is called, only the calling thread is duplicated in the child process. Synchronization variables remain in the same state in the child as they were in the parent at the time `fork()` was called. Thus, for example, mutex locks may be held by threads that no longer exist in the child process, and any associated states may be inconsistent. The intention was that the parent process could have avoided this by explicit code that acquires and releases locks critical to the child via `pthread_atfork()`. In addition, any critical threads would have needed to be recreated and reinitialized to the proper state in the child (also via `pthread_atfork()`).

A higher-level package may acquire locks on its own data structures before invoking lower-level packages. Under this scenario, the order specified for fork handler calls allows a simple rule of initialization for avoiding package deadlock: a package initializes all packages on which it depends before it calls the `pthread_atfork()` function for itself.

As explained, there is no suitable solution for functionality which requires non-atomic operations to be protected through mutexes and locks. This is why the POSIX.1 standard since the 1996 release requires that the child process after `fork()` in a multi-threaded process only calls async-signal-safe interfaces.

An additional problem arises when `pthread_atfork()` is called to register a function in a library that was loaded using `dlopen()`. If the library is unloaded using `dlclose()`, and the implementation of `dlclose()` does not unregister the function, then when `fork()` tries to call it the result will be undefined behavior. Some implementations of `dlclose()` do unregister `pthread_atfork()` handlers, but this cannot be relied upon by portable applications. The standard provides no portable method for unregistering a function installed as a handler via `pthread_atfork()`.

## See Also

- `exec`
- `fork`
- `flockfile`
- `dlopen`
- `dlclose`

## 1.106. pthread\_attr\_destroy

---

### Synopsis

```
#include <pthread.h>

int pthread_attr_destroy(pthread_attr_t *attr);
int pthread_attr_init(pthread_attr_t *attr);
```

### Description

The `pthread_attr_destroy()` function shall destroy a thread attributes object. An implementation may cause `pthread_attr_destroy()` to set `attr` to an implementation-defined invalid value. A destroyed `attr` attributes object can be reinitialized using `pthread_attr_init()`; the results of otherwise referencing the object after it has been destroyed are undefined.

The `pthread_attr_init()` function shall initialize a thread attributes object `attr` with the default value for all of the individual attributes used by a given implementation.

The resulting attributes object (possibly modified by setting individual attribute values) when used by `pthread_create()` defines the attributes of the thread created. A single attributes object can be used in multiple simultaneous calls to `pthread_create()`. Results are undefined if `pthread_attr_init()` is called specifying an already initialized `attr` attributes object.

The behavior is undefined if the value specified by the `attr` argument to `pthread_attr_destroy()` does not refer to an initialized thread attributes object.

### Return Value

Upon successful completion, these functions shall return zero; otherwise, an error number shall be returned to indicate the error.

### Errors

These functions may fail if:

- **EINVAL** - The value specified by `attr` does not refer to an initialized thread attributes object (for `pthread_attr_destroy()` ).
- **EBUSY** - The value specified by `attr` refers to an already initialized thread attributes object (for `pthread_attr_init()` ).

These functions shall not return an error code of `[EINTR]` .

## Examples

No examples are provided.

## Application Usage

Attributes objects are provided for threads, mutexes, and condition variables as a mechanism to support probable future standardization in these areas without requiring that the function itself be changed.

Attributes objects provide clean isolation of the configurable aspects of threads. For example, "stack size" is an important attribute of a thread, but it cannot be expressed portably. When porting a threaded program, stack sizes often need to be adjusted. The use of attributes objects can help by allowing the changes to be isolated in a single place, rather than being spread across every instance of thread creation.

Attributes objects can be used to set up "classes" of threads with similar attributes; for example, "threads with large stacks and high priority" or "threads with minimal stacks". These classes can be defined in a single place and then referenced wherever threads need to be created. Changes to "class" decisions become straightforward, and detailed analysis of each `pthread_create()` call is not required.

The attributes objects are defined as opaque types as an aid to extensibility. If these objects had been specified as structures, adding new attributes would force recompilation of all multi-threaded programs when the attributes objects are extended; this might not be possible if different program components were supplied by different vendors.

Additionally, opaque attributes objects present opportunities for improving performance. Argument validity can be checked once when attributes are set, rather than each time a thread is created. Implementations often need to cache kernel objects that are expensive to create. Opaque attributes objects provide an efficient mechanism to detect when cached objects become invalid due to attribute changes.

Since assignment is not necessarily defined on a given opaque type, implementation-defined default values cannot be defined in a portable way. The solution to this problem is to allow attributes objects to be initialized dynamically by attributes object initialization functions, so that default values can be supplied automatically by the implementation.

## Rationale

The following proposal was provided as a suggested alternative to the supplied attributes:

1. Maintain the style of passing a parameter formed by the bitwise-inclusive OR of flags to the initialization routines (`pthread_create()`, `pthread_mutex_init()`, `pthread_cond_init()`). The parameter containing the flags should be an opaque type for extensibility. If no flags are set in the parameter, then the objects are created with default characteristics. An implementation may specify implementation-defined flag values and associated behavior.
2. If further specialization of mutexes and condition variables is necessary, implementations may specify additional procedures that operate on the `pthread_mutex_t` and `pthread_cond_t` objects (instead of on attributes objects).

The difficulties with this solution are:

1. A bitmask is not opaque if bits have to be set into bitvector attributes objects using explicitly-coded bitwise-inclusive OR operations. If the set of options exceeds an `int`, application programmers need to know the location of each bit. If bits are set or read by encapsulation (that is, get and set functions), then the bitmask is merely an implementation of attributes objects as currently defined and should not be exposed to the programmer.
2. Many attributes are not Boolean or very small integral values. For example, scheduling policy may be placed in 3-bit or 4-bit, but priority requires 5-bit or more, thereby taking up at least 8 bits out of a possible 16 bits on machines with 16-bit integers. Because of this, the bitmask can only reasonably control whether particular attributes are set or not, and it cannot serve as the repository of the value itself. The value needs to be specified as a function parameter (which is non-extensible), or by setting a structure field (which is non-opaque), or by get and set functions (making the bitmask a redundant addition to the attributes objects).

Stack size is defined as an optional attribute because the very notion of a stack is inherently machine-dependent. Some implementations may not be able to change the size of the stack, for example, and others may not need to because stack pages may be discontiguous and can be allocated and released on demand.

The attribute mechanism has been designed in large measure for extensibility. Future extensions to the attribute mechanism or to any attributes object defined in this volume of POSIX.1-2024 has to be done with care so as not to affect binary-compatibility.

Attributes objects, even if allocated by means of dynamic allocation functions such as `malloc()`, may have their size fixed at compile time. This means, for example, a `pthread_create()` in an implementation with extensions to `pthread_attr_t` cannot look beyond the area that the binary application assumes is valid. This suggests that implementations should maintain a size field in the attributes object, as well as possibly version information, if extensions in different directions (possibly by different vendors) are to be accommodated.

If an implementation detects that the value specified by the `attr` argument to `pthread_attr_destroy()` does not refer to an initialized thread attributes object, it is recommended that the function should fail and report an `[EINVAL]` error.

If an implementation detects that the value specified by the `attr` argument to `pthread_attr_init()` refers to an already initialized thread attributes object, it is recommended that the function should fail and report an `[EBUSY]` error.

## Future Directions

None.

## See Also

`pthread_create()`, `malloc()`

The Shell and Utilities volume of POSIX.1-2024, Section 2.9.2 on the XSH System Interfaces Option

## Copyright

Portions of this text are reprinted and reproduced in electronic form from IEEE Std 1003.1-2024, Standard for Information Technology -- Portable Operating System Interface (POSIX), The Open Group Base Specifications Issue 7, Copyright (C)

2024 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group. In the event of any discrepancy between this version and the original IEEE and The Open Group Standard, the original IEEE and The Open Group Standard shall prevail. The original Standard can be obtained online at <https://pubs.opengroup.org/onlinepubs/9799919799/>.

## 1.107. `pthread_attr_getdetachstate`, `pthread_attr_setdetachstate` — get and set the detachstate attribute

### SYNOPSIS

```
#include <pthread.h>

int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate);
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

### DESCRIPTION

The `detachstate` attribute controls whether the thread is created in a detached state. If the thread is created detached, then use of the ID of the newly created thread by the `pthread_detach()` or `pthread_join()` function is an error.

The `pthread_attr_getdetachstate()` and `pthread_attr_setdetachstate()` functions, respectively, shall get and set the `detachstate` attribute in the `attr` object.

For `pthread_attr_getdetachstate()`, `detachstate` shall be set to either `PTHREAD_CREATE_DETACHED` or `PTHREAD_CREATE_JOINABLE`.

For `pthread_attr_setdetachstate()`, the application shall set `detachstate` to either `PTHREAD_CREATE_DETACHED` or `PTHREAD_CREATE_JOINABLE`.

A value of `PTHREAD_CREATE_DETACHED` shall cause all threads created with `attr` to be in the detached state, whereas using a value of `PTHREAD_CREATE_JOINABLE` shall cause all threads created with `attr` to be in the joinable state. The default value of the `detachstate` attribute shall be `PTHREAD_CREATE_JOINABLE`.

The behavior is undefined if the value specified by the `attr` argument to `pthread_attr_getdetachstate()` or `pthread_attr_setdetachstate()` does not refer to an initialized thread attributes object.

### RETURN VALUE

Upon successful completion, `pthread_attr_getdetachstate()` and `pthread_attr_setdetachstate()` shall return a value of 0; otherwise, an error

number shall be returned to indicate the error.

The `pthread_attr_getdetachstate()` function stores the value of the `detachstate` attribute in `detachstate` if successful.

## ERRORS

The `pthread_attr_setdetachstate()` function shall fail if:

- `[EINVAL]`
- The value of `detachstate` was not valid

These functions shall not return an error code of `[EINTR]`.

---

*The following sections are informative.*

## EXAMPLES

### Retrieving the detachstate Attribute

This example shows how to obtain the `detachstate` attribute of a thread attribute object.

```
#include <pthread.h>

pthread_attr_t thread_attr;
int          detachstate;
int          rc;

/* code initializing thread_attr */
...

rc = pthread_attr_getdetachstate(&thread_attr, &detachstate);
if (rc != 0) {
    /* handle error */
    ...
}
else {
    /* legal values for detachstate are:
     * PTHREAD_CREATE_DETACHED or PTHREAD_CREATE_JOINABLE
     */
    ...
}
```

# APPLICATION USAGE

None.

## RATIONALE

If an implementation detects that the value specified by the `attr` argument to `pthread_attr_getdetachstate()` or `pthread_attr_setdetachstate()` does not refer to an initialized thread attributes object, it is recommended that the function should fail and report an `[EINVAL]` error.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_attr_destroy()`
- `pthread_attr_getstacksize()`
- `pthread_create()`
- `<pthread.h>`

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

### Issue 6

The `pthread_attr_setdetachstate()` and `pthread_attr_getdetachstate()` functions are marked as part of the Threads option.

The normative text is updated to avoid use of the term "must" for application requirements.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/72 is applied, adding the example to the EXAMPLES section.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/73 is applied, updating the ERRORS section to include the optional **[EINVAL]** error.

## Issue 7

The `pthread_attr_setdetachstate()` and `pthread_attr_getdetachstate()` functions are moved from the Threads option to the Base.

The **[EINVAL]** error for an uninitialized thread attributes object is removed; this condition results in undefined behavior.

## 1.108. `pthread_attr_getguardsize`, `pthread_attr_setguardsize` — get and set the thread guardsize attribute

### SYNOPSIS

```
#include <pthread.h>

int pthread_attr_getguardsize(const pthread_attr_t *restrict attr,
                               size_t *restrict guardsize);

int pthread_attr_setguardsize(pthread_attr_t *attr,
                               size_t guardsize);
```

### DESCRIPTION

The `pthread_attr_getguardsize()` function shall get the `guardsize` attribute in the `attr` object. This attribute shall be returned in the `guardsize` parameter.

The `pthread_attr_setguardsize()` function shall set the `guardsize` attribute in the `attr` object. The new value of this attribute shall be obtained from the `guardsize` parameter. If `guardsize` is zero, a guard area shall not be provided for threads created with `attr`. If `guardsize` is greater than zero, a guard area of at least size `guardsize` bytes shall be provided for each thread created with `attr`.

The `guardsize` attribute controls the size of the guard area for the created thread's stack. The `guardsize` attribute provides protection against overflow of the stack pointer. If a thread's stack is created with guard protection, the implementation allocates extra memory at the overflow end of the stack as a buffer against stack overflow of the stack pointer. If an application overflows into this buffer an error shall result (possibly in a `SIGSEGV` signal being delivered to the thread).

A conforming implementation may round up the value contained in `guardsize` to a multiple of the configurable system variable `{PAGESIZE}` (see `<sys/mman.h>`). If an implementation rounds up the value of `guardsize` to a multiple of `{PAGESIZE}`, a call to `pthread_attr_getguardsize()` specifying `attr` shall store in the `guardsize` parameter the guard size specified by the previous `pthread_attr_setguardsize()` function call.

The default value of the `guardsize` attribute is implementation-defined.

If the `stackaddr` attribute has been set (that is, the caller is allocating and managing its own thread stacks), the `guardsize` attribute shall be ignored and no protection shall be provided by the implementation. It is the responsibility of the application to manage stack overflow along with stack allocation and management in this case.

The behavior is undefined if the value specified by the `attr` argument to `pthread_attr_getguardsize()` or `pthread_attr_setguardsize()` does not refer to an initialized thread attributes object.

## RETURN VALUE

If successful, the `pthread_attr_getguardsize()` and `pthread_attr_setguardsize()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

## ERRORS

These functions shall fail if:

- **EINVAL**
- The parameter `guardsize` is invalid.

These functions shall not return an error code of `[EINTR]`.

## EXAMPLES

### Retrieving the guardsize Attribute

This example shows how to obtain the `guardsize` attribute of a thread attribute object.

```
#include <pthread.h>

pthread_attr_t thread_attr;
size_t guardsize;
int rc;

/* code initializing thread_attr */
...

rc = pthread_attr_getguardsize(&thread_attr, &guardsize);
```

```
if (rc != 0) {
    /* handle error */
    ...
}

else {
    if (guardsize > 0) {
        /* a guard area of at least guardsize bytes is provided */
        ...
    }
    else {
        /* no guard area provided */
        ...
    }
}
```

## APPLICATION USAGE

None.

## RATIONALE

The `guardsize` attribute is provided to the application for two reasons. ▶

1. Overflow protection can potentially result in wasted system resources. An application that creates a large number of threads, and which knows its threads never overflow their stack, can save system resources by turning off guard areas.
2. When threads allocate large data structures on the stack, large guard areas may be needed to detect stack overflow.

The default size of the guard area is left implementation-defined since on systems supporting very large page sizes, the overhead might be substantial if at least one guard page is required by default.

If an implementation detects that the value specified by the `attr` argument to `pthread_attr_getguardsize()` or `pthread_attr_setguardsize()` does not refer to an initialized thread attributes object, it is recommended that the function should fail and report an `[EINVAL]` error.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `<pthread.h>`
- `<sys/mman.h>`

## CHANGE HISTORY

First released in Issue 5.

### Issue 6

- In the ERRORS section, a third `[EINVAL]` error condition is removed as it is covered by the second error condition.
- The `restrict` keyword is added to the `pthread_attr_getguardsize()` prototype for alignment with the ISO/IEC 9899:1999 standard.
- IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/74 is applied, updating the ERRORS section to remove the `[EINVAL]` error ("The attribute `attr` is invalid."), and replacing it with the optional `[EINVAL]` error.
- IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/76 is applied, adding the example to the EXAMPLES section.

### Issue 7

- SD5-XSH-ERN-111 is applied, removing the reference to the `stack` attribute in the DESCRIPTION.
- SD5-XSH-ERN-175 is applied, updating the DESCRIPTION to note that the default size of the guard area is implementation-defined.
- The `pthread_attr_getguardsize()` and `pthread_attr_setguardsize()` functions are moved from the XSI option to the Base.
- The `[EINVAL]` error for an uninitialized thread attributes object is removed; this condition results in undefined behavior.

---

## 1.109. `pthread_attr_getinheritsched`, `pthread_attr_setinheritsched` — get and set the inheritsched attribute (REALTIME THREADS)

### SYNOPSIS

```
#include <pthread.h>

int pthread_attr_getinheritsched(const pthread_attr_t *restrict attr,
                                 int *restrict inheritsched);

int pthread_attr_setinheritsched(pthread_attr_t *attr,
                                 int inheritsched);
```

### DESCRIPTION

The `pthread_attr_getinheritsched()` and `pthread_attr_setinheritsched()` functions, respectively, shall get and set the `inheritsched` attribute in the `attr` argument.

When the attributes objects are used by `pthread_create()`, the `inheritsched` attribute determines how the other scheduling attributes of the created thread shall be set.

The supported values of `inheritsched` shall be:

- `PTHREAD_INHERIT_SCHED`
- Specifies that the thread scheduling attributes shall be inherited from the creating thread, and the scheduling attributes in this `attr` argument shall be ignored.
- `PTHREAD_EXPLICIT_SCHED`
- Specifies that the thread scheduling attributes shall be set to the corresponding values from this attributes object.

The symbols `PTHREAD_INHERIT_SCHED` and `PTHREAD_EXPLICIT_SCHED` are defined in the `<pthread.h>` header.

The following thread scheduling attributes defined by POSIX.1-2024 are affected by the `inheritsched` attribute: scheduling policy (`schedpolicy`), scheduling

parameters (`schedparam`), and scheduling contention scope (`contentionscope`).

The behavior is undefined if the value specified by the `attr` argument to `pthread_attr_getinheritsched()` or `pthread_attr_setinheritsched()` does not refer to an initialized thread attributes object.

## RETURN VALUE

If successful, the `pthread_attr_getinheritsched()` and `pthread_attr_setinheritsched()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

## ERRORS

The `pthread_attr_setinheritsched()` function shall fail if:

- **[ENOTSUP]**
- An attempt was made to set the attribute to an unsupported value.

The `pthread_attr_setinheritsched()` function may fail if:

- **[EINVAL]**
- The value of `inheritsched` is not valid.

These functions shall not return an error code of **[EINTR]**.

## EXAMPLES

None.

## APPLICATION USAGE

After these attributes have been set, a thread can be created with the specified attributes using `pthread_create()`. Using these routines does not affect the current running thread.

See [2.9.4 Thread Scheduling](#) for further details on thread scheduling attributes and their default settings.

## RATIONALE

If an implementation detects that the value specified by the `attr` argument to `pthread_attr_getinheritsched()` or `pthread_attr_setinheritsched()` does not refer to an initialized thread attributes object, it is recommended that the function should fail and report an `[EINVAL]` error.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_attr_destroy()`
- `pthread_attr_getscope()`
- `pthread_attr_getschedpolicy()`
- `pthread_attr_getschedparam()`
- `pthread_create()`
- XBD `<pthread.h>`
- XBD `<sched.h>`

## CHANGE HISTORY

**First released in Issue 5. Included for alignment with the POSIX Threads Extension.**

Marked as part of the Realtime Threads Feature Group.

### Issue 6

The `pthread_attr_getinheritsched()` and `pthread_attr_setinheritsched()` functions are marked as part of the Threads and Thread Execution Scheduling options.

The `[ENOSYS]` error condition has been removed as stubs need not be provided if an implementation does not support the Thread Execution Scheduling option.

The `restrict` keyword is added to the `pthread_attr_getinheritsched()` prototype for alignment with the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/75 is applied, clarifying the values of `inheritsched` in the DESCRIPTION and adding two optional `[EINVAL]` errors to the ERRORS section for checking when `attr` refers to an uninitialized thread attribute object.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/77 is applied, adding a reference to [2.9.4 Thread Scheduling](#) in the APPLICATION USAGE section.

## Issue 7

The `pthread_attr_getinheritsched()` and `pthread_attr_setinheritsched()` functions are marked only as part of the Thread Execution Scheduling option as the Threads option is now part of the Base.

The `[EINVAL]` error for an uninitialized thread attributes object is removed; this condition results in undefined behavior.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0450 [314] is applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0262 [757] is applied.

## 1.110. pthread\_attr\_getschedparam, pthread\_attr\_setschedparam

---

### SYNOPSIS

```
#include <pthread.h>

int pthread_attr_getschedparam(const pthread_attr_t *restrict attr,
                               struct sched_param *restrict param);

int pthread_attr_setschedparam(pthread_attr_t *restrict attr,
                               const struct sched_param *restrict pa
```

### DESCRIPTION

The `pthread_attr_getschedparam()` and `pthread_attr_setschedparam()` functions, respectively, shall get and set the scheduling parameter attributes in the `attr` argument. The contents of the `param` structure are defined in the `<sched.h>` header. For the SCHED\_FIFO and SCHED\_RR policies, the only required member of `param` is `sched_priority`.

[TSP] For the SCHED\_SPORADIC policy, the required members of the `param` structure are `sched_priority`, `sched_ss_low_priority`, `sched_ss_repl_period`, `sched_ss_init_budget`, and `sched_ss_max_repl`. The specified `sched_ss_repl_period` needs to be greater than or equal to the specified `sched_ss_init_budget` for the function to succeed; if it is not, then the function shall fail. The value of `sched_ss_max_repl` shall be within the inclusive range  $[1, \{SS\_REPL\_MAX\}]$  for the function to succeed; if not, the function shall fail. It is unspecified whether the `sched_ss_repl_period` and `sched_ss_init_budget` values are stored as provided by this function or are rounded to align with the resolution of the clock being used.

The behavior is undefined if the value specified by the `attr` argument to `pthread_attr_getschedparam()` or `pthread_attr_setschedparam()` does not refer to an initialized thread attributes object.

## RETURN VALUE

If successful, the `pthread_attr_getschedparam()` and `pthread_attr_setschedparam()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

## ERRORS

The `pthread_attr_setschedparam()` function shall fail if:

- **ENOTSUP**
- An attempt was made to set the attribute to an unsupported value.

The `pthread_attr_setschedparam()` function may fail if:

- **EINVAL**
- The value of `param` is not valid.

These functions shall not return an error code of [EINTR].

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

After these attributes have been set, a thread can be created with the specified attributes using `pthread_create()`. Using these routines does not affect the current running thread.

## RATIONALE

If an implementation detects that the value specified by the `attr` argument to `pthread_attr_getschedparam()` or `pthread_attr_setschedparam()` does not refer to an initialized thread attributes object, it is recommended that the function should fail and report an [EINVAL] error.

# FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_attr_destroy()`
- `pthread_attr_getscope()`
- `pthread_attr_getinheritsched()`
- `pthread_attr_getschedpolicy()`
- `pthread_create()`

XBD `<pthread.h>`, `<sched.h>`

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

### Issue 6

The `pthread_attr_getschedparam()` and `pthread_attr_setschedparam()` functions are marked as part of the Threads option.

The SCHED\_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.

The `restrict` keyword is added to the `pthread_attr_getschedparam()` and `pthread_attr_setschedparam()` prototypes for alignment with the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/78 is applied, updating the ERRORS section to include optional errors for the case when `attr` refers to an uninitialized thread attribute object.

### Issue 7

The `pthread_attr_getschedparam()` and `pthread_attr_setschedparam()` functions are moved from the Threads option to the Base.

Austin Group Interpretation 1003.1-2001 #119 is applied, clarifying the accuracy requirements for the `sched_ss_repl_period` and `sched_ss_init_budget`

values.

The [EINVAL] error for an uninitialized thread attributes object is removed; this condition results in undefined behavior.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0451 [314] is applied.

## 1.111. `pthread_attr_getschedpolicy`, `pthread_attr_setschedpolicy`

### SYNOPSIS

```
#include <pthread.h>

int pthread_attr_getschedpolicy(const pthread_attr_t *restrict attr,
                                int *restrict policy);

int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
```

### DESCRIPTION

The `pthread_attr_getschedpolicy()` and `pthread_attr_setschedpolicy()` functions, respectively, shall get and set the schedpolicy attribute in the attr argument.

The supported values of policy shall include SCHED\_FIFO, SCHED\_RR, and SCHED\_OTHER, which are defined in the `<sched.h>` header. When threads executing with the scheduling policy SCHED\_FIFO, SCHED\_RR, or SCHED\_SPORADIC are waiting on a mutex, they shall acquire the mutex in priority order when the mutex is unlocked.

The behavior is undefined if the value specified by the attr argument to `pthread_attr_getschedpolicy()` or `pthread_attr_setschedpolicy()` does not refer to an initialized thread attributes object.

### RETURN VALUE

If successful, the `pthread_attr_getschedpolicy()` and `pthread_attr_setschedpolicy()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

### ERRORS

The `pthread_attr_setschedpolicy()` function shall fail if:

- **ENOTSUP**

- An attempt was made to set the attribute to an unsupported value.

The `pthread_attr_setschedpolicy()` function may fail if:

- **EINVAL**
- The value of policy is not valid.

These functions shall not return an error code of [EINTR].

## EXAMPLES

None.

## APPLICATION USAGE

After these attributes have been set, a thread can be created with the specified attributes using `pthread_create()`. Using these routines does not affect the current running thread.

See [2.9.4 Thread Scheduling](#) for further details on thread scheduling attributes and their default settings.

## RATIONALE

If an implementation detects that the value specified by the attr argument to `pthread_attr_getschedpolicy()` or `pthread_attr_setschedpolicy()` does not refer to an initialized thread attributes object, it is recommended that the function should fail and report an [EINVAL] error.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_attr_destroy()`
- `pthread_attr_getscope()`
- `pthread_attr_getinheritsched()`
- `pthread_attr_getschedparam()`

- `pthread_create()`

XBD `<pthread.h>`, `<sched.h>`

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Marked as part of the Realtime Threads Feature Group.

### Issue 6

The `pthread_attr_getschedpolicy()` and `pthread_attr_setschedpolicy()` functions are marked as part of the Threads and Thread Execution Scheduling options.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Thread Execution Scheduling option.

The SCHED\_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.

The `restrict` keyword is added to the `pthread_attr_getschedpolicy()` prototype for alignment with the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/79 is applied, adding a reference to [2.9.4 Thread Scheduling](#) in the APPLICATION USAGE section.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/80 is applied, updating the ERRORS section to include optional errors for the case when attr refers to an uninitialized thread attribute object.

### Issue 7

The `pthread_attr_getschedpolicy()` and `pthread_attr_setschedpolicy()` functions are marked only as part of the Thread Execution Scheduling option as the Threads option is now part of the Base.

The [EINVAL] error for an uninitialized thread attributes object is removed; this condition results in undefined behavior.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0452 [314] is applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0263 [757] is applied.

## 1.112. pthread\_attr\_getscope, pthread\_attr\_setscope

### SYNOPSIS

```
#include <pthread.h>

int pthread_attr_getscope(const pthread_attr_t *restrict attr,
                          int *restrict contentionscope);

int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);
```

### DESCRIPTION

The `pthread_attr_getscope()` and `pthread_attr_setscope()` functions, respectively, shall get and set the *contentionscope* attribute in the *attr* object.

The *contentionscope* attribute may have the values `PTHREAD_SCOPE_SYSTEM`, signifying system scheduling contention scope, or `PTHREAD_SCOPE_PROCESS`, signifying process scheduling contention scope. The symbols `PTHREAD_SCOPE_SYSTEM` and `PTHREAD_SCOPE_PROCESS` are defined in the `<pthread.h>` header.

The behavior is undefined if the value specified by the *attr* argument to `pthread_attr_getscope()` or `pthread_attr_setscope()` does not refer to an initialized thread attributes object.

### RETURN VALUE

If successful, the `pthread_attr_getscope()` and `pthread_attr_setscope()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

### ERRORS

The `pthread_attr_setscope()` function shall fail if:

- **[ENOTSUP]**

An attempt was made to set the attribute to an unsupported value.

The `pthread_attr_setscope()` function may fail if:

- **[EINVAL]**

The value of *contentionscope* is not valid.

These functions shall not return an error code of **[EINTR]**.

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

After these attributes have been set, a thread can be created with the specified attributes using `pthread_create()`. Using these routines does not affect the current running thread.

See [2.9.4 Thread Scheduling](#) for further details on thread scheduling attributes and their default settings.

## RATIONALE

If an implementation detects that the value specified by the *attr* argument to `pthread_attr_getscope()` or `pthread_attr_setscope()` does not refer to an initialized thread attributes object, it is recommended that the function should fail and report an **[EINVAL]** error.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_attr_destroy()`
- `pthread_attr_getinheritsched()`
- `pthread_attr_getschedpolicy()`
- `pthread_attr_getschedparam()`

- `pthread_create()`

XBD `<pthread.h>`, `<sched.h>`

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Marked as part of the Realtime Threads Feature Group.

### Issue 6

The `pthread_attr_getscope()` and `pthread_attr_setscope()` functions are marked as part of the Threads and Thread Execution Scheduling options.

The **[ENOSYS]** error condition has been removed as stubs need not be provided if an implementation does not support the Thread Execution Scheduling option.

The `restrict` keyword is added to the `pthread_attr_getscope()` prototype for alignment with the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/81 is applied, adding a reference to [2.9.4 Thread Scheduling](#) in the APPLICATION USAGE section.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/82 is applied, updating the ERRORS section to include optional errors for the case when *attr* refers to an uninitialized thread attribute object.

### Issue 7

The `pthread_attr_getscope()` and `pthread_attr_setscope()` functions are marked only as part of the Thread Execution Scheduling option as the Threads option is now part of the Base.

The **[EINVAL]** error for an uninitialized thread attributes object is removed; this condition results in undefined behavior.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0453 [314] is applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0264 [757] is applied.

## 1.113. pthread\_attr\_getstack, pthread\_attr\_setstack

### SYNOPSIS

```
[TSA TSS]
#include <pthread.h>

int pthread_attr_getstack(const pthread_attr_t *restrict attr,
                          void **restrict stackaddr,
                          size_t *restrict stacksize);

int pthread_attr_setstack(pthread_attr_t *attr,
                         void *stackaddr,
                         size_t stacksize);
```

### DESCRIPTION

The `pthread_attr_getstack()` and `pthread_attr_setstack()` functions, respectively, shall get and set the thread creation stack attributes `stackaddr` and `stacksize` in the `attr` object.

The stack attributes specify the area of storage to be used for the created thread's stack. The base (lowest addressable byte) of the storage shall be `stackaddr`, and the size of the storage shall be `stacksize` bytes. The `stacksize` shall be at least `{PTHREAD_STACK_MIN}`. The `pthread_attr_setstack()` function may fail with `[EINVAL]` if `stackaddr` does not meet implementation-defined alignment requirements. All pages within the stack described by `stackaddr` and `stacksize` shall be both readable and writable by the thread.

If the `pthread_attr_getstack()` function is called before the `stackaddr` attribute has been set, the behavior is unspecified.

The behavior is undefined if the value specified by the `attr` argument to `pthread_attr_getstack()` or `pthread_attr_setstack()` does not refer to an initialized thread attributes object.

### RETURN VALUE

Upon successful completion, these functions shall return a value of 0; otherwise, an error number shall be returned to indicate the error.

The `pthread_attr_getstack()` function shall store the stack attribute values in `stackaddr` and `stacksize` if successful.

# ERRORS

The `pthread_attr_setstack()` function shall fail if:

- **[EINVAL]**
- The value of `stacksize` is less than {PTHREAD\_STACK\_MIN} or exceeds an implementation-defined limit.

The `pthread_attr_setstack()` function may fail if:

- **[EINVAL]**
- The value of `stackaddr` does not have proper alignment to be used as a stack, or ((char \*)stackaddr + stacksize) lacks proper alignment.
- **[EACCES]**
- The stack page(s) described by `stackaddr` and `stacksize` are not both readable and writable by the thread.

These functions shall not return an error code of [EINTR].

---

## EXAMPLES

None.

## APPLICATION USAGE

These functions are appropriate for use by applications in an environment where the stack for a thread must be placed in some particular region of memory.

While it might seem that an application could detect stack overflow by providing a protected page outside the specified stack region, this cannot be done portably. Implementations are free to place the thread's initial stack pointer anywhere within the specified region to accommodate the machine's stack pointer behavior and allocation requirements. Furthermore, on some architectures, such as the IA-64, "overflow" might mean that two separate stack pointers allocated within the region will overlap somewhere in the middle of the region.

After a successful call to `pthread_attr_setstack()`, the storage area specified by the `stackaddr` parameter is under the control of the implementation, as described in [2.9.8 Use of Application-Managed Thread Stacks](#).

The specification of the `stackaddr` attribute presents several ambiguities that make portable use of these functions impossible. For example, the standard allows

implementations to impose arbitrary alignment requirements on `stackaddr`. Applications cannot assume that a buffer obtained from `malloc()` is suitably aligned. Note that although the `stacksize` value passed to `pthread_attr_setstack()` must satisfy alignment requirements, the same is not true for `pthread_attr_setstacksize()` where the implementation must increase the specified size if necessary to achieve the proper alignment.

## RATIONALE

If an implementation detects that the value specified by the `attr` argument to `pthread_attr_getstack()` or `pthread_attr_setstack()` does not refer to an initialized thread attributes object, it is recommended that the function should fail and report an [EINVAL] error.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_attr_destroy()`
- `pthread_attr_getdetachstate()`
- `pthread_attr_getstacksize()`
- `pthread_create()`
- XBD `<limits.h>`
- XBD `<pthread.h>`

## CHANGE HISTORY

First released in Issue 6. Developed as part of the XSI option and brought into the BASE by IEEE PASC Interpretation 1003.1 #101.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/83 is applied, updating the APPLICATION USAGE section to refer to [2.9.8 Use of Application-Managed Thread Stacks](#).

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC/D6/84 is applied, updating the ERRORS section to include optional errors for the case when `attr` refers to an uninitialized thread attribute object.

## Issue 7

SD5-XSH-ERN-66 is applied, correcting the use of `attr` in the [EINVAL] error condition.

Austin Group Interpretation 1003.1-2001 #057 is applied, clarifying the behavior if the function is called before the `stackaddr` attribute is set.

SD5-XSH-ERN-157 is applied, updating the APPLICATION USAGE section.

The description of the `stackaddr` attribute is updated in the DESCRIPTION and APPLICATION USAGE sections.

The [EINVAL] error for an uninitialized thread attributes object is removed; this condition results in undefined behavior.

## 1.114. pthread\_attr\_getstackaddr, pthread\_attr\_setstackaddr

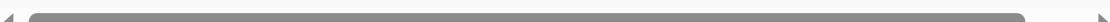
---

### SYNOPSIS

```
[0B] #include <pthread.h>

int pthread_attr_getstackaddr(const pthread_attr_t *restrict attr,
                               void **restrict stackaddr);

int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
```



### DESCRIPTION

The `pthread_attr_getstackaddr()` and `pthread_attr_setstackaddr()` functions, respectively, shall get and set the thread creation `stackaddr` attribute in the `attr` object.

The `stackaddr` attribute specifies the location of storage to be used for the created thread's stack. The size of the storage shall be at least `{PTHREAD_STACK_MIN}`.

### RETURN VALUE

Upon successful completion, `pthread_attr_getstackaddr()` and `pthread_attr_setstackaddr()` shall return a value of 0; otherwise, an error number shall be returned to indicate the error.

The `pthread_attr_getstackaddr()` function stores the `stackaddr` attribute value in `stackaddr` if successful.

### ERRORS

These functions may fail if:

- **[EINVAL]**

The value specified by `attr` does not refer to an initialized thread attribute object.

These functions shall not return an error code of [EINTR].

## APPLICATION USAGE

The specification of the `stackaddr` attribute presents several ambiguities that make portable use of these interfaces impossible. The description of the single address parameter as a "stack" does not specify a particular relationship between the address and the "stack" implied by that address. For example, the address may be taken as the low memory address of a buffer intended for use as a stack, or it may be taken as the address to be used as the initial stack pointer register value for the new thread. These two are not the same except for a machine on which the stack grows "up" from low memory to high, and on which a "push" operation first stores the value in memory and then increments the stack pointer register. Further, on a machine where the stack grows "down" from high memory to low, interpretation of the address as the "low memory" address requires a determination of the intended size of the stack. IEEE Std 1003.1-2001 has introduced the new interfaces `pthread_attr_setstack()` and `pthread_attr_getstack()` to resolve these ambiguities.

After a successful call to `pthread_attr_setstackaddr()`, the storage area specified by the `stackaddr` parameter is under the control of the implementation, as described in *Use of Application-Managed Thread Stacks*.

## SEE ALSO

- `pthread_attr_destroy()`
- `pthread_attr_getdetachstate()`
- `pthread_attr_getstack()`
- `pthread_attr_getstacksize()`
- `pthread_attr_setstack()`
- `pthread_create()`
- `<limits.h>`
- `<pthread.h>`

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

## Issue 6

The `pthread_attr_getstackaddr()` and `pthread_attr_setstackaddr()` functions are marked as part of the Threads and Thread Stack Address Attribute options.

The `restrict` keyword is added to the `pthread_attr_getstackaddr()` prototype for alignment with the ISO/IEC 9899:1999 standard.

These functions are marked obsolescent.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/85 is applied, updating the APPLICATION USAGE section to refer to *Use of Application-Managed Thread Stacks*.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/86 is applied, updating the ERRORS section to include optional errors for the case when `attr` refers to an uninitialized thread attribute object.

---

## 1.115. `pthread_attr_getstacksize`, `pthread_attr_setstacksize` — get and set the stacksize attribute

### SYNOPSIS

```
#include <pthread.h>

int pthread_attr_getstacksize(const pthread_attr_t *restrict attr,
                             size_t *restrict stacksize);

int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
```

### DESCRIPTION

The `pthread_attr_getstacksize()` and `pthread_attr_setstacksize()` functions, respectively, shall get and set the thread creation *stacksize* attribute in the *attr* object.

The *stacksize* attribute shall define the minimum stack size (in bytes) allocated for the created threads stack.

The behavior is undefined if the value specified by the *attr* argument to `pthread_attr_getstacksize()` or `pthread_attr_setstacksize()` does not refer to an initialized thread attributes object.

### RETURN VALUE

Upon successful completion, `pthread_attr_getstacksize()` and `pthread_attr_setstacksize()` shall return a value of 0; otherwise, an error number shall be returned to indicate the error.

The `pthread_attr_getstacksize()` function stores the *stacksize* attribute value in *stacksize* if successful.

### ERRORS

The `pthread_attr_setstacksize()` function shall fail if:

- **[EINVAL]**

The value of *stacksize* is less than {PTHREAD\_STACK\_MIN} or exceeds a system-imposed limit.

These functions shall not return an error code of [EINTR].

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

If an implementation detects that the value specified by the *attr* argument to `pthread_attr_getstacksize()` or `pthread_attr_setstacksize()` does not refer to an initialized thread attributes object, it is recommended that the function should fail and report an [EINVAL] error.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_attr_destroy()`
- `pthread_attr_getdetachstate()`
- `pthread_create()`
- XBD `<limits.h>`
- XBD `<pthread.h>`

# CHANGE HISTORY

## First released in Issue 5

Included for alignment with the POSIX Threads Extension.

## Issue 6

- The `pthread_attr_getstacksize()` and `pthread_attr_setstacksize()` functions are marked as part of the Threads and Thread Stack Size Attribute options.
- The **restrict** keyword is added to the `pthread_attr_getstacksize()` prototype for alignment with the ISO/IEC 9899:1999 standard.
- IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/43 is applied, correcting the margin code in the SYNOPSIS from TSA to TSS and updating the CHANGE HISTORY from "Thread Stack Address Attribute" option to "Thread Stack Size Attribute" option.
- IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/87 is applied, updating the ERRORS section to include optional errors for the case when *attr* refers to an uninitialized thread attribute object.

## Issue 7

- The `pthread_attr_getstacksize()` and `pthread_attr_setstacksize()` functions are marked only as part of the Thread Stack Size Attribute option as the Threads option is now part of the Base.
  - The [EINVAL] error for an uninitialized thread attributes object is removed; this condition results in undefined behavior.
  - POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0265 [757] is applied.
-

## 1.116. pthread\_attr\_init, pthread\_attr\_destroy - initialize and destroy thread attributes object

---

### SYNOPSIS

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

### DESCRIPTION

The `pthread_attr_destroy()` function shall destroy a thread attributes object. An implementation may cause `pthread_attr_destroy()` to set `attr` to an implementation-defined invalid value. A destroyed `attr` attributes object can be reinitialized using `pthread_attr_init()`; the results of otherwise referencing the object after it has been destroyed are undefined.

The `pthread_attr_init()` function shall initialize a thread attributes object `attr` with the default value for all of the individual attributes used by a given implementation.

The resulting attributes object (possibly modified by setting individual attribute values) when used by `pthread_create()` defines the attributes of the thread created. A single attributes object can be used in multiple simultaneous calls to `pthread_create()`. Results are undefined if `pthread_attr_init()` is called specifying an already initialized `attr` attributes object.

The behavior is undefined if the value specified by the `attr` argument to `pthread_attr_destroy()` does not refer to an initialized thread attributes object.

### RETURN VALUE

Upon successful completion, `pthread_attr_init()` and `pthread_attr_destroy()` shall return zero; otherwise, an error number shall be returned to indicate the error.

# ERRORS

The `pthread_attr_init()` and `pthread_attr_destroy()` functions shall fail if:

- **ENOMEM** - Insufficient memory exists to initialize the thread attributes object.

The `pthread_attr_init()` function may fail if:

- **EBUSY** - The value specified by `attr` refers to an already initialized thread attributes object.

The `pthread_attr_destroy()` function may fail if:

- **EINVAL** - The value specified by `attr` does not refer to an initialized thread attributes object.

These functions shall not return an error code of `EINTR`.

# EXAMPLES

None.

# APPLICATION USAGE

Attributes objects are provided for threads, mutexes, and condition variables as a mechanism to support probable future standardization in these areas without requiring that the function itself be changed.

Attributes objects provide clean isolation of the configurable aspects of threads. For example, "stack size" is an important attribute of a thread, but it cannot be expressed portably. When porting a threaded program, stack sizes often need to be adjusted. The use of attributes objects can help by allowing the changes to be isolated in a single place, rather than being spread across every instance of thread creation.

Attributes objects can be used to set up "classes" of threads with similar attributes; for example, "threads with large stacks and high priority" or "threads with minimal stacks". These classes can be defined in a single place and then referenced wherever threads need to be created. Changes to "class" decisions become straightforward, and detailed analysis of each `pthread_create()` call is not required.

## RATIONALE

The attributes objects are defined as opaque types as an aid to extensibility. If these objects had been specified as structures, adding new attributes would force recompilation of all multi-threaded programs when the attributes objects are extended; this might not be possible if different program components were supplied by different vendors.

Additionally, opaque attributes objects present opportunities for improving performance. Argument validity can be checked once when attributes are set, rather than each time a thread is created. Implementations often need to cache kernel objects that are expensive to create. Opaque attributes objects provide an efficient mechanism to detect when cached objects become invalid due to attribute changes.

Since assignment is not necessarily defined on a given opaque type, implementation-defined default values cannot be defined in a portable way. The solution to this problem is to allow attributes objects to be initialized dynamically by attributes object initialization functions, so that default values can be supplied automatically by the implementation.

Stack size is defined as an optional attribute because the very notion of a stack is inherently machine-dependent. Some implementations may not be able to change the size of the stack, for example, and others may not need to because stack pages may be discontiguous and can be allocated and released on demand.

The attribute mechanism has been designed in large measure for extensibility. Future extensions to the attribute mechanism or to any attributes object defined in this volume of POSIX.1-2024 has to be done with care so as not to affect binary-compatibility.

Attributes objects, even if allocated by means of dynamic allocation functions such as `malloc()`, may have their size fixed at compile time. This means, for example, a `pthread_create()` in an implementation with extensions to `pthread_attr_t` cannot look beyond the area that the binary application assumes is valid. This suggests that implementations should maintain a size field in the attributes object, as well as possibly version information, if extensions in different directions (possibly by different vendors) are to be accommodated.

If an implementation detects that the value specified by the `attr` argument to `pthread_attr_destroy()` does not refer to an initialized thread attributes object, it is recommended that the function should fail and report an `[EINVAL]` error.

If an implementation detects that the value specified by the `attr` argument to `pthread_attr_init()` refers to an already initialized thread attributes object, it is recommended that the function should fail and report an `[EBUSY]` error.

## FUTURE DIRECTIONS

None.

## SEE ALSO

`pthread_create()` , `malloc()`

The Base Definitions volume of POSIX.1-2024, `<pthread.h>`

## CHANGE HISTORY

First released in Issue 5. Included in Issue 6 (IEEE Std 1003.1, 2004).

---

## 1.117. `pthread_attr_getdetachstate`, `pthread_attr_setdetachstate` — get and set the detachstate attribute

### SYNOPSIS

```
#include <pthread.h>

int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate);
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

### DESCRIPTION

The `detachstate` attribute controls whether the thread is created in a detached state. If the thread is created detached, then use of the ID of the newly created thread by the `pthread_detach()` or `pthread_join()` function is an error.

The `pthread_attr_getdetachstate()` and `pthread_attr_setdetachstate()` functions, respectively, shall get and set the `detachstate` attribute in the `attr` object.

For `pthread_attr_getdetachstate()`, `detachstate` shall be set to either `PTHREAD_CREATE_DETACHED` or `PTHREAD_CREATE_JOINABLE`.

For `pthread_attr_setdetachstate()`, the application shall set `detachstate` to either `PTHREAD_CREATE_DETACHED` or `PTHREAD_CREATE_JOINABLE`.

A value of `PTHREAD_CREATE_DETACHED` shall cause all threads created with `attr` to be in the detached state, whereas using a value of `PTHREAD_CREATE_JOINABLE` shall cause all threads created with `attr` to be in the joinable state. The default value of the `detachstate` attribute shall be `PTHREAD_CREATE_JOINABLE`.

The behavior is undefined if the value specified by the `attr` argument to `pthread_attr_getdetachstate()` or `pthread_attr_setdetachstate()` does not refer to an initialized thread attributes object.

### RETURN VALUE

Upon successful completion, `pthread_attr_getdetachstate()` and `pthread_attr_setdetachstate()` shall return a value of 0; otherwise, an error

number shall be returned to indicate the error.

The `pthread_attr_getdetachstate()` function stores the value of the `detachstate` attribute in `detachstate` if successful.

## ERRORS

The `pthread_attr_setdetachstate()` function shall fail if:

- `[EINVAL]`
- The value of `detachstate` was not valid

These functions shall not return an error code of `[EINTR]`.

## EXAMPLES

### Retrieving the detachstate Attribute

This example shows how to obtain the `detachstate` attribute of a thread attribute object.

```
#include <pthread.h>

pthread_attr_t thread_attr;
int           detachstate;
int           rc;

/* code initializing thread_attr */
...

rc = pthread_attr_getdetachstate(&thread_attr, &detachstate);
if (rc != 0) {
    /* handle error */
    ...
} else {
    /* legal values for detachstate are:
     * PTHREAD_CREATE_DETACHED or PTHREAD_CREATE_JOINABLE
     */
    ...
}
```

## APPLICATION USAGE

None.

## RATIONALE

If an implementation detects that the value specified by the `attr` argument to `pthread_attr_getdetachstate()` or `pthread_attr_setdetachstate()` does not refer to an initialized thread attributes object, it is recommended that the function should fail and report an [EINVAL] error.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_attr_destroy()`
- `pthread_attr_getstacksize()`
- `pthread_create()`
- `<pthread.h>`

## CHANGE HISTORY

### First released in Issue 5

Included for alignment with the POSIX Threads Extension.

### Issue 6

- The `pthread_attr_setdetachstate()` and `pthread_attr_getdetachstate()` functions are marked as part of the Threads option.
- The normative text is updated to avoid use of the term "must" for application requirements.
- IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/72 is applied, adding the example to the EXAMPLES section.
- IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/73 is applied, updating the ERRORS section to include the optional [EINVAL] error.

## Issue 7

- The `pthread_attr_setdetachstate()` and `pthread_attr_getdetachstate()` functions are moved from the Threads option to the Base.
  - The [EINVAL] error for an uninitialized thread attributes object is removed; this condition results in undefined behavior.
-

## 1.118. pthread\_attr\_getguardsize

### SYNOPSIS

```
#include <pthread.h>

int pthread_attr_getguardsize(const pthread_attr_t *restrict attr,
                               size_t *restrict guardsize);
int pthread_attr_setguardsize(pthread_attr_t *attr,
                               size_t guardsize);
```

### DESCRIPTION

The `pthread_attr_getguardsize()` function shall get the `guardsize` attribute in the `attr` object. This attribute shall be returned in the `guardsize` parameter.

The `pthread_attr_setguardsize()` function shall set the `guardsize` attribute in the `attr` object. The new value of this attribute shall be obtained from the `guardsize` parameter. If `guardsize` is zero, a guard area shall not be provided for threads created with `attr`. If `guardsize` is greater than zero, a guard area of at least size `guardsize` bytes shall be provided for each thread created with `attr`.

The `guardsize` attribute controls the size of the guard area for the created thread's stack. The `guardsize` attribute provides protection against overflow of the stack pointer. If a thread's stack is created with guard protection, the implementation allocates extra memory at the overflow end of the stack as a buffer against stack overflow of the stack pointer. If an application overflows into this buffer an error shall result (possibly in a SIGSEGV signal being delivered to the thread).

A conforming implementation may round up the value contained in `guardsize` to a multiple of the configurable system variable {PAGESIZE} (see `<sys/mman.h>`). If an implementation rounds up the value of `guardsize` to a multiple of {PAGESIZE}, a call to `pthread_attr_getguardsize()` specifying `attr` shall store in the `guardsize` parameter the guard size specified by the previous `pthread_attr_setguardsize()` function call.

The default value of the `guardsize` attribute is implementation-defined.

If the `stackaddr` attribute has been set (that is, the caller is allocating and managing its own thread stacks), the `guardsize` attribute shall be ignored and

no protection shall be provided by the implementation. It is the responsibility of the application to manage stack overflow along with stack allocation and management in this case.

The behavior is undefined if the value specified by the `attr` argument to `pthread_attr_getguardsize()` or `pthread_attr_setguardsize()` does not refer to an initialized thread attributes object.

## RETURN VALUE

If successful, the `pthread_attr_getguardsize()` and `pthread_attr_setguardsize()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

## ERRORS

These functions shall fail if:

- **[EINVAL]** - The parameter `guardsize` is invalid.

These functions shall not return an error code of [EINTR].

---

*The following sections are informative.*

## EXAMPLES

### Retrieving the guardsize Attribute

This example shows how to obtain the `guardsize` attribute of a thread attribute object.

```
#include <pthread.h>

pthread_attr_t thread_attr;
size_t guardsize;
int rc;

/* code initializing thread_attr */
...

rc = pthread_attr_getguardsize (&thread_attr, &guardsize);
if (rc != 0) {
```

```
/* handle error */
...
}

else {
    if (guardsize > 0) {
        /* a guard area of at least guardsize bytes is provided */
        ...
    }
    else {
        /* no guard area provided */
        ...
    }
}
```

## APPLICATION USAGE

None.

## RATIONALE

The `guardsize` attribute is provided to the application for two reasons:

1. Overflow protection can potentially result in wasted system resources. An application that creates a large number of threads, and which knows its threads never overflow their stack, can save system resources by turning off guard areas.
2. When threads allocate large data structures on the stack, large guard areas may be needed to detect stack overflow.

The default size of the guard area is left implementation-defined since on systems supporting very large page sizes, the overhead might be substantial if at least one guard page is required by default.

If an implementation detects that the value specified by the `attr` argument to `pthread_attr_getguardsize()` or `pthread_attr_setguardsize()` does not refer to an initialized thread attributes object, it is recommended that the function should fail and report an [EINVAL] error.

## FUTURE DIRECTIONS

None.

## SEE ALSO

XBD `<pthread.h>`, `<sys/mman.h>`

## CHANGE HISTORY

First released in Issue 5.

### Issue 6

In the ERRORS section, a third [EINVAL] error condition is removed as it is covered by the second error condition.

The **restrict** keyword is added to the `pthread_attr_getguardsize()` prototype for alignment with the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/74 is applied, updating the ERRORS section to remove the [EINVAL] error ("The attribute `attr` is invalid."), and replacing it with the optional [EINVAL] error.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/76 is applied, adding the example to the EXAMPLES section.

### Issue 7

SD5-XSH-ERN-111 is applied, removing the reference to the `stack` attribute in the DESCRIPTION.

SD5-XSH-ERN-175 is applied, updating the DESCRIPTION to note that the default size of the guard area is implementation-defined.

The `pthread_attr_getguardsize()` and `pthread_attr_setguardsize()` functions are moved from the XSI option to the Base.

The [EINVAL] error for an uninitialized thread attributes object is removed; this condition results in undefined behavior.

---

## 1.119. `pthread_attr_getinheritsched`, `pthread_attr_setinheritsched` — get and set the inheritsched attribute (REALTIME THREADS)

### SYNOPSIS

```
#include <pthread.h>

int pthread_attr_getinheritsched(const pthread_attr_t *restrict attr,
                                 int *restrict inheritsched);

int pthread_attr_setinheritsched(pthread_attr_t *attr,
                                 int inheritsched);
```

### DESCRIPTION

The `pthread_attr_getinheritsched()` and `pthread_attr_setinheritsched()` functions, respectively, shall get and set the `inheritsched` attribute in the `attr` argument.

When the attributes objects are used by `pthread_create()`, the `inheritsched` attribute determines how the other scheduling attributes of the created thread shall be set.

The supported values of `inheritsched` shall be:

- **PTHREAD\_INHERIT\_SCHED**

Specifies that the thread scheduling attributes shall be inherited from the creating thread, and the scheduling attributes in this `attr` argument shall be ignored.

- **PTHREAD\_EXPLICIT\_SCHED**

Specifies that the thread scheduling attributes shall be set to the corresponding values from this attributes object.

The symbols `PTHREAD_INHERIT_SCHED` and `PTHREAD_EXPLICIT_SCHED` are defined in the `<pthread.h>` header.

The following thread scheduling attributes defined by POSIX.1-2024 are affected by the `inheritsched` attribute: scheduling policy (`schedpolicy`), scheduling parameters (`schedparam`), and scheduling contention scope (`contentionscope`).

The behavior is undefined if the value specified by the `attr` argument to `pthread_attr_getinheritsched()` or `pthread_attr_setinheritsched()` does not refer to an initialized thread attributes object.

## RETURN VALUE

If successful, the `pthread_attr_getinheritsched()` and `pthread_attr_setinheritsched()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

## ERRORS

The `pthread_attr_setinheritsched()` function shall fail if:

- **[ENOTSUP]**

An attempt was made to set the attribute to an unsupported value.

The `pthread_attr_setinheritsched()` function may fail if:

- **[EINVAL]**

The value of `inheritsched` is not valid.

These functions shall not return an error code of `[EINTR]`.

## EXAMPLES

None.

## APPLICATION USAGE

After these attributes have been set, a thread can be created with the specified attributes using `pthread_create()`. Using these routines does not affect the current running thread.

See [2.9.4 Thread Scheduling](#) for further details on thread scheduling attributes and their default settings.

## RATIONALE

If an implementation detects that the value specified by the `attr` argument to `pthread_attr_getinheritsched()` or `pthread_attr_setinheritsched()`

does not refer to an initialized thread attributes object, it is recommended that the function should fail and report an `[EINVAL]` error.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_attr_destroy()`
- `pthread_attr_getscope()`
- `pthread_attr_getschedpolicy()`
- `pthread_attr_getschedparam()`
- `pthread_create()`
- `<pthread.h>`
- `<sched.h>`

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Marked as part of the Realtime Threads Feature Group.

### Issue 6

The `pthread_attr_getinheritsched()` and `pthread_attr_setinheritsched()` functions are marked as part of the Threads and Thread Execution Scheduling options.

The `[ENOSYS]` error condition has been removed as stubs need not be provided if an implementation does not support the Thread Execution Scheduling option.

The `restrict` keyword is added to the `pthread_attr_getinheritsched()` prototype for alignment with the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/75 is applied, clarifying the values of `inheritsched` in the DESCRIPTION and adding two optional `[EINVAL]` errors to the ERRORS section for checking when `attr` refers to an uninitialized thread attribute object.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/77 is applied, adding a reference to [2.9.4 Thread Scheduling](#) in the APPLICATION USAGE section.

## Issue 7

The `pthread_attr_getinheritsched()` and `pthread_attr_setinheritsched()` functions are marked only as part of the Thread Execution Scheduling option as the Threads option is now part of the Base.

The `[EINVAL]` error for an uninitialized thread attributes object is removed; this condition results in undefined behavior.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0450 [314] is applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0262 [757] is applied.

## 1.120. `pthread_attr_getschedparam`, `pthread_attr_setschedparam` — get and set the schedparam attribute

### SYNOPSIS

```
#include <pthread.h>

int pthread_attr_getschedparam(const pthread_attr_t *restrict attr,
                               struct sched_param *restrict param)

int pthread_attr_setschedparam(pthread_attr_t *restrict attr,
                               const struct sched_param *restrict |
```

### DESCRIPTION

The `pthread_attr_getschedparam()` and `pthread_attr_setschedparam()` functions, respectively, shall get and set the scheduling parameter attributes in the `attr` argument. The contents of the `param` structure are defined in the `<sched.h>` header. For the SCHED\_FIFO and SCHED\_RR policies, the only required member of `param` is `sched_priority`.

[TSP] For the SCHED\_SPORADIC policy, the required members of the `param` structure are `sched_priority`, `sched_ss_low_priority`, `sched_ss_repl_period`, `sched_ss_init_budget`, and `sched_ss_max_repl`. The specified `sched_ss_repl_period` needs to be greater than or equal to the specified `sched_ss_init_budget` for the function to succeed; if it is not, then the function shall fail. The value of `sched_ss_max_repl` shall be within the inclusive range  $[1, \{SS\_REPL\_MAX\}]$  for the function to succeed; if not, the function shall fail. It is unspecified whether the `sched_ss_repl_period` and `sched_ss_init_budget` values are stored as provided by this function or are rounded to align with the resolution of the clock being used.

The behavior is undefined if the value specified by the `attr` argument to `pthread_attr_getschedparam()` or `pthread_attr_setschedparam()` does not refer to an initialized thread attributes object.

## RETURN VALUE

If successful, the `pthread_attr_getschedparam()` and `pthread_attr_setschedparam()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

## ERRORS

The `pthread_attr_setschedparam()` function shall fail if:

- **ENOTSUP** - An attempt was made to set the attribute to an unsupported value.

The `pthread_attr_setschedparam()` function may fail if:

- **EINVAL** - The value of `param` is not valid.

These functions shall not return an error code of [EINTR].

## EXAMPLES

None.

## APPLICATION USAGE

After these attributes have been set, a thread can be created with the specified attributes using `pthread_create()`. Using these routines does not affect the current running thread.

## RATIONALE

If an implementation detects that the value specified by the `attr` argument to `pthread_attr_getschedparam()` or `pthread_attr_setschedparam()` does not refer to an initialized thread attributes object, it is recommended that the function should fail and report an [EINVAL] error.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_attr_destroy()`
- `pthread_attr_getscope()`
- `pthread_attr_getinheritsched()`
- `pthread_attr_getschedpolicy()`
- `pthread_create()`
- `<pthread.h>`
- `<sched.h>`

## CHANGE HISTORY

### First released in Issue 5

Included for alignment with the POSIX Threads Extension.

### Issue 6

- The `pthread_attr_getschedparam()` and `pthread_attr_setschedparam()` functions are marked as part of the Threads option.
- The SCHED\_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.
- The `restrict` keyword is added to the `pthread_attr_getschedparam()` and `pthread_attr_setschedparam()` prototypes for alignment with the ISO/IEC 9899:1999 standard.
- IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/78 is applied, updating the ERRORS section to include optional errors for the case when `attr` refers to an uninitialized thread attribute object.

### Issue 7

- The `pthread_attr_getschedparam()` and `pthread_attr_setschedparam()` functions are moved from the Threads option to the Base.
- Austin Group Interpretation 1003.1-2001 #119 is applied, clarifying the accuracy requirements for the `sched_ss_repl_period` and

`sched_ss_init_budget` values.

- The [EINVAL] error for an uninitialized thread attributes object is removed; this condition results in undefined behavior.
  - POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0451 [314] is applied.
-

## 1.121. pthread\_attr\_getschedpolicy

### SYNOPSIS

```
#include <pthread.h>

int pthread_attr_getschedpolicy(const pthread_attr_t *restrict attr,
                                int *restrict policy);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
```

### DESCRIPTION

The `pthread_attr_getschedpolicy()` and `pthread_attr_setschedpolicy()` functions, respectively, shall get and set the `schedpolicy` attribute in the `attr` argument.

The supported values of `policy` shall include `SCHED_FIFO`, `SCHED_RR`, and `SCHED_OTHER`, which are defined in the `<sched.h>` header. When threads executing with the scheduling policy `SCHED_FIFO`, `SCHED_RR`, or `SCHED_SPORADIC` are waiting on a mutex, they shall acquire the mutex in priority order when the mutex is unlocked.

The behavior is undefined if the value specified by the `attr` argument to `pthread_attr_getschedpolicy()` or `pthread_attr_setschedpolicy()` does not refer to an initialized thread attributes object.

### RETURN VALUE

If successful, the `pthread_attr_getschedpolicy()` and `pthread_attr_setschedpolicy()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

### ERRORS

The `pthread_attr_setschedpolicy()` function shall fail if:

- **[ENOTSUP]**

An attempt was made to set the attribute to an unsupported value.

The `pthread_attr_setschedpolicy()` function may fail if:

- **[EINVAL]**

The value of `policy` is not valid.

These functions shall not return an error code of [EINTR].

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

After these attributes have been set, a thread can be created with the specified attributes using `pthread_create()`. Using these routines does not affect the current running thread.

See [2.9.4 Thread Scheduling](#) for further details on thread scheduling attributes and their default settings.

## RATIONALE

If an implementation detects that the value specified by the `attr` argument to `pthread_attr_getschedpolicy()` or `pthread_attr_setschedpolicy()` does not refer to an initialized thread attributes object, it is recommended that the function should fail and report an [EINVAL] error.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_attr_destroy()`
- `pthread_attr_getscope()`
- `pthread_attr_getinheritsched()`
- `pthread_attr_getschedparam()`

- `pthread_create()`

XBD `<pthread.h>`, `<sched.h>`

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Marked as part of the Realtime Threads Feature Group.

### Issue 6

The `pthread_attr_getschedpolicy()` and `pthread_attr_setschedpolicy()` functions are marked as part of the Threads and Thread Execution Scheduling options.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Thread Execution Scheduling option.

The SCHED\_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.

The `restrict` keyword is added to the `pthread_attr_getschedpolicy()` prototype for alignment with the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/79 is applied, adding a reference to [2.9.4 Thread Scheduling](#) in the APPLICATION USAGE section.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/80 is applied, updating the ERRORS section to include optional errors for the case when `attr` refers to an uninitialized thread attribute object.

### Issue 7

The `pthread_attr_getschedpolicy()` and `pthread_attr_setschedpolicy()` functions are marked only as part of the Thread Execution Scheduling option as the Threads option is now part of the Base.

The [EINVAL] error for an uninitialized thread attributes object is removed; this condition results in undefined behavior.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0452 [314] is applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0263 [757] is applied.

---

## 1.122. `pthread_attr_getscope`, `pthread_attr_setscope` — get and set the `contentionscope` attribute (REALTIME THREADS)

### SYNOPSIS

```
#include <pthread.h>

int pthread_attr_getscope(const pthread_attr_t *restrict attr,
                          int *restrict contentionscope);

int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);
```

### DESCRIPTION

The `pthread_attr_getscope()` and `pthread_attr_setscope()` functions, respectively, shall get and set the `contentionscope` attribute in the `attr` object.

The `contentionscope` attribute may have the values `PTHREAD_SCOPE_SYSTEM`, signifying system scheduling contention scope, or `PTHREAD_SCOPE_PROCESS`, signifying process scheduling contention scope. The symbols `PTHREAD_SCOPE_SYSTEM` and `PTHREAD_SCOPE_PROCESS` are defined in the `<pthread.h>` header.

The behavior is undefined if the value specified by the `attr` argument to `pthread_attr_getscope()` or `pthread_attr_setscope()` does not refer to an initialized thread attributes object.

### RETURN VALUE

If successful, the `pthread_attr_getscope()` and `pthread_attr_setscope()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

### ERRORS

The `pthread_attr_setscope()` function shall fail if:

- **[ENOTSUP]**: An attempt was made to set the attribute to an unsupported value.

The `pthread_attr_setscope()` function may fail if:

- **[EINVAL]**: The value of *contentionscope* is not valid.

These functions shall not return an error code of `[EINTR]`.

## EXAMPLES

None.

## APPLICATION USAGE

After these attributes have been set, a thread can be created with the specified attributes using `pthread_create()`. Using these routines does not affect the current running thread.

See [2.9.4 Thread Scheduling](#) for further details on thread scheduling attributes and their default settings.

## RATIONALE

If an implementation detects that the value specified by the *attr* argument to `pthread_attr_getscope()` or `pthread_attr_setscope()` does not refer to an initialized thread attributes object, it is recommended that the function should fail and report an `[EINVAL]` error.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_attr_destroy()`
- `pthread_attr_getinheritsched()`
- `pthread_attr_getschedpolicy()`
- `pthread_attr_getschedparam()`

- `pthread_create()`

XBD `<pthread.h>`, `<sched.h>`

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Marked as part of the Realtime Threads Feature Group.

### Issue 6

The `pthread_attr_getscope()` and `pthread_attr_setscope()` functions are marked as part of the Threads and Thread Execution Scheduling options.

The `[ENOSYS]` error condition has been removed as stubs need not be provided if an implementation does not support the Thread Execution Scheduling option.

The `restrict` keyword is added to the `pthread_attr_getscope()` prototype for alignment with the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/81 is applied, adding a reference to [2.9.4 Thread Scheduling](#) in the APPLICATION USAGE section.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/82 is applied, updating the ERRORS section to include optional errors for the case when `attr` refers to an uninitialized thread attribute object.

### Issue 7

The `pthread_attr_getscope()` and `pthread_attr_setscope()` functions are marked only as part of the Thread Execution Scheduling option as the Threads option is now part of the Base.

The `[EINVAL]` error for an uninitialized thread attributes object is removed; this condition results in undefined behavior.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0453 [314] is applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0264 [757] is applied.

---

## 1.123. `pthread_attr_getstack`, `pthread_attr_setstack` — get and set stack attributes

### SYNOPSIS

```
[TSA TSS]
#include <pthread.h>

int pthread_attr_getstack(const pthread_attr_t *restrict attr,
                          void **restrict stackaddr,
                          size_t *restrict stacksize);

int pthread_attr_setstack(pthread_attr_t *attr,
                         void *stackaddr,
                         size_t stacksize);
```

### DESCRIPTION

The `pthread_attr_getstack()` and `pthread_attr_setstack()` functions, respectively, shall get and set the thread creation stack attributes `stackaddr` and `stacksize` in the `attr` object.

The stack attributes specify the area of storage to be used for the created thread's stack. The base (lowest addressable byte) of the storage shall be `stackaddr`, and the size of the storage shall be `stacksize` bytes. The `stacksize` shall be at least `{PTHREAD_STACK_MIN}`. The `pthread_attr_setstack()` function may fail with `[EINVAL]` if `stackaddr` does not meet implementation-defined alignment requirements. All pages within the stack described by `stackaddr` and `stacksize` shall be both readable and writable by the thread.

If the `pthread_attr_getstack()` function is called before the `stackaddr` attribute has been set, the behavior is unspecified.

The behavior is undefined if the value specified by the `attr` argument to `pthread_attr_getstack()` or `pthread_attr_setstack()` does not refer to an initialized thread attributes object.

### RETURN VALUE

Upon successful completion, these functions shall return a value of 0; otherwise, an error number shall be returned to indicate the error.

The `pthread_attr_getstack()` function shall store the stack attribute values in `stackaddr` and `stacksize` if successful.

## ERRORS

The `pthread_attr_setstack()` function shall fail if:

- **[EINVAL]**

The value of `stacksize` is less than {PTHREAD\_STACK\_MIN} or exceeds an implementation-defined limit.

The `pthread_attr_setstack()` function may fail if:

- **[EINVAL]**

The value of `stackaddr` does not have proper alignment to be used as a stack, or ((char \*) `stackaddr` + `stacksize`) lacks proper alignment.

- **[EACCES]**

The stack page(s) described by `stackaddr` and `stacksize` are not both readable and writable by the thread.

These functions shall not return an error code of [EINTR].

## EXAMPLES

None.

## APPLICATION USAGE

These functions are appropriate for use by applications in an environment where the stack for a thread must be placed in some particular region of memory.

While it might seem that an application could detect stack overflow by providing a protected page outside the specified stack region, this cannot be done portably. Implementations are free to place the thread's initial stack pointer anywhere within the specified region to accommodate the machine's stack pointer behavior and allocation requirements. Furthermore, on some architectures, such as the IA-64, "overflow" might mean that two separate stack pointers allocated within the region will overlap somewhere in the middle of the region.

After a successful call to `pthread_attr_setstack()`, the storage area specified by the `stackaddr` parameter is under the control of the implementation, as described in [2.9.8 Use of Application-Managed Thread Stacks](#).

The specification of the `stackaddr` attribute presents several ambiguities that make portable use of these functions impossible. For example, the standard allows implementations to impose arbitrary alignment requirements on `stackaddr`. Applications cannot assume that a buffer obtained from `malloc()` is suitably aligned. Note that although the `stacksize` value passed to `pthread_attr_setstack()` must satisfy alignment requirements, the same is not true for `pthread_attr_setstacksize()` where the implementation must increase the specified size if necessary to achieve the proper alignment.

## RATIONALE

If an implementation detects that the value specified by the `attr` argument to `pthread_attr_getstack()` or `pthread_attr_setstack()` does not refer to an initialized thread attributes object, it is recommended that the function should fail and report an [EINVAL] error.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_attr_destroy()`
- `pthread_attr_getdetachstate()`
- `pthread_attr_getstacksize()`
- `pthread_create()`
- XBD `<limits.h>`
- XBD `<pthread.h>`

## CHANGE HISTORY

First released in Issue 6. Developed as part of the XSI option and brought into the BASE by IEEE PASC Interpretation 1003.1 #101.

## IEEE Std 1003.1-2001/Cor 2-2004

- Item XSH/TC2/D6/83 is applied, updating the APPLICATION USAGE section to refer to [2.9.8 Use of Application-Managed Thread Stacks](#).
- Item XSH/TC/D6/84 is applied, updating the ERRORS section to include optional errors for the case when `attr` refers to an uninitialized thread attribute object.

### Issue 7

- SD5-XSH-ERN-66 is applied, correcting the use of `attr` in the [EINVAL] error condition.
  - Austin Group Interpretation 1003.1-2001 #057 is applied, clarifying the behavior if the function is called before the `stackaddr` attribute is set.
  - SD5-XSH-ERN-157 is applied, updating the APPLICATION USAGE section.
  - The description of the `stackaddr` attribute is updated in the DESCRIPTION and APPLICATION USAGE sections.
  - The [EINVAL] error for an uninitialized thread attributes object is removed; this condition results in undefined behavior.
-

## 1.124. `pthread_attr_getstackaddr`, `pthread_attr_setstackaddr`

---

### SYNOPSIS

```
[0B] int pthread_attr_getstackaddr(const pthread_attr_t *restrict attr,
                                   void **restrict stackaddr);
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
```

### DESCRIPTION

The `pthread_attr_getstackaddr()` and `pthread_attr_setstackaddr()` functions, respectively, shall get and set the thread creation stackaddr attribute in the attr object.

The stackaddr attribute specifies the location of storage to be used for the created thread's stack. The size of the storage shall be at least {PTHREAD\_STACK\_MIN}.

### RETURN VALUE

Upon successful completion, `pthread_attr_getstackaddr()` and `pthread_attr_setstackaddr()` shall return a value of 0; otherwise, an error number shall be returned to indicate the error.

The `pthread_attr_getstackaddr()` function stores the stackaddr attribute value in stackaddr if successful.

### ERRORS

These functions may fail if:

- [EINVAL]
- The value specified by attr does not refer to an initialized thread attribute object.

These functions shall not return an error code of [EINTR].

## EXAMPLES

None.

## APPLICATION USAGE

The specification of the stackaddr attribute presents several ambiguities that make portable use of these interfaces impossible. The description of the single address parameter as a "stack" does not specify a particular relationship between the address and the "stack" implied by that address. For example, the address may be taken as the low memory address of a buffer intended for use as a stack, or it may be taken as the address to be used as the initial stack pointer register value for the new thread. These two are not the same except for a machine on which the stack grows "up" from low memory to high, and on which a "push" operation first stores the value in memory and then increments the stack pointer register. Further, on a machine where the stack grows "down" from high memory to low, interpretation of the address as the "low memory" address requires a determination of the intended size of the stack. IEEE Std 1003.1-2001 has introduced the new interfaces `pthread_attr_setstack()` and `pthread_attr_getstack()` to resolve these ambiguities.

After a successful call to `pthread_attr_setstackaddr()`, the storage area specified by the stackaddr parameter is under the control of the implementation, as described in Use of Application-Managed Thread Stacks.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_attr_destroy()`
- `pthread_attr_getdetachstate()`
- `pthread_attr_getstack()`
- `pthread_attr_getstacksize()`

- `pthread_attr_setstack()`
- `pthread_create()`
- the Base Definitions volume of IEEE Std 1003.1-2001, `<limits.h>`, `<pthread.h>`

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

### Issue 6

The `pthread_attr_getstackaddr()` and `pthread_attr_setstackaddr()` functions are marked as part of the Threads and Thread Stack Address Attribute options.

The **restrict** keyword is added to the `pthread_attr_getstackaddr()` prototype for alignment with the ISO/IEC 9899:1999 standard.

These functions are marked obsolescent.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/85 is applied, updating the APPLICATION USAGE section to refer to Use of Application-Managed Thread Stacks.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/86 is applied, updating the ERRORS section to include optional errors for the case when attr refers to an uninitialized thread attribute object.

---

## 1.125. `pthread_attr_getstacksize`, `pthread_attr_setstacksize` — get and set the stacksize attribute

---

### SYNOPSIS

```
#include <pthread.h>

int pthread_attr_getstacksize(const pthread_attr_t *restrict attr,
                               size_t *restrict stacksize);
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
```

### DESCRIPTION

The `pthread_attr_getstacksize()` and `pthread_attr_setstacksize()` functions, respectively, shall get and set the thread creation *stacksize* attribute in the *attr* object.

The *stacksize* attribute shall define the minimum stack size (in bytes) allocated for the created threads stack.

The behavior is undefined if the value specified by the *attr* argument to `pthread_attr_getstacksize()` or `pthread_attr_setstacksize()` does not refer to an initialized thread attributes object.

### RETURN VALUE

Upon successful completion, `pthread_attr_getstacksize()` and `pthread_attr_setstacksize()` shall return a value of 0; otherwise, an error number shall be returned to indicate the error.

The `pthread_attr_getstacksize()` function stores the *stacksize* attribute value in *stacksize* if successful.

### ERRORS

The `pthread_attr_setstacksize()` function shall fail if:

**[EINVAL]**

The value of *stacksize* is less than {PTHREAD\_STACK\_MIN} or exceeds a

system-imposed limit.

These functions shall not return an error code of [EINTR].

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

If an implementation detects that the value specified by the *attr* argument to `pthread_attr_getstacksize()` or `pthread_attr_setstacksize()` does not refer to an initialized thread attributes object, it is recommended that the function should fail and report an [EINVAL] error.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_attr_destroy()`
- `pthread_attr_getdetachstate()`
- `pthread_create()`
- `<limits.h>`
- `<pthread.h>`

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

## Issue 6

The `pthread_attr_getstacksize()` and `pthread_attr_setstacksize()` functions are marked as part of the Threads and Thread Stack Size Attribute options.

The **restrict** keyword is added to the `pthread_attr_getstacksize()` prototype for alignment with the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/43 is applied, correcting the margin code in the SYNOPSIS from TSA to TSS and updating the CHANGE HISTORY from "Thread Stack Address Attribute" option to "Thread Stack Size Attribute" option.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/87 is applied, updating the ERRORS section to include optional errors for the case when *attr* refers to an uninitialized thread attribute object.

## Issue 7

The `pthread_attr_getstacksize()` and `pthread_attr_setstacksize()` functions are marked only as part of the Thread Stack Size Attribute option as the Threads option is now part of the Base.

The [EINVAL] error for an uninitialized thread attributes object is removed; this condition results in undefined behavior.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0265 [757] is applied.

---

## 1.126. pthread\_cancel

---

### SYNOPSIS

```
#include <pthread.h>

int pthread_cancel(pthread_t thread);
```

### DESCRIPTION

The `pthread_cancel()` function shall request that `thread` be canceled. The target thread's cancelability state and type determines when the cancellation takes effect. When the cancellation is acted on, the cancellation cleanup handlers for `thread` shall be called. When the last cancellation cleanup handler returns, the thread-specific data destructor functions shall be called for `thread`. When the last destructor function returns, `thread` shall be terminated. It shall not be an error to request cancellation of a zombie thread.

The cancellation processing in the target thread shall run asynchronously with respect to the calling thread returning from `pthread_cancel()`.

If `thread` refers to a thread that was created using `thrd_create()`, the behavior is undefined.

### RETURN VALUE

If successful, the `pthread_cancel()` function shall return zero; otherwise, an error number shall be returned to indicate the error.

### ERRORS

The `pthread_cancel()` function shall not return an error code of [EINTR].

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

Two alternative functions were considered for sending the cancellation notification to a thread. One would be to define a new SIGCANCEL signal that had the cancellation semantics when delivered; the other was to define the new `pthread_cancel()` function, which would trigger the cancellation semantics.

The advantage of a new signal was that so much of the delivery criteria were identical to that used when trying to deliver a signal that making cancellation notification a signal was seen as consistent. Indeed, many implementations implement cancellation using a special signal. On the other hand, there would be no signal functions that could be used with this signal except `pthread_kill()`, and the behavior of the delivered cancellation signal would be unlike any previously existing defined signal.

The benefits of a special function include the recognition that this signal would be defined because of the similar delivery criteria and that this is the only common behavior between a cancellation request and a signal. In addition, the cancellation delivery mechanism does not have to be implemented as a signal. There are also strong, if not stronger, parallels with language exception mechanisms than with signals that are potentially obscured if the delivery mechanism is visibly closer to signals.

In the end, it was considered that as there were so many exceptions to the use of the new signal with existing signals functions it would be misleading. A special function has resolved this problem. This function was carefully defined so that an implementation wishing to provide the cancellation functions on top of signals could do so. The special function also means that implementations are not obliged to implement cancellation with signals.

If an implementation detects use of a thread ID after the end of its lifetime, it is recommended that the function should fail and report an [ESRCH] error.

Historical implementations varied on the result of a `pthread_cancel()` with a thread ID indicating a zombie thread. Some indicated success with nothing further to do because the thread had already terminated, while others gave an error of

[ESRCH]. Since the definition of thread lifetime in this standard covers zombie threads, the [ESRCH] error as described is inappropriate in this case and implementations that give this error do not conform.

Use of `pthread_cancel()` to cancel a thread that was created using `thrd_create()` is undefined because `thrd_join()` has no way to indicate a thread was cancelled. The standard developers considered adding a `thrd_canceled` enumeration constant that `thrd_join()` would return in this case. However, this return would be unexpected in code that is written to conform to the ISO C standard, and it would also not solve the problem that threads which use only ISO C `<threads.h>` interfaces (such as ones created by third party libraries written to conform to the ISO C standard) have no way to handle being cancelled, as the ISO C standard does not provide cancellation cleanup handlers.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_exit()`
- `pthread_cond_clockwait()`
- `pthread_join()`
- `pthread_setcancelstate()`

XBD `<pthread.h>`

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

### Issue 6

The `pthread_cancel()` function is marked as part of the Threads option.

### Issue 7

The `pthread_cancel()` function is moved from the Threads option to the Base.

Austin Group Interpretation 1003.1-2001 #142 is applied, removing the [ESRCH] error condition.

## Issue 8

Austin Group Defect 792 is applied, adding a requirement that passing the thread ID of a zombie thread to `pthread_cancel()` is not treated as an error.

Austin Group Defect 1302 is applied, updating the page to account for the addition of `<threads.h>` interfaces.

---

## 1.127. pthread\_cleanup\_pop, pthread\_cleanup\_push

---

### SYNOPSIS

```
#include <pthread.h>

void pthread_cleanup_pop(int execute);
void pthread_cleanup_push(void (*routine)(void*), void *arg);
```

### DESCRIPTION

The `pthread_cleanup_pop()` function shall remove the routine at the top of the calling thread's cancellation cleanup stack and optionally invoke it (if `execute` is non-zero).

The `pthread_cleanup_push()` function shall push the specified cancellation cleanup handler *routine* onto the calling thread's cancellation cleanup stack. The cancellation cleanup handler shall be popped from the cancellation cleanup stack and invoked with the argument *arg* when:

- The thread exits (that is, calls `pthread_exit()`).
- The thread acts upon a cancellation request.
- The thread calls `pthread_cleanup_pop()` with a non-zero `execute` argument.

It is unspecified whether `pthread_cleanup_push()` and `pthread_cleanup_pop()` are macros or functions. If a macro definition is suppressed in order to access an actual function, or a program defines an external identifier with any of these names, the behavior is undefined. The application shall ensure that they appear as statements, and in pairs within the same lexical scope (that is, the `pthread_cleanup_push()` macro may be thought to expand to a token list whose first token is '`{`' with `pthread_cleanup_pop()` expanding to a token list whose last token is the corresponding '`}`' ).

The effect of calling `longjmp()` or `siglongjmp()` is undefined if there have been any calls to `pthread_cleanup_push()` or `pthread_cleanup_pop()` made without the matching call since the jump buffer was filled. The effect of calling `longjmp()` or `siglongjmp()` from inside a cancellation cleanup handler is also undefined unless the jump buffer was also filled in the cancellation cleanup handler.

Invoking a cancellation cleanup handler may terminate the execution of any code block being executed by the thread whose execution began after the corresponding invocation of `pthread_cleanup_push()`.

The effect of the use of `return`, `break`, `continue`, and `goto` to prematurely leave a code block described by a pair of `pthread_cleanup_push()` and `pthread_cleanup_pop()` functions calls is undefined.

## RETURN VALUE

The `pthread_cleanup_push()` and `pthread_cleanup_pop()` functions shall not return a value.

## ERRORS

No errors are defined.

These functions shall not return an error code of [EINTR].

## EXAMPLES

The following is an example using thread primitives to implement a cancelable, writers-priority read-write lock:

```
typedef struct {
    pthread_mutex_t lock;
    pthread_cond_t rcond,
                  wcond;
    int lock_count; /* < 0 .. Held by writer. */
                    /* > 0 .. Held by lock_count readers. */
                    /* = 0 .. Held by nobody. */
    int waiting_writers; /* Count of waiting writers. */
} rwlock;

void
waiting_reader_cleanup(void *arg)
{
    rwlock *l;

    l = (rwlock *) arg;
    pthread_mutex_unlock(&l->lock);
}

void
lock_for_read(rwlock *l)
```

```

{

    pthread_mutex_lock(&l->lock);
    pthread_cleanup_push(waiting_reader_cleanup, l);
    while ((l->lock_count < 0) || (l->waiting_writers != 0))
        pthread_cond_wait(&l->rcond, &l->lock);
    l->lock_count++;
}

/*
 * Note the pthread_cleanup_pop executes
 * waiting_reader_cleanup.
 */
pthread_cleanup_pop(1);
}

void
release_read_lock(rwlock *l)
{
    pthread_mutex_lock(&l->lock);
    if (--l->lock_count == 0)
        pthread_cond_signal(&l->wcond);
    pthread_mutex_unlock(&l->lock);
}

void
waiting_writer_cleanup(void *arg)
{
    rwlock *l;

    l = (rwlock *) arg;
    if ((--l->waiting_writers == 0) && (l->lock_count >= 0)) {
        /*
         * This only happens if we have been canceled. If the
         * lock is not held by a writer, there may be readers who
         * were blocked because waiting_writers was positive; they
         * can now be unblocked.
        */
        pthread_cond_broadcast(&l->rcond);
    }
    pthread_mutex_unlock(&l->lock);
}

void
lock_for_write(rwlock *l)
{
    pthread_mutex_lock(&l->lock);
    l->waiting_writers++;
    pthread_cleanup_push(waiting_writer_cleanup, l);
    while (l->lock_count != 0)
        pthread_cond_wait(&l->wcond, &l->lock);
    l->lock_count = -1;
}

/*
 * Note the pthread_cleanup_pop executes
 */

```

```

 * waiting_writer_cleanup.
 */
pthread_cleanup_pop(1);
}

void
release_write_lock(rwlock *l)
{
    pthread_mutex_lock(&l->lock);
    l->lock_count = 0;
    if (l->waiting_writers == 0)
        pthread_cond_broadcast(&l->rcond);
    else
        pthread_cond_signal(&l->wcond);
    pthread_mutex_unlock(&l->lock);
}

/*
 * This function is called to initialize the read/write lock.
 */
void
initialize_rwlock(rwlock *l)
{
    pthread_mutex_init(&l->lock, pthread_mutexattr_default);
    pthread_cond_init(&l->wcond, pthread_condattr_default);
    pthread_cond_init(&l->rcond, pthread_condattr_default);
    l->lock_count = 0;
    l->waiting_writers = 0;
}

reader_thread()
{
    lock_for_read(&lock);
    pthread_cleanup_push(release_read_lock, &lock);
/*
 * Thread has read lock.
 */
    pthread_cleanup_pop(1);
}

writer_thread()
{
    lock_for_write(&lock);
    pthread_cleanup_push(release_write_lock, &lock);
/*
 * Thread has write lock.
 */
    pthread_cleanup_pop(1);
}

```

# APPLICATION USAGE

The two routines that push and pop cancellation cleanup handlers, `pthread_cleanup_push()` and `pthread_cleanup_pop()`, can be thought of as left and right-parentheses. They always need to be matched.

## RATIONALE

The restriction that the two routines that push and pop cancellation cleanup handlers, `pthread_cleanup_push()` and `pthread_cleanup_pop()`, have to appear in the same lexical scope allows for efficient macro or compiler implementations and efficient storage management. A sample implementation of these routines as macros might look like this:

```
#define pthread_cleanup_push(rtn,arg) { \
    struct _pthread_handler_rec __cleanup_handler, **__head; \
    __cleanup_handler.rtn = rtn; \
    __cleanup_handler.arg = arg; \
    (void) pthread_getspecific(_pthread_handler_key, &__head); \
    __cleanup_handler.next = *__head; \
    *__head = &__cleanup_handler; \
}

#define pthread_cleanup_pop(ex) \
    *__head = __cleanup_handler.next; \
    if (ex) (*__cleanup_handler.rtn)(__cleanup_handler.arg); \
}
```

A more ambitious implementation of these routines might do even better by allowing the compiler to note that the cancellation cleanup handler is a constant and can be expanded inline.

This volume of POSIX.1-2024 currently leaves unspecified the effect of calling `longjmp()` from a signal handler executing in a POSIX System Interfaces function. If an implementation wants to allow this and give the programmer reasonable behavior, the `longjmp()` function has to call all cancellation cleanup handlers that have been pushed but not popped since the time `setjmp()` was called.

Consider a multi-threaded function called by a thread that uses signals. If a signal were delivered to a signal handler during the operation of `qsort()` and that handler were to call `longjmp()` (which, in turn, did **not** call the cancellation cleanup handlers) the helper threads created by the `qsort()` function would not be canceled. Instead, they would continue to execute and write into the argument array even though the array might have been popped off the stack.

Note that the specified cleanup handling mechanism is especially tied to the C language and, while the requirement for a uniform mechanism for expressing cleanup is language-independent, the mechanism used in other languages may be quite different. In addition, this mechanism is really only necessary due to the lack of a real exception mechanism in the C language, which would be the ideal solution.

There is no notion of a cancellation cleanup-safe function. If an application has no cancellation points in its signal handlers, blocks any signal whose handler may have cancellation points while calling async-unsafe functions, or disables cancellation while calling async-unsafe functions, all functions may be safely called from cancellation cleanup routines.

## FUTURE DIRECTIONS

None.

## SEE ALSO

`pthread_cancel()`, `pthread_setcancelstate()`

XBD `<pthread.h>`

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

### Issue 6

The `pthread_cleanup_pop()` and `pthread_cleanup_push()` functions are marked as part of the Threads option.

The APPLICATION USAGE section is added.

The normative text is updated to avoid use of the term "must" for application requirements.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/88 is applied, updating the DESCRIPTION to describe the consequences of prematurely leaving a code block defined by the `pthread_cleanup_push()` and `pthread_cleanup_pop()` functions.

## Issue 7

The `pthread_cleanup_pop()` and `pthread_cleanup_push()` functions are moved from the Threads option to the Base.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0454 [229] is applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0268 [624] is applied.

---

## Issue 8

Austin Group Defect 613 is applied, clarifying the relationship of automatic object lifetimes to cancellation cleanup functions.

## 1.128. pthread\_cleanup\_push, pthread\_cleanup\_pop

---

### SYNOPSIS

```
#include <pthread.h>

void pthread_cleanup_pop(int execute);
void pthread_cleanup_push(void (*routine)(void*), void *arg);
```

### DESCRIPTION

The `pthread_cleanup_pop()` function shall remove the routine at the top of the calling thread's cancellation cleanup stack and optionally invoke it (if `execute` is non-zero).

The `pthread_cleanup_push()` function shall push the specified cancellation cleanup handler *routine* onto the calling thread's cancellation cleanup stack. The cancellation cleanup handler shall be popped from the cancellation cleanup stack and invoked with the argument *arg* when:

- The thread exits (that is, calls `pthread_exit()`).
- The thread acts upon a cancellation request.
- The thread calls `pthread_cleanup_pop()` with a non-zero `execute` argument.

It is unspecified whether `pthread_cleanup_push()` and `pthread_cleanup_pop()` are macros or functions. If a macro definition is suppressed in order to access an actual function, or a program defines an external identifier with any of these names, the behavior is undefined. The application shall ensure that they appear as statements, and in pairs within the same lexical scope (that is, the `pthread_cleanup_push()` macro may be thought to expand to a token list whose first token is '{' with `pthread_cleanup_pop()` expanding to a token list whose last token is the corresponding '}').

The effect of calling `longjmp()` or `siglongjmp()` is undefined if there have been any calls to `pthread_cleanup_push()` or `pthread_cleanup_pop()` made without the matching call since the jump buffer was filled. The effect of calling `longjmp()` or `siglongjmp()` from inside a cancellation cleanup handler is also undefined unless the jump buffer was also filled in the cancellation cleanup handler.

Invoking a cancellation cleanup handler may terminate the execution of any code block being executed by the thread whose execution began after the corresponding invocation of `pthread_cleanup_push()`.

The effect of the use of `return`, `break`, `continue`, and `goto` to prematurely leave a code block described by a pair of `pthread_cleanup_push()` and `pthread_cleanup_pop()` functions calls is undefined.

## RETURN VALUE

The `pthread_cleanup_push()` and `pthread_cleanup_pop()` functions shall not return a value.

## ERRORS

No errors are defined.

These functions shall not return an error code of [EINTR].

## EXAMPLES

The following is an example using thread primitives to implement a cancelable, writers-priority read-write lock:

```
typedef struct {
    pthread_mutex_t lock;
    pthread_cond_t rcond,
                  wcond;
    int lock_count; /* < 0 .. Held by writer. */
                    /* > 0 .. Held by lock_count readers. */
                    /* = 0 .. Held by nobody. */
    int waiting_writers; /* Count of waiting writers. */
} rwlock;

void
waiting_reader_cleanup(void *arg)
{
    rwlock *l;

    l = (rwlock *) arg;
    pthread_mutex_unlock(&l->lock);
}

void
lock_for_read(rwlock *l)
```

```

{

    pthread_mutex_lock(&l->lock);
    pthread_cleanup_push(waiting_reader_cleanup, l);
    while ((l->lock_count < 0) || (l->waiting_writers != 0))
        pthread_cond_wait(&l->rcond, &l->lock);
    l->lock_count++;
}

/*
 * Note the pthread_cleanup_pop executes
 * waiting_reader_cleanup.
 */
pthread_cleanup_pop(1);
}

void
release_read_lock(rwlock *l)
{
    pthread_mutex_lock(&l->lock);
    if (--l->lock_count == 0)
        pthread_cond_signal(&l->wcond);
    pthread_mutex_unlock(&l->lock);
}

void
waiting_writer_cleanup(void *arg)
{
    rwlock *l;

    l = (rwlock *) arg;
    if ((--l->waiting_writers == 0) && (l->lock_count >= 0)) {
        /*
         * This only happens if we have been canceled. If the
         * lock is not held by a writer, there may be readers who
         * were blocked because waiting_writers was positive; they
         * can now be unblocked.
        */
        pthread_cond_broadcast(&l->rcond);
    }
    pthread_mutex_unlock(&l->lock);
}

void
lock_for_write(rwlock *l)
{
    pthread_mutex_lock(&l->lock);
    l->waiting_writers++;
    pthread_cleanup_push(waiting_writer_cleanup, l);
    while (l->lock_count != 0)
        pthread_cond_wait(&l->wcond, &l->lock);
    l->lock_count = -1;
}

/*
 * Note the pthread_cleanup_pop executes
 */

```

```

    * waiting_writer_cleanup.
    */
    pthread_cleanup_pop(1);
}

void
release_write_lock(rwlock *l)
{
    pthread_mutex_lock(&l->lock);
    l->lock_count = 0;
    if (l->waiting_writers == 0)
        pthread_cond_broadcast(&l->rcond);
    else
        pthread_cond_signal(&l->wcond);
    pthread_mutex_unlock(&l->lock);
}

/*
 * This function is called to initialize the read/write lock.
 */
void
initialize_rwlock(rwlock *l)
{
    pthread_mutex_init(&l->lock, pthread_mutexattr_default);
    pthread_cond_init(&l->wcond, pthread_condattr_default);
    pthread_cond_init(&l->rcond, pthread_condattr_default);
    l->lock_count = 0;
    l->waiting_writers = 0;
}

reader_thread()
{
    lock_for_read(&lock);
    pthread_cleanup_push(release_read_lock, &lock);
    /*
     * Thread has read lock.
     */
    pthread_cleanup_pop(1);
}

writer_thread()
{
    lock_for_write(&lock);
    pthread_cleanup_push(release_write_lock, &lock);
    /*
     * Thread has write lock.
     */
    pthread_cleanup_pop(1);
}

```

# APPLICATION USAGE

The two routines that push and pop cancellation cleanup handlers, `pthread_cleanup_push()` and `pthread_cleanup_pop()`, can be thought of as left and right-parentheses. They always need to be matched.

## RATIONALE

The restriction that the two routines that push and pop cancellation cleanup handlers, `pthread_cleanup_push()` and `pthread_cleanup_pop()`, have to appear in the same lexical scope allows for efficient macro or compiler implementations and efficient storage management. A sample implementation of these routines as macros might look like this:

```
#define pthread_cleanup_push(rtn,arg) { \
    struct _pthread_handler_rec __cleanup_handler, ***__head; \
    __cleanup_handler.rtn = rtn; \
    __cleanup_handler.arg = arg; \
    (void) pthread_getspecific(_pthread_handler_key, &__head); \
    __cleanup_handler.next = *__head; \
    *__head = &__cleanup_handler; \
}

#define pthread_cleanup_pop(ex) \
    *__head = __cleanup_handler.next; \
    if (ex) (*__cleanup_handler.rtn)(__cleanup_handler.arg); \
}
```

A more ambitious implementation of these routines might do even better by allowing the compiler to note that the cancellation cleanup handler is a constant and can be expanded inline.

This volume of POSIX.1-2024 currently leaves unspecified the effect of calling `longjmp()` from a signal handler executing in a POSIX System Interfaces function. If an implementation wants to allow this and give the programmer reasonable behavior, the `longjmp()` function has to call all cancellation cleanup handlers that have been pushed but not popped since the time `setjmp()` was called.

Consider a multi-threaded function called by a thread that uses signals. If a signal were delivered to a signal handler during the operation of `qsort()` and that handler were to call `longjmp()` (which, in turn, did **not** call the cancellation cleanup handlers) the helper threads created by the `qsort()` function would not be canceled. Instead, they would continue to execute and write into the argument array even though the array might have been popped off the stack.

Note that the specified cleanup handling mechanism is especially tied to the C language and, while the requirement for a uniform mechanism for expressing cleanup is language-independent, the mechanism used in other languages may be quite different. In addition, this mechanism is really only necessary due to the lack of a real exception mechanism in the C language, which would be the ideal solution.

There is no notion of a cancellation cleanup-safe function. If an application has no cancellation points in its signal handlers, blocks any signal whose handler may have cancellation points while calling async-unsafe functions, or disables cancellation while calling async-unsafe functions, all functions may be safely called from cancellation cleanup routines.

## FUTURE DIRECTIONS

None.

## SEE ALSO

`pthread_cancel()` ,  
`pthread_setcancelstate()`  
XBD `<pthread.h>`

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

### Issue 6

The `pthread_cleanup_pop()` and `pthread_cleanup_push()` functions are marked as part of the Threads option.

The APPLICATION USAGE section is added.

The normative text is updated to avoid use of the term "must" for application requirements.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/88 is applied, updating the DESCRIPTION to describe the consequences of prematurely leaving a code block defined by the `pthread_cleanup_push()` and `pthread_cleanup_pop()` functions.

## Issue 7

The `pthread_cleanup_pop()` and `pthread_cleanup_push()` functions are moved from the Threads option to the Base.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0454 [229] is applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0268 [624] is applied.

## Issue 8

Austin Group Defect 613 is applied, clarifying the relationship of automatic object lifetimes to cancellation cleanup functions.

## 1.129. `pthread_cond_broadcast`, `pthread_cond_signal` — broadcast or signal a condition

---

### SYNOPSIS

```
#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
```

### DESCRIPTION

These functions shall unblock threads blocked on a condition variable.

The `pthread_cond_broadcast()` function shall, as a single atomic operation, determine which threads, if any, are blocked on the specified condition variable `cond` and unblock all of these threads.

The `pthread_cond_signal()` function shall, as a single atomic operation, determine which threads, if any, are blocked on the specified condition variable `cond` and unblock at least one of these threads.

If more than one thread is blocked on a condition variable, the scheduling policy shall determine the order in which threads are unblocked. When each thread unblocked as a result of a `pthread_cond_broadcast()` or `pthread_cond_signal()` returns from its call to `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, or `pthread_cond_wait()`, the thread shall own the mutex with which it called `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, or `pthread_cond_wait()`. The thread(s) that are unblocked shall contend for the mutex according to the scheduling policy (if applicable), and as if each had called `pthread_mutex_lock()`.

The `pthread_cond_broadcast()` or `pthread_cond_signal()` functions may be called by a thread whether or not it currently owns the mutex that threads calling `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, or `pthread_cond_wait()` have associated with the condition variable during their waits; however, if predictable scheduling behavior is required, then that mutex shall be locked by the thread calling `pthread_cond_broadcast()` or `pthread_cond_signal()`.

The `pthread_cond_broadcast()` and `pthread_cond_signal()` functions shall have no effect if they determine that there are no threads blocked on `cond`.

The behavior is undefined if the value specified by the `cond` argument to `pthread_cond_broadcast()` or `pthread_cond_signal()` does not refer to an initialized condition variable.

## RETURN VALUE

If successful, the `pthread_cond_broadcast()` and `pthread_cond_signal()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

## ERRORS

These functions shall not return an error code of `[EINTR]`.

## EXAMPLES

None.

## APPLICATION USAGE

The `pthread_cond_broadcast()` function is used whenever the shared-variable state has been changed in a way that more than one thread can proceed with its task. Consider a single producer/multiple consumer problem, where the producer can insert multiple items on a list that is accessed one item at a time by the consumers. By calling the `pthread_cond_broadcast()` function, the producer would notify all consumers that might be waiting, and thereby the application would receive more throughput on a multi-processor. In addition, `pthread_cond_broadcast()` makes it easier to implement a read-write lock. The `pthread_cond_broadcast()` function is needed in order to wake up all waiting readers when a writer releases its lock. Finally, the two-phase commit algorithm can use this broadcast function to notify all clients of an impending transaction commit.

It is not safe to use the `pthread_cond_signal()` function in a signal handler that is invoked asynchronously. Even if it were safe, there would still be a race between the test of the Boolean `pthread_cond_wait()` that could not be efficiently eliminated.

Mutexes and condition variables are thus not suitable for releasing a waiting thread by signaling from code running in a signal handler.

# RATIONALE

If an implementation detects that the value specified by the `cond` argument to `pthread_cond_broadcast()` or `pthread_cond_signal()` does not refer to an initialized condition variable, it is recommended that the function should fail and report an `[EINVAL]` error.

## Multiple Awakenings by Condition Signal

On a multi-processor, it may be impossible for an implementation of `pthread_cond_signal()` to avoid the unblocking of more than one thread blocked on a condition variable. For example, consider the following partial implementation of `pthread_cond_wait()` and `pthread_cond_signal()`, executed by two threads in the order given. One thread is trying to wait on the condition variable, another is concurrently executing `pthread_cond_signal()`, while a third thread is already waiting.

```
pthread_cond_wait(mutex, cond):
    value = cond->value;                  /* 1 */
    pthread_mutex_unlock(mutex);          /* 2 */
    pthread_mutex_lock(cond->mutex);    /* 10 */
    if (value == cond->value) {          /* 11 */
        me->next_cond = cond->waiter;
        cond->waiter = me;
        pthread_mutex_unlock(cond->mutex);
        unable_to_run(me);
    } else
        pthread_mutex_unlock(cond->mutex); /* 12 */
    pthread_mutex_lock(mutex);          /* 13 */

pthread_cond_signal(cond):
    pthread_mutex_lock(cond->mutex);    /* 3 */
    cond->value++;                     /* 4 */
    if (cond->waiter) {                /* 5 */
        sleeper = cond->waiter;        /* 6 */
        cond->waiter = sleeper->next_cond; /* 7 */
        able_to_run(sleeper);          /* 8 */
    }
    pthread_mutex_unlock(cond->mutex); /* 9 */
```

The effect is that more than one thread can return from its call to `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, or `pthread_cond_wait()` as a result of one call to `pthread_cond_signal()`.

This effect is called "spurious wakeup". Note that the situation is self-correcting in that the number of threads that are so awakened is finite; for example, the next thread to call `pthread_cond_wait()` after the sequence of events above blocks.

While this problem could be resolved, the loss of efficiency for a fringe condition that occurs only rarely is unacceptable, especially given that one has to check the predicate associated with a condition variable anyway. Correcting this problem would unnecessarily reduce the degree of concurrency in this basic building block for all higher-level synchronization operations.

An added benefit of allowing spurious wakeups is that applications are forced to code a predicate-testing-loop around the condition wait. This also makes the application tolerate superfluous condition broadcasts or signals on the same condition variable that may be coded in some other part of the application. The resulting applications are thus more robust. Therefore, POSIX.1-2024 explicitly documents that spurious wakeups may occur.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_cond_clockwait()`
- `pthread_cond_destroy()`

XBD [4.15.2 Memory Synchronization](#),

## CHANGE HISTORY

### First released in Issue 5.

Included for alignment with the POSIX Threads Extension.

### Issue 6

The `pthread_cond_broadcast()` and `pthread_cond_signal()` functions are marked as part of the Threads option.

The APPLICATION USAGE section is added.

### Issue 7

The `pthread_cond_broadcast()` and `pthread_cond_signal()` functions are moved from the Threads option to the Base.

The `[EINVAL]` error for an uninitialized condition variable is removed; this condition results in undefined behavior.

## Issue 8

Austin Group Defect 609 is applied, adding atomicity requirements.

Austin Group Defect 1216 is applied, adding `pthread_cond_clockwait()`.

---

## 1.130. pthread\_cond\_destroy, pthread\_cond\_init

---

### SYNOPSIS

```
#include <pthread.h>

int pthread_cond_destroy(pthread_cond_t *cond);

int pthread_cond_init(pthread_cond_t *restrict cond,
                      const pthread_condattr_t *restrict attr);

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

### DESCRIPTION

The `pthread_cond_destroy()` function shall destroy the given condition variable specified by `cond`; the object becomes, in effect, uninitialized. An implementation may cause `pthread_cond_destroy()` to set the object referenced by `cond` to an invalid value. A destroyed condition variable object can be reinitialized using `pthread_cond_init()`; the results of otherwise referencing the object after it has been destroyed are undefined.

It shall be safe to destroy an initialized condition variable upon which no threads are currently blocked. Attempting to destroy a condition variable upon which other threads are currently blocked results in undefined behavior.

The `pthread_cond_init()` function shall initialize the condition variable referenced by `cond` with attributes referenced by `attr`. If `attr` is NULL, the default condition variable attributes shall be used; the effect is the same as passing the address of a default condition variable attributes object. Upon successful initialization, the state of the condition variable shall become initialized.

See [2.9.9 Synchronization Object Copies and Alternative Mappings](#) for further requirements.

Attempting to initialize an already initialized condition variable results in undefined behavior.

In cases where default condition variable attributes are appropriate, the macro `PTHREAD_COND_INITIALIZER` can be used to initialize condition variables. The effect shall be equivalent to dynamic initialization by a call to `pthread_cond_init()` with parameter `attr` specified as NULL, except that no error checks are performed.

The behavior is undefined if the value specified by the `cond` argument to `pthread_cond_destroy()` does not refer to an initialized condition variable.

The behavior is undefined if the value specified by the `attr` argument to `pthread_cond_init()` does not refer to an initialized condition variable attributes object.

## RETURN VALUE

If successful, the `pthread_cond_destroy()` and `pthread_cond_init()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

## ERRORS

The `pthread_cond_init()` function shall fail if:

- **[EAGAIN]**

The system lacked the necessary resources (other than memory) to initialize another condition variable.

- **[ENOMEM]**

Insufficient memory exists to initialize the condition variable.

These functions shall not return an error code of [EINTR].

---

## EXAMPLES

A condition variable can be destroyed immediately after all the threads that are blocked on it are awakened. For example, consider the following code:

```
struct list {
    pthread_mutex_t lm;
    ...
};

struct elt {
    key k;
    int busy;
    pthread_cond_t notbusy;
    ...
};
```

```

/* Find a list element and reserve it. */
struct elt *
list_find(struct list *lp, key k)
{
    struct elt *ep;

    pthread_mutex_lock(&lp->lm);
    while ((ep = find_elt(l, k) != NULL) && ep->busy)
        pthread_cond_wait(&ep->notbusy, &lp->lm);
    if (ep != NULL)
        ep->busy = 1;
    pthread_mutex_unlock(&lp->lm);
    return(ep);
}

delete_elt(struct list *lp, struct elt *ep)
{
    pthread_mutex_lock(&lp->lm);
    assert(ep->busy);
    ... remove ep from list ...
    ep->busy = 0; /* Paranoid. */
(A) pthread_cond_broadcast(&ep->notbusy);
    pthread_mutex_unlock(&lp->lm);
(B) pthread_cond_destroy(&ep->notbusy);
    free(ep);
}

```

In this example, the condition variable and its list element may be freed (line B) immediately after all threads waiting for it are awakened (line A), since the mutex and the code ensure that no other thread can touch the element to be deleted.

## APPLICATION USAGE

None.

## RATIONALE

If an implementation detects that the value specified by the `cond` argument to `pthread_cond_destroy()` does not refer to an initialized condition variable, it is recommended that the function should fail and report an [EINVAL] error.

If an implementation detects that the value specified by the `cond` argument to `pthread_cond_destroy()` or `pthread_cond_init()` refers to a condition variable that is in use (for example, in a `pthread_cond_wait()` call) by another thread, or detects that the value specified by the `cond` argument to

`pthread_cond_init()` refers to an already initialized condition variable, it is recommended that the function should fail and report an [EBUSY] error.

If an implementation detects that the value specified by the `attr` argument to `pthread_cond_init()` does not refer to an initialized condition variable attributes object, it is recommended that the function should fail and report an [EINVAL] error.

See also `pthread_mutex_destroy()`.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_cond_broadcast()`
- `pthread_cond_clockwait()`
- `pthread_mutex_destroy()`

XBD

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

### Issue 6

The `pthread_cond_destroy()` and `pthread_cond_init()` functions are marked as part of the Threads option.

IEEE PASC Interpretation 1003.1c #34 is applied, updating the DESCRIPTION.

The `restrict` keyword is added to the `pthread_cond_init()` prototype for alignment with the ISO/IEC 9899:1999 standard.

### Issue 7

The `pthread_cond_destroy()` and `pthread_cond_init()` functions are moved from the Threads option to the Base.

The [EINVAL] error for an uninitialized condition variable and an uninitialized condition variable attributes object is removed; this condition results in undefined behavior.

The [EBUSY] error for a condition variable already in use or an already initialized condition variable is removed; this condition results in undefined behavior.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0455 [70] is applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0269 [972] and XSH/TC2-2008/0270 [910] are applied.

---

## 1.131. `pthread_cond_destroy`, `pthread_cond_init` — destroy and initialize condition variables

### SYNOPSIS

```
#include <pthread.h>

int pthread_cond_destroy(pthread_cond_t *cond);

int pthread_cond_init(pthread_cond_t *restrict cond,
                      const pthread_condattr_t *restrict attr);

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

### DESCRIPTION

The `pthread_cond_destroy()` function shall destroy the given condition variable specified by `cond`; the object becomes, in effect, uninitialized. An implementation may cause `pthread_cond_destroy()` to set the object referenced by `cond` to an invalid value. A destroyed condition variable object can be reinitialized using `pthread_cond_init()`; the results of otherwise referencing the object after it has been destroyed are undefined.

It shall be safe to destroy an initialized condition variable upon which no threads are currently blocked. Attempting to destroy a condition variable upon which other threads are currently blocked results in undefined behavior.

The `pthread_cond_init()` function shall initialize the condition variable referenced by `cond` with attributes referenced by `attr`. If `attr` is NULL, the default condition variable attributes shall be used; the effect is the same as passing the address of a default condition variable attributes object. Upon successful initialization, the state of the condition variable shall become initialized.

See [2.9.9 Synchronization Object Copies and Alternative Mappings](#) for further requirements.

Attempting to initialize an already initialized condition variable results in undefined behavior.

In cases where default condition variable attributes are appropriate, the macro `PTHREAD_COND_INITIALIZER` can be used to initialize condition variables. The effect shall be equivalent to dynamic initialization by a call to

`pthread_cond_init()` with parameter `attr` specified as NULL, except that no error checks are performed.

The behavior is undefined if the value specified by the `cond` argument to `pthread_cond_destroy()` does not refer to an initialized condition variable.

The behavior is undefined if the value specified by the `attr` argument to `pthread_cond_init()` does not refer to an initialized condition variable attributes object.

## RETURN VALUE

If successful, the `pthread_cond_destroy()` and `pthread_cond_init()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

## ERRORS

The `pthread_cond_init()` function shall fail if:

- **[EAGAIN]**

The system lacked the necessary resources (other than memory) to initialize another condition variable.

- **[ENOMEM]**

Insufficient memory exists to initialize the condition variable.

These functions shall not return an error code of **[EINTR]**.

## EXAMPLES

A condition variable can be destroyed immediately after all the threads that are blocked on it are awakened. For example, consider the following code:

```
struct list {
    pthread_mutex_t lm;
    ...
};

struct elt {
    key k;
    int busy;
    pthread_cond_t notbusy;
    ...
};
```

```

/* Find a list element and reserve it. */
struct elt *
list_find(struct list *lp, key k)
{
    struct elt *ep;

    pthread_mutex_lock(&lp->lm);
    while ((ep = find_elt(l, k) != NULL) && ep->busy)
        pthread_cond_wait(&ep->notbusy, &lp->lm);
    if (ep != NULL)
        ep->busy = 1;
    pthread_mutex_unlock(&lp->lm);
    return(ep);
}

delete_elt(struct list *lp, struct elt *ep)
{
    pthread_mutex_lock(&lp->lm);
    assert(ep->busy);
    ... remove ep from list ...
    ep->busy = 0; /* Paranoid. */
(A) pthread_cond_broadcast(&ep->notbusy);
    pthread_mutex_unlock(&lp->lm);
(B) pthread_cond_destroy(&ep->notbusy);
    free(ep);
}

```

In this example, the condition variable and its list element may be freed (line B) immediately after all threads waiting for it are awakened (line A), since the mutex and the code ensure that no other thread can touch the element to be deleted.

## APPLICATION USAGE

None.

## RATIONALE

If an implementation detects that the value specified by the `cond` argument to `pthread_cond_destroy()` does not refer to an initialized condition variable, it is recommended that the function should fail and report an `[EINVAL]` error.

If an implementation detects that the value specified by the `cond` argument to `pthread_cond_destroy()` or `pthread_cond_init()` refers to a condition variable that is in use (for example, in a `pthread_cond_wait()` call) by another thread, or detects that the value specified by the `cond` argument to

`pthread_cond_init()` refers to an already initialized condition variable, it is recommended that the function should fail and report an **[EBUSY]** error.

If an implementation detects that the value specified by the `attr` argument to `pthread_cond_init()` does not refer to an initialized condition variable attributes object, it is recommended that the function should fail and report an **[EINVAL]** error.

See also `pthread_mutex_destroy()`.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_cond_broadcast()`
- `pthread_cond_clockwait()`
- `pthread_mutex_destroy()`
- XBD `<pthread.h>`

## CHANGE HISTORY

### First released in Issue 5

Included for alignment with the POSIX Threads Extension.

### Issue 6

- The `pthread_cond_destroy()` and `pthread_cond_init()` functions are marked as part of the Threads option.
- IEEE PASC Interpretation 1003.1c #34 is applied, updating the DESCRIPTION.
- The **restrict** keyword is added to the `pthread_cond_init()` prototype for alignment with the ISO/IEC 9899:1999 standard.

## Issue 7

- The `pthread_cond_destroy()` and `pthread_cond_init()` functions are moved from the Threads option to the Base.
- The **[EINVAL]** error for an uninitialized condition variable and an uninitialized condition variable attributes object is removed; this condition results in undefined behavior.
- The **[EBUSY]** error for a condition variable already in use or an already initialized condition variable is removed; this condition results in undefined behavior.
- POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0455 [70] is applied.
- POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0269 [972] and XSH/TC2-2008/0270 [910] are applied.

## 1.132. `pthread_cond_broadcast`, `pthread_cond_signal` — broadcast or signal a condition

---

### SYNOPSIS

```
#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
```

### DESCRIPTION

These functions shall unblock threads blocked on a condition variable.

The `pthread_cond_broadcast()` function shall, as a single atomic operation, determine which threads, if any, are blocked on the specified condition variable `cond` and unblock all of these threads.

The `pthread_cond_signal()` function shall, as a single atomic operation, determine which threads, if any, are blocked on the specified condition variable `cond` and unblock at least one of these threads.

If more than one thread is blocked on a condition variable, the scheduling policy shall determine the order in which threads are unblocked. When each thread unblocked as a result of a `pthread_cond_broadcast()` or `pthread_cond_signal()` returns from its call to `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, or `pthread_cond_wait()`, the thread shall own the mutex with which it called `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, or `pthread_cond_wait()`. The thread(s) that are unblocked shall contend for the mutex according to the scheduling policy (if applicable), and as if each had called `pthread_mutex_lock()`.

The `pthread_cond_broadcast()` or `pthread_cond_signal()` functions may be called by a thread whether or not it currently owns the mutex that threads calling `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, or `pthread_cond_wait()` have associated with the condition variable during their waits; however, if predictable scheduling behavior is required, then that mutex shall be locked by the thread calling `pthread_cond_broadcast()` or `pthread_cond_signal()`.

The `pthread_cond_broadcast()` and `pthread_cond_signal()` functions shall have no effect if they determine that there are no threads blocked on `cond`.

The behavior is undefined if the value specified by the `cond` argument to `pthread_cond_broadcast()` or `pthread_cond_signal()` does not refer to an initialized condition variable.

## RETURN VALUE

If successful, the `pthread_cond_broadcast()` and `pthread_cond_signal()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

## ERRORS

These functions shall not return an error code of `[EINTR]`.

---

## EXAMPLES

None.

## APPLICATION USAGE

The `pthread_cond_broadcast()` function is used whenever the shared-variable state has been changed in a way that more than one thread can proceed with its task. Consider a single producer/multiple consumer problem, where the producer can insert multiple items on a list that is accessed one item at a time by the consumers. By calling the `pthread_cond_broadcast()` function, the producer would notify all consumers that might be waiting, and thereby the application would receive more throughput on a multi-processor. In addition, `pthread_cond_broadcast()` makes it easier to implement a read-write lock. The `pthread_cond_broadcast()` function is needed in order to wake up all waiting readers when a writer releases its lock. Finally, the two-phase commit algorithm can use this broadcast function to notify all clients of an impending transaction commit.

It is not safe to use the `pthread_cond_signal()` function in a signal handler that is invoked asynchronously. Even if it were safe, there would still be a race between the test of the Boolean `pthread_cond_wait()` that could not be efficiently eliminated.

Mutexes and condition variables are thus not suitable for releasing a waiting thread by signaling from code running in a signal handler.

## RATIONALE

If an implementation detects that the value specified by the `cond` argument to `pthread_cond_broadcast()` or `pthread_cond_signal()` does not refer to an initialized condition variable, it is recommended that the function should fail and report an `[EINVAL]` error.

### Multiple Awakenings by Condition Signal

On a multi-processor, it may be impossible for an implementation of `pthread_cond_signal()` to avoid the unblocking of more than one thread blocked on a condition variable. For example, consider the following partial implementation of `pthread_cond_wait()` and `pthread_cond_signal()`, executed by two threads in the order given. One thread is trying to wait on the condition variable, another is concurrently executing `pthread_cond_signal()`, while a third thread is already waiting.

```
pthread_cond_wait(mutex, cond):
    value = cond->value;          /* 1 */
    pthread_mutex_unlock(mutex);   /* 2 */
    pthread_mutex_lock(cond->mutex); /* 10 */
    if (value == cond->value) {    /* 11 */
        me->next_cond = cond->waiter;
        cond->waiter = me;
        pthread_mutex_unlock(cond->mutex);
        unable_to_run(me);
    } else
        pthread_mutex_unlock(cond->mutex); /* 12 */
    pthread_mutex_lock(mutex);      /* 13 */

pthread_cond_signal(cond):
    pthread_mutex_lock(cond->mutex); /* 3 */
    cond->value++;                /* 4 */
    if (cond->waiter) {           /* 5 */
        sleeper = cond->waiter;    /* 6 */
        cond->waiter = sleeper->next_cond; /* 7 */
        able_to_run(sleeper);      /* 8 */
    }
    pthread_mutex_unlock(cond->mutex); /* 9 */
```

The effect is that more than one thread can return from its call to `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, or `pthread_cond_wait()` as a result of one call to `pthread_cond_signal()`.

This effect is called "spurious wakeup". Note that the situation is self-correcting in that the number of threads that are so awakened is finite; for example, the next thread to call `pthread_cond_wait()` after the sequence of events above blocks.

While this problem could be resolved, the loss of efficiency for a fringe condition that occurs only rarely is unacceptable, especially given that one has to check the predicate associated with a condition variable anyway. Correcting this problem would unnecessarily reduce the degree of concurrency in this basic building block for all higher-level synchronization operations.

An added benefit of allowing spurious wakeups is that applications are forced to code a predicate-testing-loop around the condition wait. This also makes the application tolerate superfluous condition broadcasts or signals on the same condition variable that may be coded in some other part of the application. The resulting applications are thus more robust. Therefore, POSIX.1-2024 explicitly documents that spurious wakeups may occur.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_cond_clockwait()`
- `pthread_cond_destroy()`
- XBD 4.15.2 Memory Synchronization
- `<pthread.h>`

## CHANGE HISTORY

### First released in Issue 5

Included for alignment with the POSIX Threads Extension.

### Issue 6

The `pthread_cond_broadcast()` and `pthread_cond_signal()` functions are marked as part of the Threads option.

The APPLICATION USAGE section is added.

## Issue 7

The `pthread_cond_broadcast()` and `pthread_cond_signal()` functions are moved from the Threads option to the Base.

The `[EINVAL]` error for an uninitialized condition variable is removed; this condition results in undefined behavior.

## Issue 8

Austin Group Defect 609 is applied, adding atomicity requirements.

Austin Group Defect 1216 is applied, adding `pthread_cond_clockwait()`.

---

## 1.133. `pthread_cond_clockwait`, `pthread_cond_timedwait`, `pthread_cond_wait` — wait on a condition

### SYNOPSIS

```
#include <pthread.h>

int pthread_cond_clockwait(pthread_cond_t *restrict cond,
                           pthread_mutex_t *restrict mutex,
                           clockid_t clock_id,
                           const struct timespec *restrict abstime)

int pthread_cond_timedwait(pthread_cond_t *restrict cond,
                           pthread_mutex_t *restrict mutex,
                           const struct timespec *restrict abstime)

int pthread_cond_wait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex);
```

### DESCRIPTION

The `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, and `pthread_cond_wait()` functions shall block on a condition variable. The application shall ensure that these functions are called with `mutex` locked by the calling thread; otherwise, an error (for PTHREAD\_MUTEX\_ERRORCHECK and robust mutexes) or undefined behavior (for other mutexes) results.

These functions atomically release `mutex` and cause the calling thread to block on the condition variable `cond`; atomically here means "atomically with respect to access by another thread to the mutex and then the condition variable". That is, if another thread is able to acquire the mutex after the about-to-block thread has released it, then a subsequent call to `pthread_cond_broadcast()` or `pthread_cond_signal()` in that thread shall behave as if it were issued after the about-to-block thread has blocked.

Upon successful return, the mutex shall have been locked and shall be owned by the calling thread.

If `mutex` is a robust mutex where an owner terminated while holding the lock and the state is recoverable, the mutex shall be acquired even though the function returns [EOWNERDEAD].

When using condition variables there is always a Boolean predicate involving shared variables associated with each condition wait that is true if the thread should proceed. Spurious wakeups from the `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, or `pthread_cond_wait()` functions may occur. Since the return from `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, or `pthread_cond_wait()` does not imply anything about the value of this predicate, the predicate should be re-evaluated upon such return.

When a thread waits on a condition variable, having specified a particular mutex to the `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, or `pthread_cond_wait()` operation, a dynamic binding is formed between that mutex and condition variable that remains in effect as long as at least one thread is blocked on the condition variable. During this time, the effect of an attempt by any thread to wait on that condition variable using a different mutex is undefined. Once all waiting threads have been unblocked (as by the `pthread_cond_broadcast()` operation), the next wait operation on that condition variable shall form a new dynamic binding with the mutex specified by that wait operation. Even though the dynamic binding between condition variable and mutex may be removed or replaced between the time a thread is unblocked from a wait on the condition variable and the time that it returns to the caller or begins cancellation cleanup, the unblocked thread shall always re-acquire the mutex specified in the condition wait operation call from which it is returning.

A condition wait (whether timed or not) is a cancellation point. When the cancelability type of a thread is set to `PTHREAD_CANCEL_DEFERRED`, a side-effect of acting upon a cancellation request while in a condition wait is that the mutex is (in effect) re-acquired before calling the first cancellation cleanup handler. The effect is as if the thread were unblocked, allowed to execute up to the point of returning from the call to `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, or `pthread_cond_wait()`, but at that point notices the cancellation request and, instead of returning to the caller, starts the thread cancellation activities, which includes calling cancellation cleanup handlers.

A thread that has been unblocked because it has been canceled while blocked in a call to `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, or `pthread_cond_wait()` shall not consume any condition signal that may be directed concurrently at the condition variable if there are other threads blocked on the condition variable.

The `pthread_cond_clockwait()` function shall be equivalent to `pthread_cond_wait()`, except that an error is returned if the absolute time specified by `abstime` as measured against the clock indicated by `clock_id` passes (that is, the current time measured by that clock equals or exceeds `abstime`) before the condition `cond` is signaled or broadcasted, or if the

absolute time specified by `abstime` has already been passed at the time of the call. Implementations shall support passing `CLOCK_REALTIME` and `CLOCK_MONOTONIC` to `pthread_cond_clockwait()` as the `clock_id` argument. When such timeouts occur, `pthread_cond_clockwait()` shall nonetheless release and re-acquire the mutex referenced by `mutex`, and may consume a condition signal directed concurrently at the condition variable.

The `pthread_cond_timedwait()` function shall be equivalent to `pthread_cond_clockwait()`, except that it lacks the `clock_id` argument. The clock to measure `abstime` against shall instead come from the condition variable's clock attribute which can be set by `pthread_condattr_setclock()` prior to the condition variable's creation. If no clock attribute has been set, the default shall be `CLOCK_REALTIME`.

If a signal is delivered to a thread waiting for a condition variable, upon return from the signal handler the thread resumes waiting for the condition variable as if it was not interrupted, or it shall return zero due to spurious wakeup.

The behavior is undefined if the value specified by the `cond` or `mutex` argument to these functions does not refer to an initialized condition variable or an initialized mutex object, respectively.

## RETURN VALUE

Except for `[ETIMEDOUT]`, `[ENOTRECOVERABLE]`, and `[EOWNERDEAD]`, all these error checks shall act as if they were performed immediately at the beginning of processing for the function and shall cause an error return, in effect, prior to modifying the state of the mutex specified by `mutex` or the condition variable specified by `cond`.

Upon successful completion, a value of zero shall be returned; otherwise, an error number shall be returned to indicate the error.

## ERRORS

These functions shall fail if:

- **[EAGAIN]**

The mutex is a robust mutex and the system resources available for robust mutexes owned would be exceeded.

- **[ENOTRECOVERABLE]**

The state protected by the mutex is not recoverable.

- **[EOWNERDEAD]**

The mutex is a robust mutex and the process containing the previous owning thread terminated while holding the mutex lock. The mutex lock shall be acquired by the calling thread and it is up to the new owner to make the state consistent.

- **[EPERM]**

The mutex type is PTHREAD\_MUTEX\_ERRORCHECK or the mutex is a robust mutex, and the current thread does not own the mutex.

The `pthread_cond_clockwait()` and `pthread_cond_timedwait()` functions shall fail if:

- **[ETIMEDOUT]**

The time specified by `abstime` has passed.

- **[EINVAL]**

The `abstime` argument specified a nanosecond value less than zero or greater than or equal to 1000 million, or the `clock_id` argument passed to `pthread_cond_clockwait()` is invalid or not supported.

These functions may fail if:

- **[EOWNERDEAD]**

The mutex is a robust mutex and the previous owning thread terminated while holding the mutex lock. The mutex lock shall be acquired by the calling thread and it is up to the new owner to make the state consistent.

These functions shall not return an error code of [EINTR].

## EXAMPLES

None.

## APPLICATION USAGE

Applications that have assumed that non-zero return values are errors will need updating for use with robust mutexes, since a valid return for a thread acquiring a mutex which is protecting a currently inconsistent state is [EOWNERDEAD]. Applications that do not check the error returns, due to ruling out the possibility of such errors arising, should not use robust mutexes. If an application is supposed to work with normal and robust mutexes, it should check all return values for error conditions and if necessary take appropriate action.

# RATIONALE

If an implementation detects that the value specified by the `cond` argument to `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, or `pthread_cond_wait()` does not refer to an initialized condition variable, or detects that the value specified by the `mutex` argument does not refer to an initialized mutex object, it is recommended that the function should fail and report an [EINVAL] error.

## Condition Wait Semantics

It is important to note that when `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, and `pthread_cond_wait()` return without error, the associated predicate may still be false. Similarly, when `pthread_cond_clockwait()` or `pthread_cond_timedwait()` returns with the timeout error, the associated predicate may be true due to an unavoidable race between the expiration of the timeout and the predicate state change.

The application needs to recheck the predicate on any return because it cannot be sure there is another thread waiting on the thread to handle the signal, and if there is not then the signal is lost. The burden is on the application to check the predicate.

Some implementations, particularly on a multi-processor, may sometimes cause multiple threads to wake up when the condition variable is signaled simultaneously on different processors.

In general, whenever a condition wait returns, the thread has to re-evaluate the predicate associated with the condition wait to determine whether it can safely proceed, should wait again, or should declare a timeout. A return from the wait does not imply that the associated predicate is either true or false.

It is thus recommended that a condition wait be enclosed in the equivalent of a "while loop" that checks the predicate.

## Timed Wait Semantics

An absolute time measure was chosen for specifying the timeout parameter for two reasons. First, a relative time measure can be easily implemented on top of a function that specifies absolute time, but there is a race condition associated with specifying an absolute timeout on top of a function that specifies relative timeouts. For example, assume that `clock_gettime()` returns the current time and `cond_relative_timed_wait()` uses relative timeouts:

```
clock_gettime(CLOCK_REALTIME, &now)
reltime = sleep_til_this_absolute_time - now;
cond_relative_timed_wait(c, m, &reltime);
```

If the thread is preempted between the first statement and the last statement, the thread blocks for too long. Blocking, however, is irrelevant if an absolute timeout is used. An absolute timeout also need not be recomputed if it is used multiple times in a loop, such as that enclosing a condition wait.

For cases when the system clock is advanced discontinuously by an operator, it is expected that implementations process any timed wait expiring at an intervening time as if that time had actually occurred.

## Choice of Clock

Care should be taken to decide which clock is most appropriate when waiting with a timeout. The system clock `CLOCK_REALTIME`, as used by default with `pthread_cond_timedwait()`, may be subject to jumps forwards and backwards in order to correct it against actual time. `CLOCK_MONOTONIC` is guaranteed not to jump backwards and must also advance in real time, so using it via `pthread_cond_clockwait()` or `pthread_condattr_setclock()` may be more appropriate.

## Cancellation and Condition Wait

A condition wait, whether timed or not, is a cancellation point. That is, the functions `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, and `pthread_cond_wait()` are points where a pending (or concurrent) cancellation request is noticed. The reason for this is that an indefinite wait is possible at these points—whatever event is being waited for, even if the program is totally correct, might never occur; for example, some input data being awaited might never be sent. By making condition wait a cancellation point, the thread can be canceled and perform its cancellation cleanup handler even though it may be stuck in some indefinite wait.

A side-effect of acting on a cancellation request while a thread is blocked on a condition variable is to re-acquire the mutex before calling any of the cancellation cleanup handlers. This is done in order to ensure that the cancellation cleanup handler is executed in the same state as the critical code that lies both before and after the call to the condition wait function. This rule is also required when interfacing to POSIX threads from languages, such as Ada or C++, which may choose to map cancellation onto a language exception; this rule ensures that each exception handler guarding a critical section can always safely depend upon the fact that the associated mutex has already been locked regardless of exactly

where within the critical section the exception was raised. Without this rule, there would not be a uniform rule that exception handlers could follow regarding the lock, and so coding would become very cumbersome.

Therefore, since some statement has to be made regarding the state of the lock when a cancellation is delivered during a wait, a definition has been chosen that makes application coding most convenient and error free.

When acting on a cancellation request while a thread is blocked on a condition variable, the implementation is required to ensure that the thread does not consume any condition signals directed at that condition variable if there are any other threads waiting on that condition variable. This rule is specified in order to avoid deadlock conditions that could occur if these two independent requests (one acting on a thread and the other acting on the condition variable) were not processed independently.

## Performance of Mutexes and Condition Variables

Mutexes are expected to be locked only for a few instructions. This practice is almost automatically enforced by the desire of programmers to avoid long serial regions of execution (which would reduce total effective parallelism).

When using mutexes and condition variables, one tries to ensure that the usual case is to lock the mutex, access shared data, and unlock the mutex. Waiting on a condition variable should be a relatively rare situation. For example, when implementing a read-write lock, code that acquires a read-lock typically needs only to increment the count of readers (under mutual-exclusion) and return. The calling thread would actually wait on the condition variable only when there is already an active writer. So the efficiency of a synchronization operation is bounded by the cost of mutex lock/unlock and not by condition wait. Note that in the usual case there is no context switch.

This is not to say that the efficiency of condition waiting is unimportant. Since there needs to be at least one context switch per Ada rendezvous, the efficiency of waiting on a condition variable is important. The cost of waiting on a condition variable should be little more than the minimal cost for a context switch plus the time to unlock and lock the mutex.

## Features of Mutexes and Condition Variables

It had been suggested that the mutex acquisition and release be decoupled from condition wait. This was rejected because it is the combined nature of the operation that, in fact, facilitates realtime implementations. Those implementations can atomically move a high-priority thread between the condition variable and the mutex in a manner that is transparent to the caller. This can prevent extra context

switches and provide more deterministic acquisition of a mutex when the waiting thread is signaled. Thus, fairness and priority issues can be dealt with directly by the scheduling discipline. Furthermore, the current condition wait operation matches existing practice.

## Scheduling Behavior of Mutexes and Condition Variables

Synchronization primitives that attempt to interfere with scheduling policy by specifying an ordering rule are considered undesirable. Threads waiting on mutexes and condition variables are selected to proceed in an order dependent upon the scheduling policy rather than in some fixed order (for example, FIFO or priority). Thus, the scheduling policy determines which thread(s) are awakened and allowed to proceed.

### Timed Condition Wait

The `pthread_cond_clockwait()` and `pthread_cond_timedwait()` functions allow an application to give up waiting for a particular condition after a given amount of time. An example follows:

```
(void) pthread_mutex_lock(&t.mn);
    t.waiters++;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    ts.tv_sec += 5;
    rc = 0;
    while (! mypredicate(&t) && rc == 0)
        rc = pthread_cond_clockwait(&t.cond, &t.mn,
                                    CLOCK_MONOTONIC, &ts);
    t.waiters--;
    if (rc == 0 || mypredicate(&t))
        setmystate(&t);
(void) pthread_mutex_unlock(&t.mn);
```

By making the timeout parameter absolute, it does not need to be recomputed each time the program checks its blocking predicate. If the timeout was relative, it would have to be recomputed before each call. This would be especially difficult since such code would need to take into account the possibility of extra wakeups that result from extra broadcasts or signals on the condition variable that occur before either the predicate is true or the timeout is due. Using `CLOCK_MONOTONIC` rather than `CLOCK_REALTIME` means that the timeout is not influenced by the system clock being changed.

# FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_cond_broadcast()`
- XBD 4.15.2 Memory Synchronization, `<pthread.h>`

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

### Issue 6

The `pthread_cond_timedwait()` and `pthread_cond_wait()` functions are marked as part of the Threads option.

The Open Group Corrigendum U021/9 is applied, correcting the prototype for the `pthread_cond_wait()` function.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding semantics for the Clock Selection option.

The ERRORS section has an additional case for [EPERM] in response to IEEE PASC Interpretation 1003.1c #28.

The `restrict` keyword is added to the `pthread_cond_timedwait()` and `pthread_cond_wait()` prototypes for alignment with the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/89 is applied, updating the DESCRIPTION for consistency with the `pthread_cond_destroy()` function that states it is safe to destroy an initialized condition variable upon which no threads are currently blocked.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/90 is applied, updating words in the DESCRIPTION from "the cancelability enable state" to "the cancelability type".

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/91 is applied, updating the ERRORS section to remove the error case related to `abstime` from the `pthread_cond_wait()` function, and to make the error case related to

`abstime` mandatory for `pthread_cond_timedwait()` for consistency with other functions.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/92 is applied, adding a new paragraph to the RATIONALE section stating that an application should check the predicate on any return from this function.

## Issue 7

SD5-XSH-ERN-44 is applied, changing the definition of the "shall fail" case of the [EINVAL] error.

Changes are made from The Open Group Technical Standard, 2006, Extended API Set Part 3.

The `pthread_cond_timedwait()` and `pthread_cond_wait()` functions are moved from the Threads option to the Base.

The [EINVAL] error for an uninitialized condition variable or uninitialized mutex object is removed; this condition results in undefined behavior.

The [EPERM] error is revised and moved to the "shall fail" list of error conditions for the `pthread_cond_timedwait()` function.

The DESCRIPTION is updated to clarify the behavior when `mutex` is a robust mutex.

The ERRORS section is updated to include "shall fail" cases for PTHREAD\_MUTEX\_ERRORCHECK mutexes.

The DESCRIPTION is rewritten to clarify that undefined behavior occurs only for mutexes where the [EPERM] error is not mandated.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0456 [91,286,437] and XSH/TC1-2008/0457 [239] are applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0271 [749] is applied.

## Issue 8

Austin Group Defect 354 is applied, adding the [EAGAIN] error.

Austin Group Defect 1162 is applied, changing "an error code" to "[EOWNERDEAD].

Austin Group Defects 1216 and 1485 are applied, adding `pthread_cond_clockwait()`.

---

## 1.134. `pthread_cond_clockwait`, `pthread_cond_timedwait`, `pthread_cond_wait` — wait on a condition

### SYNOPSIS

```
#include <pthread.h>

int pthread_cond_clockwait(pthread_cond_t *restrict cond,
                           pthread_mutex_t *restrict mutex,
                           clockid_t clock_id,
                           const struct timespec *restrict abstime)

int pthread_cond_timedwait(pthread_cond_t *restrict cond,
                           pthread_mutex_t *restrict mutex,
                           const struct timespec *restrict abstime)

int pthread_cond_wait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex);
```

### DESCRIPTION

The `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, and `pthread_cond_wait()` functions shall block on a condition variable. The application shall ensure that these functions are called with `mutex` locked by the calling thread; otherwise, an error (for PTHREAD\_MUTEX\_ERRORCHECK and robust mutexes) or undefined behavior (for other mutexes) results.

These functions atomically release `mutex` and cause the calling thread to block on the condition variable `cond`; atomically here means "atomically with respect to access by another thread to the mutex and then the condition variable". That is, if another thread is able to acquire the mutex after the about-to-block thread has released it, then a subsequent call to `pthread_cond_broadcast()` or `pthread_cond_signal()` in that thread shall behave as if it were issued after the about-to-block thread has blocked.

Upon successful return, the mutex shall have been locked and shall be owned by the calling thread.

If `mutex` is a robust mutex where an owner terminated while holding the lock and the state is recoverable, the mutex shall be acquired even though the function returns [EOWNERDEAD].

When using condition variables there is always a Boolean predicate involving shared variables associated with each condition wait that is true if the thread should proceed. Spurious wakeups from the `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, or `pthread_cond_wait()` functions may occur. Since the return from `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, or `pthread_cond_wait()` does not imply anything about the value of this predicate, the predicate should be re-evaluated upon such return.

When a thread waits on a condition variable, having specified a particular mutex to the `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, or `pthread_cond_wait()` operation, a dynamic binding is formed between that mutex and condition variable that remains in effect as long as at least one thread is blocked on the condition variable. During this time, the effect of an attempt by any thread to wait on that condition variable using a different mutex is undefined. Once all waiting threads have been unblocked (as by the `pthread_cond_broadcast()` operation), the next wait operation on that condition variable shall form a new dynamic binding with the mutex specified by that wait operation. Even though the dynamic binding between condition variable and mutex may be removed or replaced between the time a thread is unblocked from a wait on the condition variable and the time that it returns to the caller or begins cancellation cleanup, the unblocked thread shall always re-acquire the mutex specified in the condition wait operation call from which it is returning.

A condition wait (whether timed or not) is a cancellation point. When the cancelability type of a thread is set to `PTHREAD_CANCEL_DEFERRED`, a side-effect of acting upon a cancellation request while in a condition wait is that the mutex is (in effect) re-acquired before calling the first cancellation cleanup handler. The effect is as if the thread were unblocked, allowed to execute up to the point of returning from the call to `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, or `pthread_cond_wait()`, but at that point notices the cancellation request and, instead of returning to the caller, starts the thread cancellation activities, which includes calling cancellation cleanup handlers.

A thread that has been unblocked because it has been canceled while blocked in a call to `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, or `pthread_cond_wait()` shall not consume any condition signal that may be directed concurrently at the condition variable if there are other threads blocked on the condition variable.

The `pthread_cond_clockwait()` function shall be equivalent to `pthread_cond_wait()`, except that an error is returned if the absolute time specified by `abstime` as measured against the clock indicated by `clock_id` passes (that is, the current time measured by that clock equals or exceeds `abstime`) before the condition `cond` is signaled or broadcasted, or if the

absolute time specified by `abstime` has already been passed at the time of the call. Implementations shall support passing `CLOCK_REALTIME` and `CLOCK_MONOTONIC` to `pthread_cond_clockwait()` as the `clock_id` argument. When such timeouts occur, `pthread_cond_clockwait()` shall nonetheless release and re-acquire the mutex referenced by `mutex`, and may consume a condition signal directed concurrently at the condition variable.

The `pthread_cond_timedwait()` function shall be equivalent to `pthread_cond_clockwait()`, except that it lacks the `clock_id` argument. The clock to measure `abstime` against shall instead come from the condition variable's clock attribute which can be set by `pthread_condattr_setclock()` prior to the condition variable's creation. If no clock attribute has been set, the default shall be `CLOCK_REALTIME`.

If a signal is delivered to a thread waiting for a condition variable, upon return from the signal handler the thread resumes waiting for the condition variable as if it was not interrupted, or it shall return zero due to spurious wakeup.

The behavior is undefined if the value specified by the `cond` or `mutex` argument to these functions does not refer to an initialized condition variable or an initialized mutex object, respectively.

## RETURN VALUE

Except for `[ETIMEDOUT]`, `[ENOTRECOVERABLE]`, and `[EOWNERDEAD]`, all these error checks shall act as if they were performed immediately at the beginning of processing for the function and shall cause an error return, in effect, prior to modifying the state of the mutex specified by `mutex` or the condition variable specified by `cond`.

Upon successful completion, a value of zero shall be returned; otherwise, an error number shall be returned to indicate the error.

## ERRORS

These functions shall fail if:

- **[EAGAIN]**
- The mutex is a robust mutex and the system resources available for robust mutexes owned would be exceeded.
- **[ENOTRECOVERABLE]**
- The state protected by the mutex is not recoverable.

- **[EOWNERDEAD]**

- The mutex is a robust mutex and the process containing the previous owning thread terminated while holding the mutex lock. The mutex lock shall be acquired by the calling thread and it is up to the new owner to make the state consistent.

- **[EPERM]**

- The mutex type is PTHREAD\_MUTEX\_ERRORCHECK or the mutex is a robust mutex, and the current thread does not own the mutex.

The `pthread_cond_clockwait()` and `pthread_cond_timedwait()` functions shall fail if:

- **[ETIMEDOUT]**

- The time specified by `abstime` has passed.

- **[EINVAL]**

- The `abstime` argument specified a nanosecond value less than zero or greater than or equal to 1000 million, or the `clock_id` argument passed to `pthread_cond_clockwait()` is invalid or not supported.

These functions may fail if:

- **[EOWNERDEAD]**

- The mutex is a robust mutex and the previous owning thread terminated while holding the mutex lock. The mutex lock shall be acquired by the calling thread and it is up to the new owner to make the state consistent.

These functions shall not return an error code of [EINTR].

## EXAMPLES

None.

## APPLICATION USAGE

Applications that have assumed that non-zero return values are errors will need updating for use with robust mutexes, since a valid return for a thread acquiring a mutex which is protecting a currently inconsistent state is [EOWNERDEAD]. Applications that do not check the error returns, due to ruling out the possibility of such errors arising, should not use robust mutexes. If an application is supposed to

work with normal and robust mutexes, it should check all return values for error conditions and if necessary take appropriate action.

## RATIONALE

If an implementation detects that the value specified by the `cond` argument to `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, or `pthread_cond_wait()` does not refer to an initialized condition variable, or detects that the value specified by the `mutex` argument does not refer to an initialized mutex object, it is recommended that the function should fail and report an [EINVAL] error.

### Condition Wait Semantics

It is important to note that when `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, and `pthread_cond_wait()` return without error, the associated predicate may still be false. Similarly, when `pthread_cond_clockwait()` or `pthread_cond_timedwait()` returns with the timeout error, the associated predicate may be true due to an unavoidable race between the expiration of the timeout and the predicate state change.

The application needs to recheck the predicate on any return because it cannot be sure there is another thread waiting on the thread to handle the signal, and if there is not then the signal is lost. The burden is on the application to check the predicate.

Some implementations, particularly on a multi-processor, may sometimes cause multiple threads to wake up when the condition variable is signaled simultaneously on different processors.

In general, whenever a condition wait returns, the thread has to re-evaluate the predicate associated with the condition wait to determine whether it can safely proceed, should wait again, or should declare a timeout. A return from the wait does not imply that the associated predicate is either true or false.

It is thus recommended that a condition wait be enclosed in the equivalent of a "while loop" that checks the predicate.

### Timed Wait Semantics

An absolute time measure was chosen for specifying the timeout parameter for two reasons. First, a relative time measure can be easily implemented on top of a function that specifies absolute time, but there is a race condition associated with specifying an absolute timeout on top of a function that specifies relative timeouts.

For example, assume that `clock_gettime()` returns the current time and `cond_relative_timed_wait()` uses relative timeouts:

```
clock_gettime(CLOCK_REALTIME, &now)
reltime = sleep_til_this_absolute_time - now;
cond_relative_timed_wait(c, m, &reltime);
```

If the thread is preempted between the first statement and the last statement, the thread blocks for too long. Blocking, however, is irrelevant if an absolute timeout is used. An absolute timeout also need not be recomputed if it is used multiple times in a loop, such as that enclosing a condition wait.

For cases when the system clock is advanced discontinuously by an operator, it is expected that implementations process any timed wait expiring at an intervening time as if that time had actually occurred.

## Choice of Clock

Care should be taken to decide which clock is most appropriate when waiting with a timeout. The system clock `CLOCK_REALTIME`, as used by default with `pthread_cond_timedwait()`, may be subject to jumps forwards and backwards in order to correct it against actual time. `CLOCK_MONOTONIC` is guaranteed not to jump backwards and must also advance in real time, so using it via `pthread_cond_clockwait()` or `pthread_condattr_setclock()` may be more appropriate.

## Cancellation and Condition Wait

A condition wait, whether timed or not, is a cancellation point. That is, the functions `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, and `pthread_cond_wait()` are points where a pending (or concurrent) cancellation request is noticed. The reason for this is that an indefinite wait is possible at these points—whatever event is being waited for, even if the program is totally correct, might never occur; for example, some input data being awaited might never be sent. By making condition wait a cancellation point, the thread can be canceled and perform its cancellation cleanup handler even though it may be stuck in some indefinite wait.

A side-effect of acting on a cancellation request while a thread is blocked on a condition variable is to re-acquire the mutex before calling any of the cancellation cleanup handlers. This is done in order to ensure that the cancellation cleanup handler is executed in the same state as the critical code that lies both before and after the call to the condition wait function. This rule is also required when interfacing to POSIX threads from languages, such as Ada or C++, which may choose to map cancellation onto a language exception; this rule ensures that each

exception handler guarding a critical section can always safely depend upon the fact that the associated mutex has already been locked regardless of exactly where within the critical section the exception was raised. Without this rule, there would not be a uniform rule that exception handlers could follow regarding the lock, and so coding would become very cumbersome.

Therefore, since some statement has to be made regarding the state of the lock when a cancellation is delivered during a wait, a definition has been chosen that makes application coding most convenient and error free.

When acting on a cancellation request while a thread is blocked on a condition variable, the implementation is required to ensure that the thread does not consume any condition signals directed at that condition variable if there are any other threads waiting on that condition variable. This rule is specified in order to avoid deadlock conditions that could occur if these two independent requests (one acting on a thread and the other acting on the condition variable) were not processed independently.

## Performance of Mutexes and Condition Variables

Mutexes are expected to be locked only for a few instructions. This practice is almost automatically enforced by the desire of programmers to avoid long serial regions of execution (which would reduce total effective parallelism).

When using mutexes and condition variables, one tries to ensure that the usual case is to lock the mutex, access shared data, and unlock the mutex. Waiting on a condition variable should be a relatively rare situation. For example, when implementing a read-write lock, code that acquires a read-lock typically needs only to increment the count of readers (under mutual-exclusion) and return. The calling thread would actually wait on the condition variable only when there is already an active writer. So the efficiency of a synchronization operation is bounded by the cost of mutex lock/unlock and not by condition wait. Note that in the usual case there is no context switch.

This is not to say that the efficiency of condition waiting is unimportant. Since there needs to be at least one context switch per Ada rendezvous, the efficiency of waiting on a condition variable is important. The cost of waiting on a condition variable should be little more than the minimal cost for a context switch plus the time to unlock and lock the mutex.

## Features of Mutexes and Condition Variables

It had been suggested that the mutex acquisition and release be decoupled from condition wait. This was rejected because it is the combined nature of the operation that, in fact, facilitates realtime implementations. Those implementations

can atomically move a high-priority thread between the condition variable and the mutex in a manner that is transparent to the caller. This can prevent extra context switches and provide more deterministic acquisition of a mutex when the waiting thread is signaled. Thus, fairness and priority issues can be dealt with directly by the scheduling discipline. Furthermore, the current condition wait operation matches existing practice.

## Scheduling Behavior of Mutexes and Condition Variables

Synchronization primitives that attempt to interfere with scheduling policy by specifying an ordering rule are considered undesirable. Threads waiting on mutexes and condition variables are selected to proceed in an order dependent upon the scheduling policy rather than in some fixed order (for example, FIFO or priority). Thus, the scheduling policy determines which thread(s) are awakened and allowed to proceed.

## Timed Condition Wait

The `pthread_cond_clockwait()` and `pthread_cond_timedwait()` functions allow an application to give up waiting for a particular condition after a given amount of time. An example follows:

```
(void) pthread_mutex_lock(&t.mn);
    t.waiters++;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    ts.tv_sec += 5;
    rc = 0;
    while (! mypredicate(&t) && rc == 0)
        rc = pthread_cond_clockwait(&t.cond, &t.mn,
                                    CLOCK_MONOTONIC, &ts);
    t.waiters--;
    if (rc == 0 || mypredicate(&t))
        setmystate(&t);
(void) pthread_mutex_unlock(&t.mn);
```

By making the timeout parameter absolute, it does not need to be recomputed each time the program checks its blocking predicate. If the timeout was relative, it would have to be recomputed before each call. This would be especially difficult since such code would need to take into account the possibility of extra wakeups that result from extra broadcasts or signals on the condition variable that occur before either the predicate is true or the timeout is due. Using `CLOCK_MONOTONIC` rather than `CLOCK_REALTIME` means that the timeout is not influenced by the system clock being changed.

# FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_cond_broadcast()`
- XBD 4.15.2 Memory Synchronization, `<pthread.h>`

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

### Issue 6

The `pthread_cond_timedwait()` and `pthread_cond_wait()` functions are marked as part of the Threads option.

The Open Group Corrigendum U021/9 is applied, correcting the prototype for the `pthread_cond_wait()` function.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding semantics for the Clock Selection option.

The ERRORS section has an additional case

---

## 1.135. pthread\_condattr\_destroy, pthread\_condattr\_init

---

### SYNOPSIS

```
#include <pthread.h>

int pthread_condattr_destroy(pthread_condattr_t *attr);
int pthread_condattr_init(pthread_condattr_t *attr);
```

### DESCRIPTION

The `pthread_condattr_destroy()` function shall destroy a condition variable attributes object; the object becomes, in effect, uninitialized. An implementation may cause `pthread_condattr_destroy()` to set the object referenced by attr to an invalid value. A destroyed attr attributes object can be reinitialized using `pthread_condattr_init()`; the results of otherwise referencing the object after it has been destroyed are undefined.

The `pthread_condattr_init()` function shall initialize a condition variable attributes object attr with the default value for all of the attributes defined by the implementation.

Results are undefined if `pthread_condattr_init()` is called specifying an already initialized attr attributes object.

After a condition variable attributes object has been used to initialize one or more condition variables, any function affecting the attributes object (including destruction) shall not affect any previously initialized condition variables.

This volume of POSIX.1-2024 requires two attributes, the **clock** attribute and the **process-shared** attribute.

Additional attributes, their default values, and the names of the associated functions to get and set those attribute values are implementation-defined.

The behavior is undefined if the value specified by the attr argument to `pthread_condattr_destroy()` does not refer to an initialized condition variable attributes object.

### RETURN VALUE

If successful, the `pthread_condattr_destroy()` and `pthread_condattr_init()` functions shall return zero; otherwise, an error

number shall be returned to indicate the error.

## ERRORS

The `pthread_condattr_init()` function shall fail if:

- **[ENOMEM]**
- Insufficient memory exists to initialize the condition variable attributes object.

These functions shall not return an error code of [EINTR].

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

A **process-shared** attribute has been defined for condition variables for the same reason it has been defined for mutexes.

If an implementation detects that the value specified by the attr argument to `pthread_condattr_destroy()` does not refer to an initialized condition variable attributes object, it is recommended that the function should fail and report an [EINVAL] error.

See also `pthread_attr_destroy()` and `pthread_mutex_destroy()`.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_attr_destroy()`
- `pthread_cond_destroy()`
- `pthread_condattr_getpshared()`
- `pthread_create()`
- `pthread_mutex_destroy()`
- XBD

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

### Issue 6

The `pthread_condattr_destroy()` and `pthread_condattr_init()` functions are marked as part of the Threads option.

### Issue 7

The `pthread_condattr_destroy()` and `pthread_condattr_init()` functions are moved from the Threads option to the Base.

The [EINVAL] error for an uninitialized condition variable attributes object is removed; this condition results in undefined behavior.

## 1.136. `pthread_condattr_getclock`, `pthread_condattr_setclock`

### SYNOPSIS

```
#include <pthread.h>

int pthread_condattr_getclock(const pthread_condattr_t *restrict attr,
                               clockid_t *restrict clock_id);

int pthread_condattr_setclock(pthread_condattr_t *attr,
                               clockid_t clock_id);
```

### DESCRIPTION

The `pthread_condattr_getclock()` function shall obtain the value of the *clock* attribute from the attributes object referenced by *attr*.

The `pthread_condattr_setclock()` function shall set the *clock* attribute in an initialized attributes object referenced by *attr*. If `pthread_condattr_setclock()` is called with a *clock\_id* argument that refers to a CPU-time clock, the call shall fail.

The *clock* attribute is the clock ID of the clock that shall be used to measure the timeout service of `pthread_cond_timedwait()`. The default value of the *clock* attribute shall refer to the system clock. The *clock* attribute shall have no effect on the `pthread_cond_clockwait()` function.

The behavior is undefined if the value specified by the *attr* argument to `pthread_condattr_getclock()` or `pthread_condattr_setclock()` does not refer to an initialized condition variable attributes object.

### RETURN VALUE

If successful, the `pthread_condattr_getclock()` function shall return zero and store the value of the *clock* attribute of *attr* into the object referenced by the *clock\_id* argument. Otherwise, an error number shall be returned to indicate the error.

If successful, the `pthread_condattr_setclock()` function shall return zero; otherwise, an error number shall be returned to indicate the error.

# ERRORS

The `pthread_condattr_setclock()` function may fail if:

- **[EINVAL]**
- The value specified by `clock_id` does not refer to a known clock, or is a CPU-time clock.

These functions shall not return an error code of [EINTR].

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

If an implementation detects that the value specified by the `attr` argument to `pthread_condattr_getclock()` or `pthread_condattr_setclock()` does not refer to an initialized condition variable attributes object, it is recommended that the function should fail and report an [EINVAL] error.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_cond_clockwait()`
- `pthread_cond_destroy()`
- `pthread_condattr_destroy()`

- `pthread_condattr_getpshared()`
- `pthread_create()`
- `pthread_mutex_destroy()`
- XBD `<pthread.h>`

## CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

### Issue 7

The `pthread_condattr_getclock()` and `pthread_condattr_setclock()` functions are moved from the Clock Selection option to the Base.

The [EINVAL] error for an uninitialized condition variable attributes object is removed; this condition results in undefined behavior.

### Issue 8

Austin Group Defect 1216 is applied, adding `pthread_cond_clockwait()`.

---

## 1.137. pthread\_condattr\_destroy, pthread\_condattr\_init

---

### SYNOPSIS

```
#include <pthread.h>

int pthread_condattr_destroy(pthread_condattr_t *attr);
int pthread_condattr_init(pthread_condattr_t *attr);
```

### DESCRIPTION

The `pthread_condattr_destroy()` function shall destroy a condition variable attributes object; the object becomes, in effect, uninitialized. An implementation may cause `pthread_condattr_destroy()` to set the object referenced by `attr` to an invalid value. A destroyed `attr` attributes object can be reinitialized using `pthread_condattr_init()`; the results of otherwise referencing the object after it has been destroyed are undefined.

The `pthread_condattr_init()` function shall initialize a condition variable attributes object `attr` with the default value for all of the attributes defined by the implementation.

Results are undefined if `pthread_condattr_init()` is called specifying an already initialized `attr` attributes object.

After a condition variable attributes object has been used to initialize one or more condition variables, any function affecting the attributes object (including destruction) shall not affect any previously initialized condition variables.

This volume of POSIX.1-2024 requires two attributes, the `clock` attribute and the `process-shared` attribute.

Additional attributes, their default values, and the names of the associated functions to get and set those attribute values are implementation-defined.

The behavior is undefined if the value specified by the `attr` argument to `pthread_condattr_destroy()` does not refer to an initialized condition variable attributes object.

### RETURN VALUE

If successful, the `pthread_condattr_destroy()` and `pthread_condattr_init()` functions shall return zero; otherwise, an error

number shall be returned to indicate the error.

## ERRORS

The `pthread_condattr_init()` function shall fail if:

- **[ENOMEM]**
- Insufficient memory exists to initialize the condition variable attributes object.

These functions shall not return an error code of **[EINTR]**.

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

A **process-shared** attribute has been defined for condition variables for the same reason it has been defined for mutexes.

If an implementation detects that the value specified by the `attr` argument to `pthread_condattr_destroy()` does not refer to an initialized condition variable attributes object, it is recommended that the function should fail and report an **[EINVAL]** error.

See also `pthread_attr_destroy()` and `pthread_mutex_destroy()`.

## FUTURE DIRECTIONS

None.

## SEE ALSO

`pthread_attr_destroy()` ,  
`pthread_condattr_getpshared()` ,  
`pthread_mutex_destroy()`

`pthread_cond_destroy()` ,  
`pthread_create()` ,

XBD

[<pthread.h>]]

(<https://pubs.opengroup.org/onlinepubs/9799919799/basedefs/pthread.h.html>)

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

### Issue 6

The `pthread_condattr_destroy()` and `pthread_condattr_init()` functions are marked as part of the Threads option.

### Issue 7

The `pthread_condattr_destroy()` and `pthread_condattr_init()` functions are moved from the Threads option to the Base.

The **[EINVAL]** error for an uninitialized condition variable attributes object is removed; this condition results in undefined behavior.

## 1.138. pthread\_condattr\_getclock, pthread\_condattr\_setclock

### SYNOPSIS

```
#include <pthread.h>

int pthread_condattr_getclock(const pthread_condattr_t *restrict attr,
                               clockid_t *restrict clock_id);

int pthread_condattr_setclock(pthread_condattr_t *attr,
                               clockid_t clock_id);
```

### DESCRIPTION

The `pthread_condattr_getclock()` function shall obtain the value of the *clock* attribute from the attributes object referenced by *attr*.

The `pthread_condattr_setclock()` function shall set the *clock* attribute in an initialized attributes object referenced by *attr*. If `pthread_condattr_setclock()` is called with a *clock\_id* argument that refers to a CPU-time clock, the call shall fail.

The *clock* attribute is the clock ID of the clock that shall be used to measure the timeout service of `pthread_cond_timedwait()`. The default value of the *clock* attribute shall refer to the system clock. The *clock* attribute shall have no effect on the `pthread_cond_clockwait()` function.

The behavior is undefined if the value specified by the *attr* argument to `pthread_condattr_getclock()` or `pthread_condattr_setclock()` does not refer to an initialized condition variable attributes object.

### RETURN VALUE

If successful, the `pthread_condattr_getclock()` function shall return zero and store the value of the *clock* attribute of *attr* into the object referenced by the *clock\_id* argument. Otherwise, an error number shall be returned to indicate the error.

If successful, the `pthread_condattr_setclock()` function shall return zero; otherwise, an error number shall be returned to indicate the error.

# ERRORS

The `pthread_condattr_setclock()` function may fail if:

- **[EINVAL]**
- The value specified by *clock\_id* does not refer to a known clock, or is a CPU-time clock.

These functions shall not return an error code of **[EINTR]**.

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

If an implementation detects that the value specified by the *attr* argument to `pthread_condattr_getclock()` or `pthread_condattr_setclock()` does not refer to an initialized condition variable attributes object, it is recommended that the function should fail and report an **[EINVAL]** error.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_cond_clockwait()`
- `pthread_cond_destroy()`
- `pthread_condattr_destroy()`

- `pthread_condattr_getpshared()`
- `pthread_create()`
- `pthread_mutex_destroy()`

XBD `<pthread.h>`

## CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

### Issue 7

The `pthread_condattr_getclock()` and `pthread_condattr_setclock()` functions are moved from the Clock Selection option to the Base.

The **[EINVAL]** error for an uninitialized condition variable attributes object is removed; this condition results in undefined behavior.

### Issue 8

Austin Group Defect 1216 is applied, adding `pthread_cond_clockwait()`.

## 1.139. pthread\_create - thread creation

---

### SYNOPSIS

```
#include <pthread.h>

int pthread_create(pthread_t *restrict thread,
                  const pthread_attr_t *restrict attr,
                  void *(*start_routine)(void*),
                  void *restrict arg);
```

### DESCRIPTION

The `pthread_create()` function shall create a new thread, with attributes specified by `attr`, within a process. If `attr` is NULL, the default attributes shall be used. If the attributes specified by `attr` are modified later, the thread's attributes shall not be affected. Upon successful completion, `pthread_create()` shall store the ID of the created thread in the location referenced by `thread`.

The thread is created executing `start_routine` with `arg` as its sole argument. If the `start_routine` returns, the effect shall be as if there was an implicit call to `pthread_exit()` using the return value of `start_routine` as the exit status. Note that the thread in which `main()` was originally invoked differs from this. When it returns from `main()`, the effect shall be as if there was an implicit call to `exit()` using the return value of `main()` as the exit status.

### Signal State Initialization

The signal state of the new thread shall be initialized as follows:

- The signal mask shall be inherited from the creating thread.
- The set of signals pending for the new thread shall be empty.

The thread-local current locale [XSI] and the alternate stack shall not be inherited.

The floating-point environment shall be inherited from the creating thread.

If `pthread_create()` fails, no new thread is created and the contents of the location referenced by `thread` are undefined.

[TCT] If `_POSIX_THREAD_CPUTIME` is defined, the new thread shall have a CPU-time clock accessible, and the initial value of this clock shall be set to zero.

The behavior is undefined if the value specified by the `attr` argument to `pthread_create()` does not refer to an initialized thread attributes object.

## RETURN VALUE

If successful, the `pthread_create()` function shall return zero; otherwise, an error number shall be returned to indicate the error.

## ERRORS

The `pthread_create()` function shall fail if:

### [EAGAIN]

The system lacked the necessary resources to create another thread, or the system-imposed limit on the total number of threads in a process `{PTHREAD_THREADS_MAX}` would be exceeded.

### [EPERM]

The caller does not have appropriate privileges to set the required scheduling parameters or scheduling policy.

The `pthread_create()` function shall not return an error code of `[EINTR]`.

## EXAMPLES

None.

## APPLICATION USAGE

There is no requirement on the implementation that the ID of the created thread be available before the newly created thread starts executing. The calling thread can obtain the ID of the created thread through the `thread` argument of the `pthread_create()` function, and the newly created thread can obtain its ID by a call to `pthread_self()`.

## RATIONALE

A suggested alternative to `pthread_create()` would be to define two separate operations: create and start. Some applications would find such behavior more natural. Ada, in particular, separates the "creation" of a task from its "activation".

Splitting the operation was rejected by the standard developers for many reasons:

- The number of calls required to start a thread would increase from one to two and thus place an additional burden on applications that do not require the additional synchronization. The second call, however, could be avoided by the additional complication of a start-up state attribute.
- An extra state would be introduced: "created but not started". This would require the standard to specify the behavior of the thread operations when the target has not yet started executing.
- For those applications that require such behavior, it is possible to simulate the two separate steps with the facilities that are currently provided. The `start_routine()` can synchronize by waiting on a condition variable that is signaled by the start operation.

An Ada implementor can choose to create the thread at either of two points in the Ada program: when the task object is created, or when the task is activated (generally at a "begin"). If the first approach is adopted, the `start_routine()` needs to wait on a condition variable to receive the order to begin "activation". The second approach requires no such condition variable or extra synchronization. In either approach, a separate Ada task control block would need to be created when the task object is created to hold rendezvous queues, and so on.

An extension of the preceding model would be to allow the state of the thread to be modified between the create and start. This would allow the thread attributes object to be eliminated. This has been rejected because:

- All state in the thread attributes object has to be able to be set for the thread. This would require the definition of functions to modify thread attributes. There would be no reduction in the number of function calls required to set up the thread. In fact, for an application that creates all threads using identical attributes, the number of function calls required to set up the threads would be dramatically increased. Use of a thread attributes object permits the application to make one set of attribute setting function calls. Otherwise, the set of attribute setting function calls needs to be made for each thread creation.
- Depending on the implementation architecture, functions to set thread state would require kernel calls, or for other implementation reasons would not be able to be implemented as macros, thereby increasing the cost of thread creation.
- The ability for applications to segregate threads by class would be lost.

Another suggested alternative uses a model similar to that for process creation, such as "thread fork". The fork semantics would provide more flexibility and the

"create" function can be implemented simply by doing a thread fork followed immediately by a call to the desired "start routine" for the thread. This alternative has these problems:

- For many implementations, the entire stack of the calling thread would need to be duplicated, since in many architectures there is no way to determine the size of the calling frame.
- Efficiency is reduced since at least some part of the stack has to be copied, even though in most cases the thread never needs the copied context, since it merely calls the desired start routine.

If an implementation detects that the value specified by the `attr` argument to `pthread_create()` does not refer to an initialized thread attributes object, it is recommended that the function should fail and report an `[EINVAL]` error.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `fork()`
- `pthread_exit()`
- `pthread_join()`

XBD 4.15.2 Memory Synchronization, `<pthread.h>`

## CHANGE HISTORY

### First released in Issue 5

Included for alignment with the POSIX Threads Extension.

### Issue 6

The `pthread_create()` function is marked as part of the Threads option.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The `[EPERM]` mandatory error condition is added.

The thread CPU-time clock semantics are added for alignment with IEEE Std 1003.1d-1999.

The `restrict` keyword is added to the `pthread_create()` prototype for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION is updated to make it explicit that the floating-point environment is inherited from the creating thread.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/44 is applied, adding text that the alternate stack is not inherited.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/93 is applied, updating the ERRORS section to remove the mandatory `[EINVAL]` error ("The value specified by `attr` is invalid"), and adding the optional `[EINVAL]` error ("The attributes specified by `attr` are invalid").

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/94 is applied, adding the APPLICATION USAGE section.

## Issue 7

The `pthread_create()` function is moved from the Threads option to the Base.

The `[EINVAL]` error for an uninitialized thread attributes object is removed; this condition results in undefined behavior.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0458 [302] is applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0274 [849] is applied.

---

## 1.140. `pthread_detach` — detach a thread

---

### SYNOPSIS

```
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

### DESCRIPTION

The `pthread_detach()` function shall change the thread `thread` from joinable to detached, indicating to the implementation that storage for the thread can be reclaimed when the thread terminates. If `thread` has not terminated, `pthread_detach()` shall not cause it to terminate, but shall prevent the thread from becoming a zombie thread when it does terminate.

The behavior is undefined if the value specified by the `thread` argument to `pthread_detach()` does not refer to a joinable thread.

### RETURN VALUE

If the call succeeds, `pthread_detach()` shall return 0; otherwise, an error number shall be returned to indicate the error.

### ERRORS

The `pthread_detach()` function shall not return an error code of `[EINTR]`.

*The following sections are informative.*

### EXAMPLES

None.

### APPLICATION USAGE

None.

## RATIONALE

The `pthread_join()` or `pthread_detach()` functions should eventually be called for every thread that is created so that storage associated with the thread may be reclaimed.

It has been suggested that a "detach" function is not necessary; the `detachstate` thread creation attribute is sufficient, since a thread need never be dynamically detached. However, need arises in at least two cases:

1. In a cancellation handler for a `pthread_join()` it is nearly essential to have a `pthread_detach()` function in order to detach the thread on which `pthread_join()` was waiting. Without it, it would be necessary to have the handler do another `pthread_join()` to attempt to detach the thread, which would both delay the cancellation processing for an unbounded period and introduce a new call to `pthread_join()`, which might itself need a cancellation handler. A dynamic detach is nearly essential in this case.
2. In order to detach the "initial thread" (as may be desirable in processes that set up server threads).

If an implementation detects that the value specified by the `thread` argument to `pthread_detach()` does not refer to a joinable thread, it is recommended that the function should fail and report an `[EINVAL]` error.

If an implementation detects use of a thread ID after the end of its lifetime, it is recommended that the function should fail and report an `[ESRCH]` error.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_join()`
- XBD `<pthread.h>`

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

## Issue 6

The `pthread_detach()` function is marked as part of the Threads option.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/95 is applied, updating the ERRORS section so that the `[EINVAL]` and `[ESRCH]` error cases become optional.

## Issue 7

The `pthread_detach()` function is moved from the Threads option to the Base.

Austin Group Interpretation 1003.1-2001 #142 is applied, removing the `[ESRCH]` error condition.

The `[EINVAL]` error for a non-joinable thread is removed; this condition results in undefined behavior.

## Issue 8

Austin Group Defect 792 is applied, clarifying that detaching a live thread prevents it becoming a zombie thread when it terminates.

Austin Group Defect 1167 is applied, clarifying that a thread is no longer joinable after `pthread_detach()` has been called for it.

---

## 1.141. `pthread_equal` — compare thread IDs

---

### SYNOPSIS

```
#include <pthread.h>

int pthread_equal(pthread_t t1, pthread_t t2);
```

### DESCRIPTION

This function shall compare the thread IDs `t1` and `t2`.

### RETURN VALUE

The `pthread_equal()` function shall return a non-zero value if `t1` and `t2` are equal; otherwise, zero shall be returned.

If either `t1` or `t2` is not a valid thread ID and is not equal to `PTHREAD_NULL`, the behavior is undefined.

### ERRORS

No errors are defined.

The `pthread_equal()` function shall not return an error code of `[EINTR]`.

*The following sections are informative.*

### EXAMPLES

None.

### APPLICATION USAGE

None.

## RATIONALE

Implementations may choose to define a thread ID as a structure. This allows additional flexibility and robustness over using an `int`. For example, a thread ID could include a sequence number that allows detection of "dangling IDs" (copies of a thread ID that has been detached). Since the C language does not support comparison on structure types, the `pthread_equal()` function is provided to compare thread IDs.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_create()`
- `pthread_self()`
- XBD `<pthread.h>`

## CHANGE HISTORY

**First released in Issue 5.** Included for alignment with the POSIX Threads Extension.

### Issue 6

- The `pthread_equal()` function is marked as part of the Threads option.

### Issue 7

- The `pthread_equal()` function is moved from the Threads option to the Base.

### Issue 8

- Austin Group Defect 599 is applied, changing the RETURN VALUE section to mention PTHREAD\_NULL.

---

## 1.142. `pthread_exit` — thread termination

---

### SYNOPSIS

```
#include <pthread.h>

_Noreturn void pthread_exit(void *value_ptr);
```

### DESCRIPTION

The `pthread_exit()` function shall terminate the calling thread and make the value `value_ptr` available to any successful join with the terminating thread. Any cancellation cleanup handlers that have been pushed and not yet popped shall be popped in the reverse order that they were pushed and then executed. After all cancellation cleanup handlers have been executed, if the thread has any thread-specific data (whether associated with key type `tss_t` or `pthread_key_t`), appropriate destructor functions shall be called in an unspecified order. Thread termination does not release any application visible process resources, including, but not limited to, mutexes and file descriptors, nor does it perform any process-level cleanup actions, including, but not limited to, calling any `atexit()` routines that may exist.

An implicit call to `pthread_exit()` is made when a thread that was not created using `thrd_create()`, and is not the thread in which `main()` was first invoked, returns from the start routine that was used to create it. The function's return value shall serve as the thread's exit status.

The behavior of `pthread_exit()` is undefined if called from a cancellation cleanup handler or destructor function that was invoked as a result of either an implicit or explicit call to `pthread_exit()`.

After a thread has terminated, the result of access to local (auto) variables of the thread is undefined. Thus, references to local variables of the exiting thread should not be used for the `pthread_exit()` `value_ptr` parameter value.

The process shall exit with an exit status of 0 after the last thread has been terminated. The behavior shall be as if the implementation called `_exit()` with a zero argument at thread termination time.

# RETURN VALUE

The `pthread_exit()` function cannot return to its caller.

# ERRORS

No errors are defined.

# EXAMPLES

None.

# APPLICATION USAGE

Calls to `pthread_exit()` should not be made from threads created using `thrd_create()`, as their exit status has a different type (`int` instead of `void *`). If `pthread_exit()` is called from the initial thread and it is not the last thread to terminate, other threads should not try to obtain its exit status using `thrd_join()`.

# RATIONALE

The normal mechanism by which a thread that was started using `pthread_create()` terminates is to return from the routine that was specified in the `pthread_create()` call that started it. The `pthread_exit()` function provides the capability for a thread to terminate without requiring a return from the start routine of that thread, thereby providing a function analogous to `_exit()`.

Regardless of the method of thread termination, any cancellation cleanup handlers that have been pushed and not yet popped are executed, and the destructors for any existing thread-specific data are executed. This volume of POSIX.1-2024 requires that cancellation cleanup handlers be popped and called in order. After all cancellation cleanup handlers have been executed, thread-specific data destructors are called, in an unspecified order, for each item of thread-specific data that exists in the thread. This ordering is necessary because cancellation cleanup handlers may rely on thread-specific data.

As the meaning of the status is determined by the application (except when the thread has been canceled, in which case it is `PTHREAD_CANCELED`), the implementation has no idea what an illegal status value is, which is why no address error checking is done.

# FUTURE DIRECTIONS

None.

## SEE ALSO

- `_exit()`
- `pthread_create()`
- `pthread_join()`
- `pthread_key_create()`
- `thrd_create()`
- `thrd_exit()`
- `tss_create()`
- `<pthread.h>`

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

### Issue 6

The `pthread_exit()` function is marked as part of the Threads option.

### Issue 7

The `pthread_exit()` function is moved from the Threads option to the Base.

### Issue 8

Austin Group Defect 1302 is applied, adding `_Noreturn` to the SYNOPSIS, and updating the page to account for the addition of `<threads.h>` interfaces.

## 1.143. pthread\_getconcurrency

---

### Synopsis

```
#include <pthread.h>

int pthread_getconcurrency(void);
```

### Description

Unbound threads in a process may or may not be required to be simultaneously active. By default, the threads implementation ensures that a sufficient number of threads are active so that the process can continue to make progress. While this conserves system resources, it may not produce the most effective level of concurrency.

The `pthread_setconcurrency()` function allows an application to inform the threads implementation of its desired concurrency level, `new_level`. The actual level of concurrency provided by the implementation as a result of this function call is unspecified.

If `new_level` is zero, it causes the implementation to maintain the concurrency level at its discretion as if `pthread_setconcurrency()` had never been called.

The `pthread_getconcurrency()` function shall return the value set by a previous call to the `pthread_setconcurrency()` function. If the `pthread_setconcurrency()` function was not previously called, this function shall return zero to indicate that the implementation is maintaining the concurrency level.

A call to `pthread_setconcurrency()` shall inform the implementation of its desired concurrency level. The implementation shall use this as a hint, not a requirement.

### Return Value

The `pthread_getconcurrency()` function shall always return the concurrency level set by a previous call to `pthread_setconcurrency()`. If the `pthread_setconcurrency()` function has never been called, `pthread_getconcurrency()` shall return zero.

# Errors

No errors are defined for `pthread_getconcurrency()`.

## Examples

### Example 1: Get Current Concurrency Level

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

int main(void) {
    int level;

    level = pthread_getconcurrency();
    printf("Current concurrency level: %d\n", level);

    if (level == 0) {
        printf("Implementation is maintaining concurrency level\n"
    }

    return 0;
}
```

### Example 2: Set and Get Concurrency Level

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

int main(void) {
    int ret, level;

    // Set desired concurrency level
    ret = pthread_setconcurrency(4);
    if (ret != 0) {
        printf("Failed to set concurrency level\n");
        return 1;
    }

    // Get current concurrency level
    level = pthread_getconcurrency();
    printf("Concurrency level set to: %d\n", level);

    // Reset to implementation-discretion
    ret = pthread_setconcurrency(0);
```

```
    if (ret == 0) {
        level = pthread_getconcurrency();
        printf("After reset, concurrency level: %d\n", level);
    }

    return 0;
}
```

## Application Usage

Applications should use `pthread_getconcurrency()` to determine the current concurrency level that was previously set using `pthread_setconcurrency()`. The function is particularly useful for:

- Checking if a specific concurrency level has been set
- Verifying the effect of previous `pthread_setconcurrency()` calls
- Debugging and monitoring thread concurrency behavior

## Rationale

If an implementation does not support multiplexing of user threads on top of several kernel-scheduled entities, the `pthread_setconcurrency()` and `pthread_getconcurrency()` functions are provided for source code compatibility but they shall have no effect when called. To maintain the function semantics, the `new_level` parameter is saved when `pthread_setconcurrency()` is called so that a subsequent call to `pthread_getconcurrency()` shall return the same value.

## Future Directions

None.

## See Also

- `pthread_setconcurrency()`
- `pthread_create()`
- `pthread_attr_init()`

# Copyright

Portions of this text are reprinted and reproduced in electronic form from the IEEE Std 1003.1, 2013 Edition, Standard for Information Technology -- Portable Operating System Interface (POSIX), The Open Group Base Specifications Issue 7, Copyright (C) 2013 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group. In the event of any discrepancy between this version and the original IEEE and The Open Group Standard, the original IEEE and The Open Group Standard shall be the referee. The original Standard can be obtained online at <http://www.opengroup.org/unix/online.html>.

Any typographical or formatting errors that appear in this page are most likely to have been introduced during the conversion of the source files to man page format. To report such errors, see [https://www.kernel.org/doc/man-pages/reporting\\_bugs.html](https://www.kernel.org/doc/man-pages/reporting_bugs.html).

## 1.144. pthread\_getcpuclockid

---

### SYNOPSIS

```
#include <pthread.h>
#include <time.h>

int pthread_getcpuclockid(pthread_t thread_id, clockid_t *clock_id
```



### DESCRIPTION

The `pthread_getcpuclockid()` function shall return in `clock_id` the clock ID of the CPU-time clock of the thread specified by `thread_id`, if the thread specified by `thread_id` exists.

### RETURN VALUE

Upon successful completion, `pthread_getcpuclockid()` shall return zero; otherwise, an error number shall be returned to indicate the error.

### ERRORS

No errors are defined.

---

*The following sections are informative.*

### EXAMPLES

None.

### APPLICATION USAGE

The `pthread_getcpuclockid()` function is part of the Thread CPU-Time Clocks option and need not be provided on all implementations.

## RATIONALE

If an implementation detects use of a thread ID after the end of its lifetime, it is recommended that the function should fail and report an [ESRCH] error.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `clock_getcpuclockid()`
- `clock_getres()`
- `timer_create()`

XBD `<pthread.h>`, `<time.h>`

## CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

In the SYNOPSIS, the inclusion of `<sys/types.h>` is no longer required.

### Issue 7

The `pthread_getcpuclockid()` function is marked only as part of the Thread CPU-Time Clocks option as the Threads option is now part of the Base.

Austin Group Interpretation 1003.1-2001 #142 is applied, removing the [ESRCH] error condition.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0275 [757] is applied.

## 1.145. `pthread_getschedparam`, `pthread_setschedparam` — dynamic thread scheduling parameters access (REALTIME THREADS)

### SYNOPSIS

```
#include <pthread.h>

int pthread_getschedparam(pthread_t thread,
                          int *restrict policy,
                          struct sched_param *restrict param);

int pthread_setschedparam(pthread_t thread,
                          int policy,
                          const struct sched_param *param);
```

### DESCRIPTION

The `pthread_getschedparam()` and `pthread_setschedparam()` functions shall, respectively, get and set the scheduling policy and parameters of individual threads within a multi-threaded process to be retrieved and set. For SCHED\_FIFO and SCHED\_RR, the only required member of the `sched_param` structure is the priority `sched_priority`. For SCHED\_OTHER, the affected scheduling parameters are implementation-defined.

The `pthread_getschedparam()` function shall retrieve the scheduling policy and scheduling parameters for the thread whose thread ID is given by `thread` and shall store those values in `policy` and `param`, respectively. The priority value returned from `pthread_getschedparam()` shall be the value specified by the most recent `pthread_setschedparam()`, `pthread_setschedprio()`, or `pthread_create()` call affecting the target thread. It shall not reflect any temporary adjustments to its priority as a result of any priority inheritance or ceiling functions. The `pthread_setschedparam()` function shall set the scheduling policy and associated scheduling parameters for the thread whose thread ID is given by `thread` to the policy and associated parameters provided in `policy` and `param`, respectively.

The `policy` parameter may have the value SCHED\_OTHER, SCHED\_FIFO, or SCHED\_RR. The scheduling parameters for the SCHED\_OTHER policy are implementation-defined. The SCHED\_FIFO and SCHED\_RR policies shall have a single scheduling parameter, `priority`.

If `_POSIX_THREAD_SPORADIC_SERVER` is defined, then the `policy` argument may have the value `SCHED_SPORADIC`, with the exception for the `pthread_setschedparam()` function that if the scheduling policy was not `SCHED_SPORADIC` at the time of the call, it is implementation-defined whether the function is supported; in other words, the implementation need not allow the application to dynamically change the scheduling policy to `SCHED_SPORADIC`. The sporadic server scheduling policy has the associated parameters `sched_ss_low_priority`, `sched_ss_repl_period`, `sched_ss_init_budget`, `sched_priority`, and `sched_ss_max_repl`. The specified `sched_ss_repl_period` shall be greater than or equal to the specified `sched_ss_init_budget` for the function to succeed; if it is not, then the function shall fail. The value of `sched_ss_max_repl` shall be within the inclusive range  $[1, \{SS\_REPL\_MAX\}]$  for the function to succeed; if not, the function shall fail. It is unspecified whether the `sched_ss_repl_period` and `sched_ss_init_budget` values are stored as provided by this function or are rounded to align with the resolution of the clock being used.

If the `pthread_setschedparam()` function fails, the scheduling parameters shall not be changed for the target thread.

## RETURN VALUE

If successful, the `pthread_getschedparam()` and `pthread_setschedparam()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

## ERRORS

The `pthread_setschedparam()` function shall fail if:

- **ENOTSUP**: An attempt was made to set the policy or scheduling parameters to an unsupported value.
- **ENOTSUP**: An attempt was made to dynamically change the scheduling policy to `SCHED_SPORADIC`, and the implementation does not support this change.

The `pthread_setschedparam()` function may fail if:

- **EINVAL**: The value specified by `policy` or one of the scheduling parameters associated with the scheduling policy `policy` is invalid.
- **EPERM**: The caller does not have appropriate privileges to set either the scheduling parameters or the scheduling policy of the specified thread.

- **EPERM**: The implementation does not allow the application to modify one of the parameters to the value specified.

These functions shall not return an error code of [EINTR].

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

If an implementation detects use of a thread ID after the end of its lifetime, it is recommended that the function should fail and report an [ESRCH] error.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_setschedprio()`
- `sched_getparam()`
- `sched_getscheduler()`
- XBD `<pthread.h>`
- XBD `<sched.h>`

## CHANGE HISTORY

### First released in Issue 5

Included for alignment with the POSIX Threads Extension.

## Issue 6

- The `pthread_getschedparam()` and `pthread_setschedparam()` functions are marked as part of the Threads and Thread Execution Scheduling options.
- The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Thread Execution Scheduling option.
- The Open Group Corrigendum U026/2 is applied, correcting the prototype for the `pthread_setschedparam()` function so that its second argument is of type `int`.
- The SCHED\_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.
- The `restrict` keyword is added to the `pthread_getschedparam()` prototype for alignment with the ISO/IEC 9899:1999 standard.
- The Open Group Corrigendum U047/1 is applied.
- IEEE PASC Interpretation 1003.1 #96 is applied, noting that priority values can also be set by a call to the `pthread_setschedprio()` function.

## Issue 7

- The `pthread_getschedparam()` and `pthread_setschedparam()` functions are marked only as part of the Thread Execution Scheduling option as the Threads option is now part of the Base.
- Austin Group Interpretation 1003.1-2001 #119 is applied, clarifying the accuracy requirements for the `sched_ss_repl_period` and `sched_ss_init_budget` values.
- Austin Group Interpretation 1003.1-2001 #142 is applied, removing the [ESRCH] error condition.
- POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0459 [314] is applied.
- POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0276 [757] is applied.

---

## 1.146. `pthread_getspecific`, `pthread_setspecific` — thread-specific data management

---

### SYNOPSIS

```
#include <pthread.h>

void *pthread_getspecific(pthread_key_t key);
int pthread_setspecific(pthread_key_t key, const void *value);
```

### DESCRIPTION

The `pthread_getspecific()` function shall return the value currently bound to the specified `key` on behalf of the calling thread.

The `pthread_setspecific()` function shall associate a thread-specific `value` with a `key` obtained via a previous call to `pthread_key_create()`. Different threads may bind different values to the same key. These values are typically pointers to blocks of dynamically allocated memory that have been reserved for use by the calling thread.

The effect of calling `pthread_getspecific()` or `pthread_setspecific()` with a `key` value not obtained from `pthread_key_create()` or after `key` has been deleted with `pthread_key_delete()` is undefined.

Both `pthread_getspecific()` and `pthread_setspecific()` may be called from a thread-specific data destructor function. A call to `pthread_getspecific()` for the thread-specific data key being destroyed shall return the value `NULL`, unless the value is changed (after the destructor starts) by a call to `pthread_setspecific()`. Calling `pthread_setspecific()` from a thread-specific data destructor routine may result either in lost storage (after at least `PTHREAD_DESTRUCTOR_ITERATIONS` attempts at destruction) or in an infinite loop.

Both functions may be implemented as macros.

### RETURN VALUE

The `pthread_getspecific()` function shall return the thread-specific data value associated with the given `key`. If no thread-specific data value is associated with `key`, then the value `NULL` shall be returned.

If successful, the `pthread_setspecific()` function shall return zero; otherwise, an error number shall be returned to indicate the error.

## ERRORS

No errors are returned from `pthread_getspecific()`.

The `pthread_setspecific()` function shall fail if:

- **[ENOMEM]**
- Insufficient memory exists to associate the non-NULL value with the key.

The `pthread_setspecific()` function shall not return an error code of [EINTR].

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

Performance and ease-of-use of `pthread_getspecific()` are critical for functions that rely on maintaining state in thread-specific data. Since no errors are required to be detected by it, and since the only error that could be detected is the use of an invalid key, the function to `pthread_getspecific()` has been designed to favor speed and simplicity over error reporting.

If an implementation detects that the value specified by the `key` argument to `pthread_setspecific()` does not refer to a key value obtained from `pthread_key_create()` or refers to a key that has been deleted with `pthread_key_delete()`, it is recommended that the function should fail and report an [EINVAL] error.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_key_create()`
- `<pthread.h>`

## CHANGE HISTORY

### First released in Issue 5

Included for alignment with the POSIX Threads Extension.

### Issue 6

- The `pthread_getspecific()` and `pthread_setspecific()` functions are marked as part of the Threads option.
- IEEE PASC Interpretation 1003.1c #3 (Part 6) is applied, updating the DESCRIPTION.
- IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/96 is applied, updating the ERRORS section so that the [ENOMEM] error case is changed from "to associate the value with the key" to "to associate the non-NUL value with the key".

### Issue 7

- Austin Group Interpretation 1003.1-2001 #063 is applied, updating the ERRORS section.
  - The `pthread_getspecific()` and `pthread_setspecific()` functions are moved from the Threads option to the Base.
  - The [EINVAL] error for a key value not obtained from `pthread_key_create()` or a key deleted with `pthread_key_delete()` is removed; this condition results in undefined behavior.
-

## 1.147. `pthread_join` — wait for thread termination

---

### SYNOPSIS

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **value_ptr);
```

### DESCRIPTION

The `pthread_join()` function shall suspend execution of the calling thread until the target thread terminates, unless the target thread has already terminated. On return from a successful `pthread_join()` call with a non-NULL `value_ptr` argument, the value passed to `pthread_exit()` by the terminating thread shall be made available in the location referenced by `value_ptr`. When a `pthread_join()` returns successfully, the target thread has been terminated. The results of multiple simultaneous calls to `pthread_join()` specifying the same target thread are undefined. If the thread calling `pthread_join()` is canceled, then the target thread shall not be detached.

It is unspecified whether a zombie thread counts against `{PTHREAD_THREADS_MAX}`.

The behavior is undefined if the value specified by the thread argument to `pthread_join()` does not refer to a joinable thread.

The behavior is undefined if the value specified by the thread argument to `pthread_join()` refers to the calling thread.

If `thread` refers to a thread that was created using `thrd_create()` and the thread terminates, or has already terminated, by returning from its start routine, the behavior of `pthread_join()` is undefined. If `thread` refers to a thread that terminates, or has already terminated, by calling `thrd_exit()`, the behavior of `pthread_join()` is undefined.

### RETURN VALUE

If successful, the `pthread_join()` function shall return zero; otherwise, an error number shall be returned to indicate the error.

# ERRORS

The `pthread_join()` function may fail if:

- **[EDEADLK]** A deadlock was detected.

The `pthread_join()` function shall not return an error code of [EINTR].

# EXAMPLES

An example of thread creation and deletion follows:

```
typedef struct {
    int *ar;
    long n;
} subarray;

void *
incer(void *arg)
{
    long i;

    for (i = 0; i < ((subarray *)arg)->n; i++)
        ((subarray *)arg)->ar[i]++;
}

int main(void)
{
    int         ar[1000000];
    pthread_t   th1, th2;
    subarray   sb1, sb2;

    sb1.ar = &ar[0];
    sb1.n  = 500000;
    (void) pthread_create(&th1, NULL, incer, &sb1);

    sb2.ar = &ar[500000];
    sb2.n  = 500000;
    (void) pthread_create(&th2, NULL, incer, &sb2);

    (void) pthread_join(th1, NULL);
    (void) pthread_join(th2, NULL);
    return 0;
}
```

# APPLICATION USAGE

None.

## RATIONALE

The `pthread_join()` function is a convenience that has proven useful in multi-threaded applications. It is true that a programmer could simulate this function if it were not provided by passing extra state as part of the argument to the `start_routine()`. The terminating thread would set a flag to indicate termination and broadcast a condition that is part of that state; a joining thread would wait on that condition variable. While such a technique would allow a thread to wait on more complex conditions (for example, waiting for multiple threads to terminate), waiting on individual thread termination is considered widely useful. Also, including the `pthread_join()` function in no way precludes a programmer from coding such complex waits. Thus, while not a primitive, including `pthread_join()` in this volume of POSIX.1-2024 was considered valuable.

The `pthread_join()` function provides a simple mechanism allowing an application to wait for a thread to terminate. After the thread terminates, the application may then choose to clean up resources that were used by the thread. For instance, after `pthread_join()` returns, any application-provided stack storage could be reclaimed.

The `pthread_join()` or `pthread_detach()` function should eventually be called for every thread that is created with the detachstate attribute set to PTHREAD\_CREATE\_JOINABLE so that storage associated with the thread may be reclaimed.

The interaction between `pthread_join()` and cancellation is well-defined for the following reasons:

- The `pthread_join()` function, like all other non-async-cancel-safe functions, can only be called with deferred cancelability type.
- Cancellation cannot occur in the disabled cancelability state.

Thus, only the default cancelability state need be considered. As specified, either the `pthread_join()` call is canceled, or it succeeds, but not both. The difference is obvious to the application, since either a cancellation handler is run or `pthread_join()` returns. There are no race conditions since `pthread_join()` was called in the deferred cancelability state.

If an implementation detects that the value specified by the thread argument to `pthread_join()` does not refer to a joinable thread, it is recommended that the function should fail and report an [EINVAL] error.

If an implementation detects that the value specified by the thread argument to `pthread_join()` refers to the calling thread, it is recommended that the function should fail and report an [EDEADLK] error.

If an implementation detects use of a thread ID after the end of its lifetime, it is recommended that the function should fail and report an [ESRCH] error.

The `pthread_join()` function cannot be used to obtain the exit status of a thread that was created using `thrd_create()` and which terminates by returning from its start routine, or of a thread that terminates by calling `thrd_exit()`, because such threads have an `int` exit status, instead of the `void *` that `pthread_join()` returns via its `value_ptr` argument.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_create()`
- `thrd_create()`
- `thrd_exit()`
- `wait()`
- XBD 4.15.2 Memory Synchronization, `<pthread.h>`

## CHANGE HISTORY

**First released in Issue 5.** Included for alignment with the POSIX Threads Extension.

### Issue 6

The `pthread_join()` function is marked as part of the Threads option.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/97 is applied, updating the ERRORS section so that the [EINVAL] error is made optional and the words "the implementation has detected" are removed from it.

### Issue 7

The `pthread_join()` function is moved from the Threads option to the Base.

Austin Group Interpretation 1003.1-2001 #142 is applied, removing the [ESRCH] error condition.

The [EINVAL] error for a non-joinable thread is removed; this condition results in undefined behavior.

The [EDEADLK] error for the calling thread is removed; this condition results in undefined behavior.

### **Issue 8**

Austin Group Defect 792 is applied, changing "a thread that has exited but remains unjoined" to "a zombie thread".

Austin Group Defect 1302 is applied, updating the page to account for the addition of `<threads.h>` interfaces.

---

## 1.148. `pthread_key_create` — thread-specific data key creation

---

### SYNOPSIS

```
#include <pthread.h>

int pthread_key_create(pthread_key_t *key, void (*destructor)(void*
```



### DESCRIPTION

The `pthread_key_create()` function shall create a thread-specific data key visible to all threads in the process. Key values provided by `pthread_key_create()` are opaque objects used to locate thread-specific data. Although the same key value may be used by different threads, the values bound to the key by `pthread_setspecific()` are maintained on a per-thread basis and persist for the life of the calling thread.

Upon key creation, the value `NULL` shall be associated with the new key in all active threads. Upon thread creation, the value `NULL` shall be associated with all defined keys in the new thread.

An optional destructor function may be associated with each key value. At thread exit, if a key value has a non-`NULL` destructor pointer, and the thread has a non-`NULL` value associated with that key, the value of the key is set to `NULL`, and then the function pointed to is called with the previously associated value as its sole argument. The order of destructor calls is unspecified if more than one destructor exists for a thread when it exits.

If, after all the destructors have been called for all non-`NULL` values with associated destructors, there are still some non-`NULL` values with associated destructors, then the process is repeated. If, after at least `{PTHREAD_DESTRUCTOR_ITERATIONS}` iterations of destructor calls for outstanding non-`NULL` values, there are still some non-`NULL` values with associated destructors, implementations may stop calling destructors, or they may continue calling destructors until no non-`NULL` values with associated destructors exist, even though this might result in an infinite loop.

## RETURN VALUE

If successful, the `pthread_key_create()` function shall store the newly created key value at `*key` and shall return zero. Otherwise, an error number shall be returned to indicate the error.

## ERRORS

The `pthread_key_create()` function shall fail if:

- **[EAGAIN]**
  - The system lacked the necessary resources to create another thread-specific data key, or the system-imposed limit on the total number of keys per process {PTHREAD\_KEYS\_MAX} has been exceeded.
- **[ENOMEM]**
  - Insufficient memory exists to create the key.

The `pthread_key_create()` function shall not return an error code of [EINTR].

## EXAMPLES

The following example demonstrates a function that initializes a thread-specific data key when it is first called, and associates a thread-specific object with each calling thread, initializing this object when necessary.

```
static pthread_key_t key;
static pthread_once_t key_once = PTHREAD_ONCE_INIT;

static void
make_key()
{
    (void) pthread_key_create(&key, NULL);
}

func()
{
    void *ptr;

    (void) pthread_once(&key_once, make_key);
    if ((ptr = pthread_getspecific(key)) == NULL) {
        ptr = malloc(OBJECT_SIZE);
        ...
        (void) pthread_setspecific(key, ptr);
    }
}
```

```
...  
}
```

Note that the key has to be initialized before `pthread_getspecific()` or `pthread_setspecific()` can be used. The `pthread_key_create()` call could either be explicitly made in a module initialization routine, or it can be done implicitly by the first call to a module as in this example. Any attempt to use the key before it is initialized is a programming error, making the code below incorrect.

```
static pthread_key_t key;  
  
func()  
{  
    void *ptr;  
  
    /* KEY NOT INITIALIZED!!! THIS WILL NOT WORK!!! */  
    if ((ptr = pthread_getspecific(key)) == NULL &&  
        pthread_setspecific(key, NULL) != 0) {  
        pthread_key_create(&key, NULL);  
        ...  
    }  
}
```

## APPLICATION USAGE

None.

## RATIONALE

### Destructor Functions

Normally, the value bound to a key on behalf of a particular thread is a pointer to storage allocated dynamically on behalf of the calling thread. The destructor functions specified with `pthread_key_create()` are intended to be used to free this storage when the thread exits. Thread cancellation cleanup handlers cannot be used for this purpose because thread-specific data may persist outside the lexical scope in which the cancellation cleanup handlers operate.

If the value associated with a key needs to be updated during the lifetime of the thread, it may be necessary to release the storage associated with the old value before the new value is bound. Although the `pthread_setspecific()` function could do this automatically, this feature is not needed often enough to justify the added complexity. Instead, the programmer is responsible for freeing the stale storage:

```
old = pthread_getspecific(key);
new = allocate();
destructor(old);
pthread_setspecific(key, new);
```

#### Note:

The above example could leak storage if run with asynchronous cancellation enabled. No such problems occur in the default cancellation state if no cancellation points occur between the get and set.

There is no notion of a destructor-safe function. If an application does not call `pthread_exit()` from a signal handler, or if it blocks any signal whose handler may call `pthread_exit()` while calling async-unsafe functions, all functions may be safely called from destructors.

## Non-Idempotent Data Key Creation

There were requests to make `pthread_key_create()` idempotent with respect to a given key address parameter. This would allow applications to call `pthread_key_create()` multiple times for a given key address and be guaranteed that only one key would be created. Doing so would require the key value to be previously initialized (possibly at compile time) to a known null value and would require that implicit mutual-exclusion be performed based on the address and contents of the key parameter in order to guarantee that exactly one key would be created.

Unfortunately, the implicit mutual-exclusion would not be limited to only `pthread_key_create()`. On many implementations, implicit mutual-exclusion would also have to be performed by `pthread_getspecific()` and `pthread_setspecific()` in order to guard against using incompletely stored or not-yet-visible key values. This could significantly increase the cost of important operations, particularly `pthread_getspecific()`.

Thus, this proposal was rejected. The `pthread_key_create()` function performs no implicit synchronization. It is the responsibility of the programmer to ensure that it is called exactly once per key before use of the key. Several straightforward mechanisms can already be used to accomplish this, including calling explicit module initialization functions, using mutexes, and using `pthread_once()`. This places no significant burden on the programmer, introduces no possibly confusing *ad hoc* implicit synchronization mechanism, and potentially allows commonly used thread-specific data operations to be more efficient.

# FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_getspecific()`
- `pthread_key_delete()`
- `<pthread.h>`

## CHANGE HISTORY

### First released in Issue 5

Included for alignment with the POSIX Threads Extension.

### Issue 6

The `pthread_key_create()` function is marked as part of the Threads option.

IEEE PASC Interpretation 1003.1c #8 is applied, updating the DESCRIPTION.

### Issue 7

The `pthread_key_create()` function is moved from the Threads option to the Base.

### Issue 8

Austin Group Defect 1059 is applied, changing the example code in the RATIONALE section.

---

## 1.149. pthread\_key\_delete

---

### Synopsis

```
#include <pthread.h>

int pthread_key_delete(pthread_key_t key);
```

### Description

The `pthread_key_delete()` function shall delete a thread-specific data key previously returned by `pthread_key_create()`. The thread-specific data values associated with `key` need not be NULL at the time `pthread_key_delete()` is called. It is the responsibility of the application to free any application storage or perform any cleanup actions for data structures related to the deleted key or associated thread-specific data in any threads; this cleanup can be done either before or after `pthread_key_delete()` is called. Any attempt to use `key` following the call to `pthread_key_delete()` results in undefined behavior.

The `pthread_key_delete()` function shall be callable from within destructor functions. No destructor functions shall be invoked by `pthread_key_delete()`. Any destructor function that may have been associated with `key` shall no longer be called upon thread exit.

### Return Value

Upon successful completion, `pthread_key_delete()` shall return zero; otherwise, an error number shall be returned to indicate the error.

### Errors

The `pthread_key_delete()` function may fail if:

- **EINVAL** - The value specified by `key` does not refer to a key value obtained from `pthread_key_create()` or refers to a key that has been deleted with `pthread_key_delete()`.

## Examples

*No examples are provided.*

## Application Usage

A thread-specific data key deletion function has been included in order to allow the resources associated with an unused thread-specific data key to be freed. Unused thread-specific data keys can arise, among other scenarios, when a dynamically loaded module that allocated a key is unloaded.

Conforming applications are responsible for performing any cleanup actions needed for data structures associated with the key to be deleted, including data referenced by thread-specific data values. No such cleanup is done by `pthread_key_delete()`. In particular, destructor functions are not called. There are several reasons for this division of responsibility:

1. The associated destructor functions used to free thread-specific data at thread exit time are only guaranteed to work correctly when called in the thread that allocated the thread-specific data. (Destructors themselves may utilize thread-specific data.) Thus, they cannot be used to free thread-specific data in other threads at key deletion time. Attempting to have them called by other threads at key deletion time would require other threads to be asynchronously interrupted. But since interrupted threads could be in an arbitrary state, including holding locks necessary for the destructor to run, this approach would fail. In general, there is no safe mechanism whereby an implementation could free thread-specific data at key deletion time.
2. Even if there were a means of safely freeing thread-specific data associated with keys to be deleted, doing so would require that implementations be able to enumerate the threads with non-NULL data and potentially keep them from creating more thread-specific data while the key deletion is occurring. This special case could cause extra synchronization in the normal case, which would otherwise be unnecessary.

For an application to know that it is safe to delete a key, it has to know that all the threads that might potentially ever use the key do not attempt to use it again. For example, it could know this if all the client threads have called a cleanup procedure declaring that they are through with the module that is being shut down, perhaps by setting a reference count to zero.

## Rationale

If an implementation detects that the value specified by the `key` argument to `pthread_key_delete()` does not refer to a key value obtained from `pthread_key_create()` or refers to a key that has been deleted with `pthread_key_delete()`, it is recommended that the function should fail and report an `[EINVAL]` error.

## Future Directions

*None.*

## See Also

`pthread_key_create()`

## XSI

*None.*

## Copyright

Portions of this text are reprinted and reproduced in electronic form from the IEEE Std 1003.1-2017 for POSIX.1-2017, Standard for Information Technology -- Portable Operating System Interface (POSIX), The Open Group Base Specifications Issue 7, 2018 Edition, Copyright (C) 2018 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group. In the event of any discrepancy between this version and the original IEEE and The Open Group Standard, the original IEEE and The Open Group Standard shall govern. The original Standard can be obtained online at <http://www.opengroup.org/unix/online.html>.

Any typographical or formatting errors that appear in this page are most likely to have been introduced during the conversion of the source files to man page format. To report such errors, see [https://www.kernel.org/doc/man-pages/reporting\\_bugs.html](https://www.kernel.org/doc/man-pages/reporting_bugs.html).

## 1.150. pthread\_kill

---

### SYNOPSIS

```
#include <signal.h>

int pthread_kill(pthread_t thread, int sig);
```

### DESCRIPTION

The `pthread_kill()` function shall request that a signal be delivered to the specified thread. It shall not be an error if `thread` is a zombie thread.

### RETURN VALUE

Upon successful completion, the function shall return a value of zero. Otherwise, the function shall return an error number. If the `pthread_kill()` function fails, no signal shall be sent.

### ERRORS

The `pthread_kill()` function may fail if:

- **[EINVAL]**
- The value of the `sig` argument is zero.

The `pthread_kill()` function shall fail if:

- **[EINVAL]**
- The value of the `sig` argument is non-zero and is an invalid or unsupported signal number.

The `pthread_kill()` function shall not return an error code of **[EINTR]**.

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

The `pthread_kill()` function provides a mechanism for asynchronously directing a signal at a thread in the calling process. This could be used, for example, by one thread to affect broadcast delivery of a signal to a set of threads.

Note that `pthread_kill()` only causes the signal to be handled in the context of the given thread; the signal action (termination or stopping) affects the process as a whole.

## RATIONALE

If an implementation detects use of a thread ID after the end of its lifetime, it is recommended that the function should fail and report an **[ESRCH]** error.

Historical implementations varied on the result of a `pthread_kill()` with a thread ID indicating a zombie thread. Some indicated success on such a call, while others gave an error of **[ESRCH]**. Since the definition of thread lifetime in this volume of POSIX.1-2024 covers zombie threads, the **[ESRCH]** error as described is inappropriate in this case and implementations that give this error do not conform. In particular, this means that an application cannot have one thread check for termination of another by calling `pthread_kill()` with a `sig` argument of zero, and implementations may indicate that it is not possible by returning **[EINVAL]** when `sig` is zero.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `kill()`
- `pthread_self()`
- `raise()`

XBD `<signal.h>`

# CHANGE HISTORY

**First released in Issue 5.** Included for alignment with the POSIX Threads Extension.

## Issue 6

The `pthread_kill()` function is marked as part of the Threads option.

The APPLICATION USAGE section is added.

## Issue 7

The `pthread_kill()` function is moved from the Threads option to the Base.

Austin Group Interpretation 1003.1-2001 #142 is applied, removing the **[ESRCH]** error condition.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0277 [765] is applied.

## Issue 8

Austin Group Defect 792 is applied, adding a requirement that passing the thread ID of a zombie thread to `pthread_kill()` is not treated as an error.

Austin Group Defect 1214 is applied, allowing `pthread_kill()` to fail with **[EINVAL]** when the `sig` argument is zero.

## 1.151. pthread\_mutex\_destroy

---

### Synopsis

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                      const pthread_mutexattr_t *restrict attr);

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

### Description

The `pthread_mutex_destroy()` function shall destroy the mutex object referenced by `mutex`; the mutex object becomes, in effect, uninitialized. An implementation may cause `pthread_mutex_destroy()` to set the object referenced by `mutex` to an invalid value.

A destroyed mutex object can be reinitialized using `pthread_mutex_init()`; the results of otherwise referencing the object after it has been destroyed are undefined.

It shall be safe to destroy an initialized mutex that is unlocked. Attempting to destroy a locked mutex, or a mutex that another thread is attempting to lock, or a mutex that is being used in a `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, or `pthread_cond_wait()` call by another thread, results in undefined behavior.

The `pthread_mutex_init()` function shall initialize the mutex referenced by `mutex` with attributes specified by `attr`. If `attr` is NULL, the default mutex attributes are used; the effect shall be the same as passing the address of a default mutex attributes object. Upon successful initialization, the state of the mutex becomes initialized and unlocked.

See [2.9.9 Synchronization Object Copies and Alternative Mappings](#) for further requirements.

Attempting to initialize an already initialized mutex results in undefined behavior.

In cases where default mutex attributes are appropriate, the macro `PTHREAD_MUTEX_INITIALIZER` can be used to initialize mutexes. The effect shall be equivalent to dynamic initialization by a call to `pthread_mutex_init()` with parameter `attr` specified as NULL, except that no error checks are performed.

The behavior is undefined if the value specified by the `mutex` argument to `pthread_mutex_destroy()` does not refer to an initialized mutex.

The behavior is undefined if the value specified by the `attr` argument to `pthread_mutex_init()` does not refer to an initialized mutex attributes object.

## Return Value

If successful, the `pthread_mutex_destroy()` and `pthread_mutex_init()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

## Errors

The `pthread_mutex_destroy()` function may fail if:

- **EBUSY** - The implementation has detected an attempt to destroy the object referenced by `mutex` while it is locked or referenced (for example, while being used in a `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, or `pthread_cond_wait()` call) by another thread.
- **EINVAL** - The value specified by `mutex` does not refer to an initialized mutex object.

The `pthread_mutex_init()` function may fail if:

- **EBUSY** - The implementation has detected an attempt to reinitialize the object referenced by `mutex`, a previously initialized, but not yet destroyed, mutex.
- **EINVAL** - The value specified by `attr` does not refer to an initialized mutex attributes object, or the value specified by `mutex` does not refer to an initialized mutex.
- **ENOMEM** - Insufficient memory exists to initialize the mutex.

These functions shall not return an error code of [EINTR].

## Examples

### Static Initialization

```
static pthread_mutex_t foo_mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
void foo()
{
    pthread_mutex_lock(&foo_mutex);
    /* Do work. */
    pthread_mutex_unlock(&foo_mutex);
}
```

## Dynamic Initialization

```
static pthread_once_t foo_once = PTHREAD_ONCE_INIT;
static pthread_mutex_t foo_mutex;

void foo_init()
{
    pthread_mutex_init(&foo_mutex, NULL);
}

void foo()
{
    pthread_once(&foo_once, foo_init);
    pthread_mutex_lock(&foo_mutex);
    /* Do work. */
    pthread_mutex_unlock(&foo_mutex);
}
```

## Application Usage

The behavior is undefined if the value specified by the `mutex` argument to `pthread_mutex_destroy()` does not refer to an initialized mutex.

The behavior is undefined if the value specified by the `attr` argument to `pthread_mutex_init()` does not refer to an initialized mutex attributes object.

## Rationale

### Alternate Implementations Possible

This volume of POSIX.1-2024 supports several alternative implementations of mutexes. An implementation may store the lock directly in the object of type `pthread_mutex_t`. Alternatively, an implementation may store the lock in the heap and merely store a pointer, handle, or unique ID in the mutex object. Either implementation has advantages or may be required on certain hardware configurations. So that portable code can be written that is invariant to this choice,

this volume of POSIX.1-2024 does not define assignment or equality for this type, and it uses the term "initialize" to reinforce the (more restrictive) notion that the lock may actually reside in the mutex object itself.

Note that this precludes an over-specification of the type of the mutex or condition variable and motivates the opaqueness of the type.

An implementation is permitted, but not required, to have `pthread_mutex_destroy()` store an illegal value into the mutex. This may help detect erroneous programs that try to lock (or otherwise reference) a mutex that has already been destroyed.

## Tradeoff Between Error Checks and Performance Supported

Many error conditions that can occur are not required to be detected by the implementation in order to let implementations trade off performance versus degree of error checking according to the needs of their specific applications and execution environment. As a general rule, conditions caused by the system (such as insufficient memory) are required to be detected, but conditions caused by an erroneously coded application (such as failing to provide adequate synchronization to prevent a mutex from being deleted while in use) are specified to result in undefined behavior.

## Static Initializers for Mutexes and Condition Variables

Providing for static initialization of statically allocated synchronization objects allows modules with private static synchronization variables to avoid runtime initialization tests and overhead. Furthermore, it simplifies the coding of self-initializing modules. Such modules are common in C libraries, where for various reasons the design calls for self-initialization instead of requiring an explicit module initialization function to be called.

## Destroying Mutexes

A mutex can be destroyed immediately after it is unlocked. However, since attempting to destroy a locked mutex, or a mutex that another thread is attempting to lock, or a mutex that is being used in a `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, or `pthread_cond_wait()` call by another thread, results in undefined behavior, care must be taken to ensure that no other thread may be referencing the mutex.

## Robust Mutexes

Implementations are required to provide robust mutexes for mutexes with the process-shared attribute set to `PTHREAD_PROCESS_SHARED`. Implementations are allowed, but not required, to provide robust mutexes when the process-shared attribute is set to `PTHREAD_PROCESS_PRIVATE`.

## See Also

- `pthread_cond_clockwait()`
- `pthread_cond_timedwait()`
- `pthread_cond_wait()`
- `pthread_mutex_lock()`
- `pthread_mutex_unlock()`
- `pthread_once()`
- [Base Definitions, `<pthread.h>`](#)

## Change History

Derived from the POSIX Threads Extension (1003.1c-1995).

## 1.152. pthread\_mutex\_getprioceiling, pthread\_mutex\_setprioceiling

### SYNOPSIS

```
[RPP|TPP]
#include <pthread.h>

int pthread_mutex_getprioceiling(const pthread_mutex_t *restrict mutex,
                                  int *restrict prioceiling);

int pthread_mutex_setprioceiling(pthread_mutex_t *restrict mutex,
                                 int prioceiling, int *restrict old_ceiling);
```

### DESCRIPTION

The `pthread_mutex_getprioceiling()` function shall return the current priority ceiling of the mutex.

The `pthread_mutex_setprioceiling()` function shall attempt to lock the mutex as if by a call to `pthread_mutex_lock()`, except that the process of locking the mutex need not adhere to the priority protect protocol. On acquiring the mutex it shall change the mutex's priority ceiling and then release the mutex as if by a call to `pthread_mutex_unlock()`. When the change is successful, the previous value of the priority ceiling shall be returned in `old_ceiling`.

If the `pthread_mutex_setprioceiling()` function fails, the mutex priority ceiling shall not be changed.

### RETURN VALUE

If successful, the `pthread_mutex_getprioceiling()` and `pthread_mutex_setprioceiling()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

### ERRORS

These functions shall fail if:

- **[EINVAL]**

The protocol attribute of `mutex` is PTHREAD\_PRIO\_NONE.

- **[EPERM]**

The implementation requires appropriate privileges to perform the operation and the caller does not have appropriate privileges.

The `pthread_mutex_setprioceiling()` function shall fail if:

- **[EAGAIN]**

The mutex could not be acquired because the maximum number of recursive locks for `mutex` has been exceeded.

- **[EAGAIN]**

The mutex is a robust mutex and the system resources available for robust mutexes owned would be exceeded.

- **[EDEADLK]**

The mutex type is PTHREAD\_MUTEX\_ERRORCHECK and the current thread already owns the mutex.

- **[EINVAL]**

The mutex was created with the protocol attribute having the value PTHREAD\_PRIO\_PROTECT and the calling thread's priority is higher than the mutex's current priority ceiling, and the implementation adheres to the priority protect protocol in the process of locking the mutex.

- **[ENOTRECOVERABLE]**

The mutex is a robust mutex and the state protected by the mutex is not recoverable.

- **[EOWNERDEAD]**

The mutex is a robust mutex and the process containing the previous owning thread terminated while holding the mutex lock. The mutex lock shall be acquired by the calling thread and it is up to the new owner to make the state consistent (see `pthread_mutex_lock()` ).

The `pthread_mutex_setprioceiling()` function may fail if:

- **[EDEADLK]**

A deadlock condition was detected.

- **[EINVAL]**

The priority requested by `prioceiling` is out of range.

- **[EOWNERDEAD]**

The mutex is a robust mutex and the previous owning thread terminated while holding the mutex lock. The mutex lock shall be acquired by the calling thread

and it is up to the new owner to make the state consistent (see [pthread\\_mutex\\_lock\(\)](#) ).

These functions shall not return an error code of [EINTR].

---

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [pthread\\_mutex\\_clocklock\(\)](#)
- [pthread\\_mutex\\_destroy\(\)](#)
- [pthread\\_mutex\\_lock\(\)](#)
- XBD [`<pthread.h>`](#)

## CHANGE HISTORY

### First released in Issue 5

Included for alignment with the POSIX Threads Extension.  
Marked as part of the Realtime Threads Feature Group.

## Issue 6

The `pthread_mutex_getprioceiling()` and `pthread_mutex_setprioceiling()` functions are marked as part of the Threads and Thread Priority Protection options.

The [ENOSYS] error conditions have been removed.

The `pthread_mutex_timedlock()` function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

The **restrict** keyword is added to the `pthread_mutex_getprioceiling()` and `pthread_mutex_setprioceiling()` prototypes for alignment with the ISO/IEC 9899:1999 standard.

## Issue 7

SD5-XSH-ERN-39 is applied.

Austin Group Interpretation 1003.1-2001 #052 is applied, adding [EDEADLK] as a "may fail" error.

SD5-XSH-ERN-158 is applied, updating the ERRORS section to include a "shall fail" error case for when the protocol attribute of `mutex` is `PTHREAD_PRIO_NONE`.

The `pthread_mutex_getprioceiling()` and `pthread_mutex_setprioceiling()` functions are moved from the Threads option to require support of either the Robust Mutex Priority Protection option or the Non-Robust Mutex Priority Protection option.

The DESCRIPTION and ERRORS sections are updated to account properly for all of the various mutex types.

## Issue 8

Austin Group Defect 354 is applied, adding the [EAGAIN] error for exceeding system resources available for robust mutexes owned.

## 1.153. pthread\_mutex\_destroy, pthread\_mutex\_init

---

### SYNOPSIS

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                      const pthread_mutexattr_t *restrict attr);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

### DESCRIPTION

The `pthread_mutex_destroy()` function shall destroy the mutex object referenced by `mutex`; the mutex object becomes, in effect, uninitialized. An implementation may cause `pthread_mutex_destroy()` to set the object referenced by `mutex` to an invalid value.

A destroyed mutex object can be reinitialized using `pthread_mutex_init()`; the results of otherwise referencing the object after it has been destroyed are undefined.

It shall be safe to destroy an initialized mutex that is unlocked. Attempting to destroy a locked mutex, or a mutex that another thread is attempting to lock, or a mutex that is being used in a `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, or `pthread_cond_wait()` call by another thread, results in undefined behavior.

The `pthread_mutex_init()` function shall initialize the mutex referenced by `mutex` with attributes specified by `attr`. If `attr` is NULL, the default mutex attributes are used; the effect shall be the same as passing the address of a default mutex attributes object. Upon successful initialization, the state of the mutex becomes initialized and unlocked.

See [2.9.9 Synchronization Object Copies and Alternative Mappings](#) for further requirements.

Attempting to initialize an already initialized mutex results in undefined behavior.

In cases where default mutex attributes are appropriate, the macro `PTHREAD_MUTEX_INITIALIZER` can be used to initialize mutexes. The effect shall be equivalent to dynamic initialization by a call to `pthread_mutex_init()` with parameter `attr` specified as NULL, except that no error checks are performed.

The behavior is undefined if the value specified by the `mutex` argument to `pthread_mutex_destroy()` does not refer to an initialized mutex.

The behavior is undefined if the value specified by the `attr` argument to `pthread_mutex_init()` does not refer to an initialized mutex attributes object.

## RETURN VALUE

If successful, the `pthread_mutex_destroy()` and `pthread_mutex_init()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

## ERRORS

The `pthread_mutex_init()` function shall fail if:

- **[EAGAIN]**: The system lacked the necessary resources (other than memory) to initialize another mutex.
- **[ENOMEM]**: Insufficient memory exists to initialize the mutex.
- **[EPERM]**: The caller does not have the privilege to perform the operation.

The `pthread_mutex_init()` function may fail if:

- **[EINVAL]**: The attributes object referenced by `attr` has the robust mutex attribute set without the process-shared attribute being set.

These functions shall not return an error code of [EINTR].

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

If an implementation detects that the value specified by the `mutex` argument to `pthread_mutex_destroy()` does not refer to an initialized mutex, it is recommended that the function should fail and report an [EINVAL] error.

If an implementation detects that the value specified by the `mutex` argument to `pthread_mutex_destroy()` or `pthread_mutex_init()` refers to a locked mutex or a mutex that is referenced (for example, while being used in a `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, or `pthread_cond_wait()` call) by another thread, or detects that the value specified by the `mutex` argument to `pthread_mutex_init()` refers to an already initialized mutex, it is recommended that the function should fail and report an [EBUSY] error.

If an implementation detects that the value specified by the `attr` argument to `pthread_mutex_init()` does not refer to an initialized mutex attributes object, it is recommended that the function should fail and report an [EINVAL] error.

## Alternate Implementations Possible

This volume of POSIX.1-2024 supports several alternative implementations of mutexes. An implementation may store the lock directly in the object of type `pthread_mutex_t`. Alternatively, an implementation may store the lock in the heap and merely store a pointer, handle, or unique ID in the mutex object. Either implementation has advantages or may be required on certain hardware configurations. So that portable code can be written that is invariant to this choice, this volume of POSIX.1-2024 does not define assignment or equality for this type, and it uses the term "initialize" to reinforce the (more restrictive) notion that the lock may actually reside in the mutex object itself.

Note that this precludes an over-specification of the type of the mutex or condition variable and motivates the opaqueness of the type.

An implementation is permitted, but not required, to have `pthread_mutex_destroy()` store an illegal value into the mutex. This may help detect erroneous programs that try to lock (or otherwise reference) a mutex that has already been destroyed.

## Tradeoff Between Error Checks and Performance Supported

Many error conditions that can occur are not required to be detected by the implementation in order to let implementations trade off performance versus degree of error checking according to the needs of their specific applications and execution environment. As a general rule, conditions caused by the system (such

as insufficient memory) are required to be detected, but conditions caused by an erroneously coded application (such as failing to provide adequate synchronization to prevent a mutex from being deleted while in use) are specified to result in undefined behavior.

A wide range of implementations is thus made possible. For example, an implementation intended for application debugging may implement all of the error checks, but an implementation running a single, provably correct application under very tight performance constraints in an embedded computer might implement minimal checks. An implementation might even be provided in two versions, similar to the options that compilers provide: a full-checking, but slower version; and a limited-checking, but faster version. To forbid this optionality would be a disservice to users.

By carefully limiting the use of "undefined behavior" only to things that an erroneous (badly coded) application might do, and by defining that resource-not-available errors are mandatory, this volume of POSIX.1-2024 ensures that a fully-conforming application is portable across the full range of implementations, while not forcing all implementations to add overhead to check for numerous things that a correct program never does. When the behavior is undefined, no error number is specified to be returned on implementations that do detect the condition. This is because undefined behavior means anything can happen, which includes returning with any value (which might happen to be a valid, but different, error number). However, since the error number might be useful to application developers when diagnosing problems during application development, a recommendation is made in rationale that implementors should return a particular error number if their implementation does detect the condition.

## Why No Limits are Defined

Defining symbols for the maximum number of mutexes and condition variables was considered but rejected because the number of these objects may change dynamically. Furthermore, many implementations place these objects into application memory; thus, there is no explicit maximum.

## Static Initializers for Mutexes and Condition Variables

Providing for static initialization of statically allocated synchronization objects allows modules with private static synchronization variables to avoid runtime initialization tests and overhead. Furthermore, it simplifies the coding of self-initializing modules. Such modules are common in C libraries, where for various reasons the design calls for self-initialization instead of requiring an explicit module initialization function to be called. An example use of static initialization follows.

Without static initialization, a self-initializing routine `foo()` might look as follows:

```

static pthread_once_t foo_once = PTHREAD_ONCE_INIT;
static pthread_mutex_t foo_mutex;

void foo_init()
{
    pthread_mutex_init(&foo_mutex, NULL);
}

void foo()
{
    pthread_once(&foo_once, foo_init);
    pthread_mutex_lock(&foo_mutex);
    /* Do work. */
    pthread_mutex_unlock(&foo_mutex);
}

```

With static initialization, the same routine could be coded as follows:

```

static pthread_mutex_t foo_mutex = PTHREAD_MUTEX_INITIALIZER;

void foo()
{
    pthread_mutex_lock(&foo_mutex);
    /* Do work. */
    pthread_mutex_unlock(&foo_mutex);
}

```

Note that the static initialization both eliminates the need for the initialization test inside `pthread_once()` and the fetch of `foo_mutex` to learn the address to be passed to `pthread_mutex_lock()` or `pthread_mutex_unlock()`.

Thus, the C code written to initialize static objects is simpler on all systems and is also faster on a large class of systems; those where the (entire) synchronization object can be stored in application memory.

Yet the locking performance question is likely to be raised for machines that require mutexes to be allocated out of special memory. Such machines actually have to have mutexes and possibly condition variables contain pointers to the actual hardware locks. For static initialization to work on such machines, `pthread_mutex_lock()` also has to test whether or not the pointer to the actual lock has been allocated. If it has not, `pthread_mutex_lock()` has to initialize it before use. The reservation of such resources can be made when the program is loaded, and hence return codes have not been added to mutex locking and condition variable waiting to indicate failure to complete initialization.

This runtime test in `pthread_mutex_lock()` would at first seem to be extra work; an extra test is required to see whether the pointer has been initialized. On most machines this would actually be implemented as a fetch of the pointer, testing the

pointer against zero, and then using the pointer if it has already been initialized. While the test might seem to add extra work, the extra effort of testing a register is usually negligible since no extra memory references are actually done. As more and more machines provide caches, the real expenses are memory references, not instructions executed.

Alternatively, depending on the machine architecture, there are often ways to eliminate all overhead in the most important case: on the lock operations that occur after the lock has been initialized. This can be done by shifting more overhead to the less frequent operation: initialization. Since out-of-line mutex allocation also means that an address has to be dereferenced to find the actual lock, one technique that is widely applicable is to have static initialization store a bogus value for that address; in particular, an address that causes a machine fault to occur. When such a fault occurs upon the first attempt to lock such a mutex, validity checks can be done, and then the correct address for the actual lock can be filled in. Subsequent lock operations incur no extra overhead since they do not "fault". This is merely one technique that can be used to support static initialization, while not adversely affecting the performance of lock acquisition. No doubt there are other techniques that are highly machine-dependent.

The locking overhead for machines doing out-of-line mutex allocation is thus similar for modules being implicitly initialized, where it is improved for those doing mutex allocation entirely inline. The inline case is thus made much faster, and the out-of-line case is not significantly worse.

Besides the issue of locking performance for such machines, a concern is raised that it is possible that threads would serialize contending for initialization locks when attempting to finish initializing statically allocated mutexes. (Such finishing would typically involve taking an internal lock, allocating a structure, storing a pointer to the structure in the mutex, and releasing the internal lock.) First, many implementations would reduce such serialization by hashing on the mutex address. Second, such serialization can only occur a bounded number of times. In particular, it can happen at most as many times as there are statically allocated synchronization objects. Dynamically allocated objects would still be initialized via `pthread_mutex_init()` or `pthread_cond_init()`.

Finally, if none of the above optimization techniques for out-of-line allocation yields sufficient performance for an application on some implementation, the application can avoid static initialization altogether by explicitly initializing all synchronization objects with the corresponding `pthread_*_init()` functions, which are supported by all implementations. An implementation can also document the tradeoffs and advise which initialization technique is more efficient for that particular implementation.

## Destroying Mutexes

A mutex can be destroyed immediately after it is unlocked. However, since attempting to destroy a locked mutex, or a mutex that another thread is attempting to lock, or a mutex that is being used in a `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, or `pthread_cond_wait()` call by another thread, results in undefined behavior, care must be taken to ensure that no other thread may be referencing the mutex.

## Robust Mutexes

Implementations are required to provide robust mutexes for mutexes with the process-shared attribute set to PTHREAD\_PROCESS\_SHARED. Implementations are allowed, but not required, to provide robust mutexes when the process-shared attribute is set to PTHREAD\_PROCESS\_PRIVATE.

## FUTURE DIRECTIONS

None.

## SEE ALSO

`pthread_mutex_getprioceiling()`,  
`pthread_mutex_clockclock()`,  
`pthread_mutexattr_getpshared()`

`pthread_mutexattr_getrobust()`,  
`pthread_mutex_lock()`,

XBD `<pthread.h>`

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

## Issue 6

The `pthread_mutex_destroy()` and `pthread_mutex_init()` functions are marked as part of the Threads option.

The `pthread_mutex_timedlock()` function is added to the SEE ALSO section for alignment with IEEE Std 1003.1-2001.

## 1.154. pthread\_mutex\_lock

### SYNOPSIS

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

### DESCRIPTION

The mutex object referenced by *mutex* shall be locked by a call to `pthread_mutex_lock()` that returns zero or [EOWNERDEAD]. If the mutex is already locked by another thread, the calling thread shall block until the mutex becomes available. This operation shall return with the mutex object referenced by *mutex* in the locked state with the calling thread as its owner. If a thread attempts to relock a mutex that it has already locked, `pthread_mutex_lock()` shall behave as described in the **Relock** column of the following table. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, `pthread_mutex_unlock()` shall behave as described in the **Unlock When Not Owner** column of the following table.

Mutex Type	Robustness	Relock	Unlock When Not Owner
NORMAL	non-robust	deadlock	undefined behavior
NORMAL	robust	deadlock	error returned
ERRORCHECK	either	error returned	error returned
RECURSIVE	either	recursive (see below)	error returned
DEFAULT	non-robust	undefined behavior†	undefined behavior†
DEFAULT	robust	undefined behavior†	error returned

† If the mutex type is PTHREAD\_MUTEX\_DEFAULT, the behavior of `pthread_mutex_lock()` may correspond to one of the three other standard mutex types as described in the table above. If it does not correspond to one of those three, the behavior is undefined for the cases marked †.

Where the table indicates recursive behavior, the mutex shall maintain the concept of a lock count. When a thread successfully acquires a mutex for the first time, the lock count shall be set to one. Every time a thread relocks this mutex, the lock count shall be incremented by one. Each time the thread unlocks the mutex, the lock count shall be decremented by one. When the lock count reaches zero, the mutex shall become available for other threads to acquire.

The `pthread_mutex_trylock()` function shall be equivalent to `pthread_mutex_lock()`, except that if the mutex object referenced by *mutex* is currently locked (by any thread, including the current thread), the call shall return immediately. If the mutex type is PTHREAD\_MUTEX\_RECURSIVE and the mutex is currently owned by the calling thread, the mutex lock count shall be incremented by one and the `pthread_mutex_trylock()` function shall immediately return success.

The `pthread_mutex_unlock()` function shall release the mutex object referenced by *mutex*. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by *mutex* when `pthread_mutex_unlock()` is called, resulting in the mutex becoming available, the scheduling policy shall determine which thread shall acquire the mutex.

(In the case of PTHREAD\_MUTEX\_RECURSIVE mutexes, the mutex shall become available when the count reaches zero and the calling thread no longer has any locks on this mutex.)

If a signal is delivered to a thread waiting for a mutex, upon return from the signal handler the thread shall resume waiting for the mutex as if it was not interrupted.

If *mutex* is a robust mutex and the process containing the owning thread terminated while holding the mutex lock, a call to `pthread_mutex_lock()` shall return the error value [EOWNERDEAD]. If *mutex* is a robust mutex and the owning thread terminated while holding the mutex lock, a call to `pthread_mutex_lock()` may return the error value [EOWNERDEAD] even if the process in which the owning thread resides has not terminated. In these cases, the mutex shall be locked by the calling thread but the state it protects is marked as inconsistent. The application should ensure that the state is made consistent for reuse and when that is complete call `pthread_mutex_consistent()`. If the application is unable to recover the state, it should unlock the mutex without a prior call to `pthread_mutex_consistent()`, after which the mutex is marked permanently unusable.

If *mutex* does not refer to an initialized mutex object, the behavior of `pthread_mutex_lock()`, `pthread_mutex_trylock()`, and `pthread_mutex_unlock()` is undefined.

## RETURN VALUE

If successful, the `pthread_mutex_lock()`, `pthread_mutex_trylock()`, and `pthread_mutex_unlock()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

## ERRORS

The `pthread_mutex_lock()` and `pthread_mutex_trylock()` functions shall fail if:

- **[EAGAIN]**
  - The mutex could not be acquired because the maximum number of recursive locks for *mutex* has been exceeded.
  - The mutex is a robust mutex and the system resources available for robust mutexes owned would be exceeded.
- **[EINVAL] [RPP|TPP]**
  - The *mutex* was created with the protocol attribute having the value `PTHREAD_PRIO_PROTECT` and the calling thread's priority is higher than the mutex's current priority ceiling.
- **[ENOTRECOVERABLE]**
  - The state protected by the mutex is not recoverable.
- **[EOWNERDEAD]**
  - The mutex is a robust mutex and the process containing the previous owning thread terminated while holding the mutex lock. The mutex lock shall be acquired by the calling thread and it is up to the new owner to make the state consistent.

The `pthread_mutex_lock()` function shall fail if:

- **[EDEADLK]**
  - The mutex type is `PTHREAD_MUTEX_ERRORCHECK` and the current thread already owns the mutex.

The `pthread_mutex_trylock()` function shall fail if:

- **[EBUSY]**

- The *mutex* could not be acquired because it was already locked.

The `pthread_mutex_unlock()` function shall fail if:

- **[EPERM]**

- The mutex type is PTHREAD\_MUTEX\_ERRORCHECK or PTHREAD\_MUTEX\_RECURSIVE, or the mutex is a robust mutex, and the current thread does not own the mutex.

The `pthread_mutex_lock()` and `pthread_mutex_trylock()` functions may fail if:

- **[EOWNERDEAD]**

- The mutex is a robust mutex and the previous owning thread terminated while holding the mutex lock. The mutex lock shall be acquired by the calling thread and it is up to the new owner to make the state consistent.

The `pthread_mutex_lock()` function may fail if:

- **[EDEADLK]**

- A deadlock condition was detected.

These functions shall not return an error code of [EINTR].

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

Applications that have assumed that non-zero return values are errors will need updating for use with robust mutexes, since a valid return for a thread acquiring a mutex which is protecting a currently inconsistent state is [EOWNERDEAD]. Applications that do not check the error returns, due to ruling out the possibility of such errors arising, should not use robust mutexes. If an application is supposed to work with normal and robust mutexes it should check all return values for error conditions and if necessary take appropriate action.

## RATIONALE

Mutex objects are intended to serve as a low-level primitive from which other thread synchronization functions can be built. As such, the implementation of mutexes should be as efficient as possible, and this has ramifications on the features available at the interface.

The mutex functions and the particular default settings of the mutex attributes have been motivated by the desire to not preclude fast, inlined implementations of mutex locking and unlocking.

Since most attributes only need to be checked when a thread is going to be blocked, the use of attributes does not slow the (common) mutex-locking case.

Likewise, while being able to extract the thread ID of the owner of a mutex might be desirable, it would require storing the current thread ID when each mutex is locked, and this could incur unacceptable levels of overhead. Similar arguments apply to a `mutex_tryunlock` operation.

For further rationale on the extended mutex types, see XRAT *Threads Extensions*.

If an implementation detects that the value specified by the `mutex` argument does not refer to an initialized mutex object, it is recommended that the function should fail and report an [EINVAL] error.

## FUTURE DIRECTIONS

None.

## SEE ALSO

`pthread_mutex_clocklock()` , `pthread_mutex_consistent()` ,  
`pthread_mutex_destroy()` , `pthread_mutexattr_getrobust()`

XBD 4.15.2 *Memory Synchronization* , `<pthread.h>`

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

## Issue 6

The `pthread_mutex_lock()`, `pthread_mutex_trylock()`, and `pthread_mutex_unlock()` functions are marked as part of the Threads option.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The behavior when attempting to relock a mutex is defined.

The `pthread_mutex_timedlock()` function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/98 is applied, updating the ERRORS section so that the [EDEADLK] error includes detection of a deadlock condition. The RATIONALE section is also reworded to take into account non-XSI-conformant systems.

## Issue 7

SD5-XSH-ERN-43 is applied, marking the "shall fail" case of the [EINVAL] error as dependent on the Thread Priority Protection option.

Changes are made from The Open Group Technical Standard, 2006, Extended API Set Part 3.

The `pthread_mutex_lock()`, `pthread_mutex_trylock()`, and `pthread_mutex_unlock()` functions are moved from the Threads option to the Base.

The following extended mutex types are moved from the XSI option to the Base:

```
PTHREAD_MUTEX_NORMAL
PTHREAD_MUTEX_ERRORCHECK
PTHREAD_MUTEX_RECURSIVE
PTHREAD_MUTEX_DEFAULT
```

The DESCRIPTION is updated to clarify the behavior when *mutex* does not refer to an initialized mutex.

The ERRORS section is updated to account properly for all of the various mutex types.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0461 [121], XSH/TC1-2008/0462 [92,428], and XSH/TC1-2008/0463 [121] are applied.

## Issue 8

Austin Group Defect 354 is applied, adding the [EAGAIN] error for exceeding system resources available for robust mutexes owned.

Austin Group Defect 1115 is applied, changing "the thread" to "the calling thread".

---

## 1.155. pthread\_mutex\_getprioceiling, pthread\_mutex\_setprioceiling

---

### SYNOPSIS

```
#include <pthread.h>

int pthread_mutex_getprioceiling(const pthread_mutex_t *restrict mutex,
                                  int *restrict prioceiling);

int pthread_mutex_setprioceiling(pthread_mutex_t *restrict mutex,
                                 int prioceiling, int *restrict old_ceiling);
```

### DESCRIPTION

The `pthread_mutex_getprioceiling()` function shall return the current priority ceiling of the mutex.

The `pthread_mutex_setprioceiling()` function shall attempt to lock the mutex as if by a call to `pthread_mutex_lock()`, except that the process of locking the mutex need not adhere to the priority protect protocol. On acquiring the mutex it shall change the mutex's priority ceiling and then release the mutex as if by a call to `pthread_mutex_unlock()`. When the change is successful, the previous value of the priority ceiling shall be returned in `old_ceiling`.

If the `pthread_mutex_setprioceiling()` function fails, the mutex priority ceiling shall not be changed.

### RETURN VALUE

If successful, the `pthread_mutex_getprioceiling()` and `pthread_mutex_setprioceiling()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

### ERRORS

These functions shall fail if:

- **[EINVAL]**

The protocol attribute of `mutex` is PTHREAD\_PRIO\_NONE.

- **[EPERM]**

The implementation requires appropriate privileges to perform the operation and the caller does not have appropriate privileges.

The `pthread_mutex_setprioceiling()` function shall fail if:

- **[EAGAIN]**

The mutex could not be acquired because the maximum number of recursive locks for `mutex` has been exceeded.

- **[EAGAIN]**

The mutex is a robust mutex and the system resources available for robust mutexes owned would be exceeded.

- **[EDEADLK]**

The mutex type is `PTHREAD_MUTEX_ERRORCHECK` and the current thread already owns the mutex.

- **[EINVAL]**

The mutex was created with the protocol attribute having the value `PTHREAD_PRIO_PROTECT` and the calling thread's priority is higher than the mutex's current priority ceiling, and the implementation adheres to the priority protect protocol in the process of locking the mutex.

- **[ENOTRECOVERABLE]**

The mutex is a robust mutex and the state protected by the mutex is not recoverable.

- **[EOWNERDEAD]**

The mutex is a robust mutex and the process containing the previous owning thread terminated while holding the mutex lock. The mutex lock shall be acquired by the calling thread and it is up to the new owner to make the state consistent (see `pthread_mutex_lock()` ).

The `pthread_mutex_setprioceiling()` function may fail if:

- **[EDEADLK]**

A deadlock condition was detected.

- **[EINVAL]**

The priority requested by `prioceiling` is out of range.

- **[EOWNERDEAD]**

The mutex is a robust mutex and the previous owning thread terminated while holding the mutex lock. The mutex lock shall be acquired by the calling thread and it is up to the new owner to make the state consistent (see `pthread_mutex_lock()` ).

These functions shall not return an error code of [EINTR].

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_mutex_clocklock()`
- `pthread_mutex_destroy()`
- `pthread_mutex_lock()`
- `<pthread.h>`

## CHANGE HISTORY

### First released in Issue 5

Included for alignment with the POSIX Threads Extension. Marked as part of the Realtime Threads Feature Group.

### Issue 6

- The `pthread_mutex_getprioceiling()` and `pthread_mutex_setprioceiling()` functions are marked as part of the

Threads and Thread Priority Protection options.

- The [ENOSYS] error conditions have been removed.
- The `pthread_mutex_timedlock()` function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.
- The `restrict` keyword is added to the `pthread_mutex_getprioceiling()` and `pthread_mutex_setprioceiling()` prototypes for alignment with the ISO/IEC 9899:1999 standard.

## Issue 7

- SD5-XSH-ERN-39 is applied.
- Austin Group Interpretation 1003.1-2001 #052 is applied, adding [EDEADLK] as a "may fail" error.
- SD5-XSH-ERN-158 is applied, updating the ERRORS section to include a "shall fail" error case for when the protocol attribute of `mutex` is `PTHREAD_PRIO_NONE`.
- The `pthread_mutex_getprioceiling()` and `pthread_mutex_setprioceiling()` functions are moved from the Threads option to require support of either the Robust Mutex Priority Protection option or the Non-Robust Mutex Priority Protection option.
- The DESCRIPTION and ERRORS sections are updated to account properly for all of the various mutex types.

## Issue 8

- Austin Group Defect 354 is applied, adding the [EAGAIN] error for exceeding system resources available for robust mutexes owned.
-

## 1.156. pthread\_mutex\_trylock - 锁定和解锁互斥锁

### 概要

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

### 描述

由 `mutex` 引用的互斥锁对象应通过调用 `pthread_mutex_lock()` 来锁定，该调用返回零或 `[EOWNERDEAD]`。如果互斥锁已被另一个线程锁定，调用线程将阻塞直到互斥锁变得可用。此操作应返回时，`mutex` 引用的互斥锁对象处于锁定状态，调用线程为其所有者。如果线程尝试重新锁定它已经锁定的互斥锁，`pthread_mutex_lock()` 的行为应如下表重新锁定列所述。如果线程尝试解锁它没有锁定的互斥锁或未锁定的互斥锁，`pthread_mutex_unlock()` 的行为应如下表非所有者解锁列所述。

互斥锁类型	健壮性	重新锁定	非所有者解锁
NORMAL	非健壮	死锁	未定义行为
NORMAL	健壮	死锁	返回错误
ERRORCHECK	任意	返回错误	返回错误
RECURSIVE	任意	递归（见下文）	返回错误
DEFAULT	非健壮	未定义行为†	未定义行为†
DEFAULT	健壮	未定义行为†	返回错误

† 如果互斥锁类型是 `PTHREAD_MUTEX_DEFAULT`，`pthread_mutex_lock()` 的行为可能对应于上表中所述的三种其他标准互斥锁类型之一。如果它不对应于这三种类型之一，则标记为 † 的情况下的行为是未定义的。

在表指示递归行为的情况下，互斥锁应保持锁计数概念。当线程第一次成功获取互斥锁时，锁计数应设置为一。每次线程重新锁定此互斥锁时，锁计数应加一。每次

线程解锁互斥锁时，锁计数应减一。当锁计数达到零时，互斥锁应对其他线程可用以获取。

`pthread_mutex_trylock()` 函数应等价于 `pthread_mutex_lock()`，但如果 `mutex` 引用的互斥锁对象当前被锁定（由任何线程，包括当前线程），调用应立即返回。如果互斥锁类型是 `PTHREAD_MUTEX_RECURSIVE` 且互斥锁当前由调用线程拥有，互斥锁锁计数应加一，`pthread_mutex_trylock()` 函数应立即返回成功。

`pthread_mutex_unlock()` 函数应释放 `mutex` 引用的互斥锁对象。释放互斥锁的方式取决于互斥锁的类型属性。如果在调用 `pthread_mutex_unlock()` 时有线程在 `mutex` 引用的互斥锁对象上阻塞，导致互斥锁变得可用，调度策略应确定哪个线程应获取互斥锁。

（在 `PTHREAD_MUTEX_RECURSIVE` 互斥锁的情况下，当计数达到零且调用线程不再拥有此互斥锁的任何锁时，互斥锁应变为可用。）

如果信号传递给等待互斥锁的线程，从信号处理程序返回后，线程应恢复等待互斥锁，就像它未被中断一样。

如果 `mutex` 是健壮互斥锁且拥有互斥锁的进程在保持互斥锁时终止，调用 `pthread_mutex_lock()` 应返回错误值 `[EOWNERDEAD]`。如果 `mutex` 是健壮互斥锁且拥有线程在保持互斥锁时终止，即使拥有线程所在的进程尚未终止，调用 `pthread_mutex_lock()` 也可能返回错误值 `[EOWNERDEAD]`。在这些情况下，互斥锁应由调用线程锁定，但它保护的状态被标记为不一致。应用程序应确保状态变为一致以供重用，完成后调用 `pthread_mutex_consistent()`。如果应用程序无法恢复状态，应在未调用 `pthread_mutex_consistent()` 的情况下解锁互斥锁，此后互斥锁被标记为永久不可用。

如果 `mutex` 不引用已初始化的互斥锁对象，`pthread_mutex_lock()`、`pthread_mutex_trylock()` 和 `pthread_mutex_unlock()` 的行为是未定义的。

## 返回值

如果成功，`pthread_mutex_lock()`、`pthread_mutex_trylock()` 和 `pthread_mutex_unlock()` 函数应返回零；否则，应返回错误编号以指示错误。

## 错误

`pthread_mutex_lock()` 和 `pthread_mutex_trylock()` 函数应在以下情况下失败：

- `[EAGAIN]` — 互斥锁无法获取，因为超过了 `mutex` 的递归锁最大次数。
- `[EAGAIN]` — 互斥锁是健壮互斥锁，可用的健壮互斥锁系统资源将被超出。

- `[EINVAL]` — `[RPP|TPP]` `mutex` 是用协议属性值为 `PTHREAD_PRIO_PROTECT` 创建的，调用线程的优先级高于互斥锁的当前优先级上限。
- `[ENOTRECOVERABLE]` — 互斥锁保护的状态不可恢复。
- `[EOWNERDEAD]` — 互斥锁是健壮互斥锁，包含先前拥有线程的进程在保持互斥锁时终止。互斥锁应由调用线程获取，由新所有者负责使状态一致。

`pthread_mutex_lock()` 函数应在以下情况下失败：

- `[EDEADLK]` — 互斥锁类型是 `PTHREAD_MUTEX_ERRORCHECK` 且当前线程已经拥有互斥锁。

`pthread_mutex_trylock()` 函数应在以下情况下失败：

- `[EBUSY]` — `mutex` 无法获取，因为它已被锁定。

`pthread_mutex_unlock()` 函数应在以下情况下失败：

- `[EPERM]` — 互斥锁类型是 `PTHREAD_MUTEX_ERRORCHECK` 或 `PTHREAD_MUTEX_RECURSIVE`，或互斥锁是健壮互斥锁，且当前线程不拥有互斥锁。

`pthread_mutex_lock()` 和 `pthread_mutex_trylock()` 函数可能在以下情况下失败：

- `[EOWNERDEAD]` — 互斥锁是健壮互斥锁，先前拥有线程在保持互斥锁时终止。互斥锁应由调用线程获取，由新所有者负责使状态一致。

`pthread_mutex_lock()` 函数可能在以下情况下失败：

- `[EDEADLK]` — 检测到死锁条件。

这些函数不应返回 `[EINTR]` 错误代码。

## 应用程序用法

假设非零返回值是错误的应用程序将需要更新以与健壮互斥锁一起使用，因为线程获取保护当前不一致状态的互斥锁的有效返回是 `[EOWNERDEAD]`。由于排除了此类错误出现的可能性而不检查错误返回的应用程序不应使用健壮互斥锁。如果应用程序应该与普通和健壮互斥锁一起工作，它应检查所有错误条件的返回值，并在必要时采取适当行动。

## 基本原理

互斥锁对象旨在作为可以构建其他线程同步函数的低级原语。因此，互斥锁的实现应尽可能高效，这对接口上可用功能有影响。

互斥锁函数和互斥锁属性的特定默认设置受到不排除快速、内联互斥锁锁定和解锁实现的愿望的激励。

由于大多数属性只需要在线程将被阻塞时检查，属性的使用不会减慢（常见）互斥锁锁定情况。

同样，虽然能够提取互斥锁拥有者的线程 ID 可能是可取的，但这需要在锁定每个互斥锁时存储当前线程 ID，这可能导致不可接受的级别开销。类似的论点适用于 `mutex_tryunlock` 操作。

有关扩展互斥锁类型的进一步基本原理，请参见 X RAT 线程扩展。

如果实现检测到 `mutex` 参数指定的值不引用已初始化的互斥锁对象，建议函数应失败并报告 `[EINVAL]` 错误。

## 另见

- `pthread_mutex_clocklock()`
- `pthread_mutex_consistent()`
- `pthread_mutex_destroy()`
- `pthread_mutexattr_getrobust()`
- 4.15.2 内存同步
- `<pthread.h>`

## 变更历史

首次在 Issue 5 中发布。包含用于与 POSIX 线程扩展对齐。

### Issue 6

`pthread_mutex_lock()` 、 `pthread_mutex_trylock()` 和 `pthread_mutex_unlock()` 函数被标记为线程选项的一部分。

以下对 POSIX 实现的新要求源于与单一 UNIX 规范的对齐：

- 重新锁定互斥锁时的行为被定义。

`pthread_mutex_timedlock()` 函数被添加到另见部分，以便与 IEEE Std 1003.1d-1999 对齐。

应用 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/98，更新错误部分，使 `[EDEADLK]` 错误包含死锁条件的检测。基本原理部分也被重新措辞，以考虑非 XSI 符合系统。

## Issue 7

应用 SD5-XSH-ERN-43，将 `[EINVAL]` 错误的"应失败"情况标记为依赖于线程优先级保护选项。

从 The Open Group 技术标准 2006 扩展 API 集第 3 部分进行更改。

`pthread_mutex_lock()` 、 `pthread_mutex_trylock()` 和 `pthread_mutex_unlock()` 函数从线程选项移动到基本。

以下扩展互斥锁类型从 XSI 选项移动到基本：

- `PTHREAD_MUTEX_NORMAL`
- `PTHREAD_MUTEX_ERRORCHECK`
- `PTHREAD_MUTEX_RECURSIVE`
- `PTHREAD_MUTEX_DEFAULT`

描述被更新以澄清当 `mutex` 不引用已初始化互斥锁时的行为。

错误部分被更新以正确处理所有各种互斥锁类型。

应用 POSIX.1-2008，技术勘误 1，XSH/TC1-2008/0461 [121]、XSH/TC1-2008/0462 [92,428] 和 XSH/TC1-2008/0463 [121]。

## Issue 8

应用 Austin Group Defect 354，为超出可用的健壮互斥锁系统资源添加 `[EAGAIN]` 错误。

应用 Austin Group Defect 1115，将"该线程"更改为"调用线程"。

## 1.157. `pthread_mutex_lock`, `pthread_mutex_trylock`, `pthread_mutex_unlock`

### SYNOPSIS

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

### DESCRIPTION

The mutex object referenced by `mutex` shall be locked by a call to `pthread_mutex_lock()` that returns zero or `[EOWNERDEAD]`. If the mutex is already locked by another thread, the calling thread shall block until the mutex becomes available. This operation shall return with the mutex object referenced by `mutex` in the locked state with the calling thread as its owner. If a thread attempts to relock a mutex that it has already locked, `pthread_mutex_lock()` shall behave as described in the **Relock** column of the following table. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, `pthread_mutex_unlock()` shall behave as described in the **Unlock When Not Owner** column of the following table.

Mutex Type	Robustness	Relock	Unlock When Not Owner
NORMAL	non-robust	deadlock	undefined behavior
NORMAL	robust	deadlock	error returned
ERRORCHECK	either	error returned	error returned
RECURSIVE	either	recursive (see below)	error returned
DEFAULT	non-robust	undefined behavior†	undefined behavior†

Mutex Type	Robustness	Relock	Unlock When Not Owner
DEFAULT	robust	undefined behavior†	error returned

† If the mutex type is `PTHREAD_MUTEX_DEFAULT`, the behavior of `pthread_mutex_lock()` may correspond to one of the three other standard mutex types as described in the table above. If it does not correspond to one of those three, the behavior is undefined for the cases marked †.

Where the table indicates recursive behavior, the mutex shall maintain the concept of a lock count. When a thread successfully acquires a mutex for the first time, the lock count shall be set to one. Every time a thread relocks this mutex, the lock count shall be incremented by one. Each time the thread unlocks the mutex, the lock count shall be decremented by one. When the lock count reaches zero, the mutex shall become available for other threads to acquire.

The `pthread_mutex_trylock()` function shall be equivalent to `pthread_mutex_lock()`, except that if the mutex object referenced by `mutex` is currently locked (by any thread, including the current thread), the call shall return immediately. If the mutex type is `PTHREAD_MUTEX_RECURSIVE` and the mutex is currently owned by the calling thread, the mutex lock count shall be incremented by one and the `pthread_mutex_trylock()` function shall immediately return success.

The `pthread_mutex_unlock()` function shall release the mutex object referenced by `mutex`. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by `mutex` when `pthread_mutex_unlock()` is called, resulting in the mutex becoming available, the scheduling policy shall determine which thread shall acquire the mutex.

(In the case of `PTHREAD_MUTEX_RECURSIVE` mutexes, the mutex shall become available when the count reaches zero and the calling thread no longer has any locks on this mutex.)

If a signal is delivered to a thread waiting for a mutex, upon return from the signal handler the thread shall resume waiting for the mutex as if it was not interrupted.

If `mutex` is a robust mutex and the process containing the owning thread terminated while holding the mutex lock, a call to `pthread_mutex_lock()` shall return the error value `[EOWNERDEAD]`. If `mutex` is a robust mutex and the owning thread terminated while holding the mutex lock, a call to `pthread_mutex_lock()` may return the error value `[EOWNERDEAD]` even if the process in which the owning thread resides has not terminated. In these cases, the

mutex shall be locked by the calling thread but the state it protects is marked as inconsistent. The application should ensure that the state is made consistent for reuse and when that is complete call `pthread_mutex_consistent()`. If the application is unable to recover the state, it should unlock the mutex without a prior call to `pthread_mutex_consistent()`, after which the mutex is marked permanently unusable.

If `mutex` does not refer to an initialized mutex object, the behavior of `pthread_mutex_lock()`, `pthread_mutex_trylock()`, and `pthread_mutex_unlock()` is undefined.

## RETURN VALUE

If successful, the `pthread_mutex_lock()`, `pthread_mutex_trylock()`, and `pthread_mutex_unlock()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

## ERRORS

The `pthread_mutex_lock()` and `pthread_mutex_trylock()` functions shall fail if:

- **[EAGAIN]** The mutex could not be acquired because the maximum number of recursive locks for `mutex` has been exceeded.
- **[EAGAIN]** The mutex is a robust mutex and the system resources available for robust mutexes owned would be exceeded.
- **[EINVAL] [RPP|TPP]** The `mutex` was created with the protocol attribute having the value `PTHREAD_PRIO_PROTECT` and the calling thread's priority is higher than the mutex's current priority ceiling.
- **[ENOTRECOVERABLE]** The state protected by the mutex is not recoverable.
- **[EOWNERDEAD]** The mutex is a robust mutex and the process containing the previous owning thread terminated while holding the mutex lock. The mutex lock shall be acquired by the calling thread and it is up to the new owner to make the state consistent.

The `pthread_mutex_lock()` function shall fail if:

- **[EDEADLK]** The mutex type is `PTHREAD_MUTEX_ERRORCHECK` and the current thread already owns the mutex.

The `pthread_mutex_trylock()` function shall fail if:

- **[EBUSY]** The `mutex` could not be acquired because it was already locked.

The `pthread_mutex_unlock()` function shall fail if:

- **[EPERM]** The mutex type is `PTHREAD_MUTEX_ERRORCHECK` or `PTHREAD_MUTEX_RECURSIVE`, or the mutex is a robust mutex, and the current thread does not own the mutex.

The `pthread_mutex_lock()` and `pthread_mutex_trylock()` functions may fail if:

- **[EOWNERDEAD]** The mutex is a robust mutex and the previous owning thread terminated while holding the mutex lock. The mutex lock shall be acquired by the calling thread and it is up to the new owner to make the state consistent.

The `pthread_mutex_lock()` function may fail if:

- **[EDEADLK]** A deadlock condition was detected.

These functions shall not return an error code of `[EINTR]`.

## EXAMPLES

None.

## APPLICATION USAGE

Applications that have assumed that non-zero return values are errors will need updating for use with robust mutexes, since a valid return for a thread acquiring a mutex which is protecting a currently inconsistent state is `[EOWNERDEAD]`. Applications that do not check the error returns, due to ruling out the possibility of such errors arising, should not use robust mutexes. If an application is supposed to work with normal and robust mutexes it should check all return values for error conditions and if necessary take appropriate action.

## RATIONALE

Mutex objects are intended to serve as a low-level primitive from which other thread synchronization functions can be built. As such, the implementation of mutexes should be as efficient as possible, and this has ramifications on the features available at the interface.

The mutex functions and the particular default settings of the mutex attributes have been motivated by the desire to not preclude fast, inlined implementations of mutex locking and unlocking.

Since most attributes only need to be checked when a thread is going to be blocked, the use of attributes does not slow the (common) mutex-locking case.

Likewise, while being able to extract the thread ID of the owner of a mutex might be desirable, it would require storing the current thread ID when each mutex is locked, and this could incur unacceptable levels of overhead. Similar arguments apply to a `mutex_tryunlock` operation.

For further rationale on the extended mutex types, see XRAT [Threads Extensions](#).

If an implementation detects that the value specified by the `mutex` argument does not refer to an initialized mutex object, it is recommended that the function should fail and report an `[EINVAL]` error.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_mutex_clocklock()`
- `pthread_mutex_consistent()`
- `pthread_mutex_destroy()`
- `pthread_mutexattr_getrobust()`
- XBD [4.15.2 Memory Synchronization](#)
- 

## CHANGE HISTORY

### Issue 6

The `pthread_mutex_lock()`, `pthread_mutex_trylock()`, and `pthread_mutex_unlock()` functions are marked as part of the Threads option.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The behavior when attempting to relock a mutex is defined.

The `pthread_mutex_timedlock()` function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/98 is applied, updating the ERRORS section so that the **[EDEADLK]** error includes detection of a deadlock condition. The RATIONALE section is also reworded to take into account non-XSI-conformant systems.

## Issue 7

SD5-XSH-ERN-43 is applied, marking the "shall fail" case of the **[EINVAL]** error as dependent on the Thread Priority Protection option.

Changes are made from The Open Group Technical Standard, 2006, Extended API Set Part 3.

The `pthread_mutex_lock()`, `pthread_mutex_trylock()`, and `pthread_mutex_unlock()` functions are moved from the Threads option to the Base.

The following extended mutex types are moved from the XSI option to the Base:

- **PTHREAD\_MUTEX\_NORMAL**
- **PTHREAD\_MUTEX\_ERRORCHECK**
- **PTHREAD\_MUTEX\_RECURSIVE**
- **PTHREAD\_MUTEX\_DEFAULT**

The DESCRIPTION is updated to clarify the behavior when `mutex` does not refer to an initialized mutex.

The ERRORS section is updated to account properly for all of the various mutex types.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0461 [121], XSH/TC1-2008/0462 [92,428], and XSH/TC1-2008/0463 [121] are applied.

## Issue 8

Austin Group Defect 354 is applied, adding the **[EAGAIN]** error for exceeding system resources available for robust mutexes owned.

Austin Group Defect 1115 is applied, changing "the thread" to "the calling thread".

## 1.158. pthread\_mutexattr\_destroy

---

### SYNOPSIS

```
#include <pthread.h>

int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
```

### DESCRIPTION

The `pthread_mutexattr_destroy()` function shall destroy a mutex attributes object; the object becomes, in effect, uninitialized. An implementation may cause `pthread_mutexattr_destroy()` to set the object referenced by `attr` to an invalid value.

A destroyed `attr` attributes object can be reinitialized using `pthread_mutexattr_init()`; the results of otherwise referencing the object after it has been destroyed are undefined.

The `pthread_mutexattr_init()` function shall initialize a mutex attributes object `attr` with the default value for all of the attributes defined by the implementation.

Results are undefined if `pthread_mutexattr_init()` is called specifying an already initialized `attr` attributes object.

After a mutex attributes object has been used to initialize one or more mutexes, any function affecting the attributes object (including destruction) shall not affect any previously initialized mutexes.

The behavior is undefined if the value specified by the `attr` argument to `pthread_mutexattr_destroy()` does not refer to an initialized mutex attributes object.

### RETURN VALUE

Upon successful completion, `pthread_mutexattr_destroy()` and `pthread_mutexattr_init()` shall return zero; otherwise, an error number shall be returned to indicate the error.

# ERRORS

The `pthread_mutexattr_destroy()` function may fail if:

- **EINVAL** - The value specified by `attr` does not refer to an initialized mutex attributes object.

The `pthread_mutexattr_init()` function may fail if:

- **ENOMEM** - Insufficient memory exists to initialize the mutex attributes object.

These functions shall not return an error code of [EINTR].

## EXAMPLES

None.

## APPLICATION USAGE

If an implementation detects that the value specified by the `attr` argument to `pthread_mutexattr_destroy()` does not refer to an initialized mutex attributes object, it is recommended that the function should fail and report an [EINVAL] error.

See `pthread_attr_destroy()` for a general explanation of attributes. Attributes objects allow implementations to experiment with useful extensions and permit extension of this volume of POSIX.1-2024 without changing the existing functions. Thus, they provide for future extensibility of this volume of POSIX.1-2024 and reduce the temptation to standardize prematurely on semantics that are not yet widely implemented or understood.

Examples of possible additional mutex attributes that have been discussed are `spin_only`, `limited_spin`, `no_spin`, `recursive`, and `metered`. (To explain what the latter attributes might mean: recursive mutexes would allow for multiple re-locking by the current owner; metered mutexes would transparently keep records of queue length, wait time, and so on.) Since there is not yet wide agreement on the usefulness of these resulting from shared implementation and usage experience, they are not yet specified in this volume of POSIX.1-2024. Mutex attributes objects, however, make it possible to test out these concepts for possible standardization at a later time.

### Mutex Attributes and Performance

Care has been taken to ensure that the default values of the mutex attributes have been defined such that mutexes initialized with the defaults have simple enough

semantics so that the locking and unlocking can be done with the equivalent of a test-and-set instruction (plus possibly a few other basic instructions).

There is at least one implementation method that can be used to reduce the cost of testing at lock-time if a mutex has non-default attributes. One such method that an implementation can employ (and this can be made fully transparent to fully conforming POSIX applications) is to secretly pre-lock any mutexes that are initialized to non-default attributes. Any later attempt to lock such a mutex causes the implementation to branch to the "slow path" as if the mutex were unavailable; then, on the slow path, the implementation can do the "real work" to lock a non-default mutex. The underlying unlock operation is more complicated since the implementation never really wants to release the pre-lock on this kind of mutex. This illustrates that, depending on the hardware, there may be certain optimizations that can be used so that whatever mutex attributes are considered "most frequently used" can be processed most efficiently.

## Process Shared Memory and Synchronization

The existence of memory mapping functions in this volume of POSIX.1-2024 leads to the possibility that an application may allocate the synchronization objects from this section in memory that is accessed by multiple processes (and therefore, by threads of multiple processes).

In order to permit such usage, while at the same time keeping the usual case (that is, usage within a single process) efficient, a `process-shared` option has been defined.

If an implementation supports the `_POSIX_THREAD_PROCESS_SHARED` option, then the `process-shared` attribute can be used to indicate that mutexes or condition variables may be accessed by threads of multiple processes.

The default setting of `PTHREAD_PROCESS_PRIVATE` has been chosen for the `process-shared` attribute so that the most efficient forms of these synchronization objects are created by default.

Synchronization variables that are initialized with the `PTHREAD_PROCESS_PRIVATE` `process-shared` attribute may only be operated on by threads in the process that initialized them. Synchronization variables that are initialized with the `PTHREAD_PROCESS_SHARED` `process-shared` attribute may be operated on by any thread in any process that has access to it. In particular, these processes may exist beyond the lifetime of the initializing process. For example, the following code implements a simple counting semaphore in a mapped file that may be used by many processes.

```
/* sem.h */  
struct semaphore {
```

```
pthread_mutex_t lock;
pthread_cond_t nonzero;
unsigned count;
};

typedef struct semaphore semaphore_t;

semaphore_t *semaphore_create(char *semaphore_name);
semaphore_t *semaphore_open(char *semaphore_name);
void semaphore_post(semaphore_t *semaphore);
void semaphore_wait(semaphore_t *semaphore);
void semaphore_close(semaphore_t *semaphore);
```

```
/* sem.c */
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <pthread.h>
#include "sem.h"

semaphore_t *
semaphore_create(char *semaphore_name)
{
    int fd;
    semaphore_t *semaphore;
    pthread_mutexattr_t psharedm;
    pthread_condattr_t psharedc;

    fd = open(semaphore_name, O_RDWR | O_CREAT | O_EXCL, 0666);
    if (fd < 0)
        return (NULL);
    (void) ftruncate(fd, sizeof(semaphore_t));
    (void) pthread_mutexattr_init(&psharedm);
    (void) pthread_mutexattr_setpshared(&psharedm,
        PTHREAD_PROCESS_SHARED);
    (void) pthread_condattr_init(&psharedc);
    (void) pthread_condattr_setpshared(&psharedc,
        PTHREAD_PROCESS_SHARED);
    semaphore = (semaphore_t *) mmap(NULL, sizeof(semaphore_t),
        PROT_READ | PROT_WRITE, MAP_SHARED,
        fd, 0);
    close (fd);
    (void) pthread_mutex_init(&semaphore->lock, &psharedm);
    (void) pthread_cond_init(&semaphore->nonzero, &psharedc);
    semaphore->count = 0;
    return (semaphore);
}

semaphore_t *
semaphore_open(char *semaphore_name)
```

```

{
    int fd;
    semaphore_t *semap;

    fd = open(semaphore_name, O_RDWR, 0666);
    if (fd < 0)
        return (NULL);
    semap = (semaphore_t *) mmap(NULL, sizeof(semaphore_t),
        PROT_READ | PROT_WRITE, MAP_SHARED,
        fd, 0);
    close (fd);
    return (semap);
}

void
semaphore_post(semaphore_t *semap)
{
    pthread_mutex_lock(&semap->lock);
    if (semap->count == 0)
        pthread_cond_signal(&semap->nonzero);
    semap->count++;
    pthread_mutex_unlock(&semap->lock);
}

void
semaphore_wait(semaphore_t *semap)
{
    pthread_mutex_lock(&semap->lock);
    while (semap->count == 0)
        pthread_cond_wait(&semap->nonzero, &semap->lock);
    semap->count--;
    pthread_mutex_unlock(&semap->lock);
}

void
semaphore_close(semaphore_t *semap)
{
    munmap((void *) semap, sizeof(semaphore_t));
}

```

The following code is for three separate processes that create, post, and wait on a semaphore in the file `/tmp/semaphore`. Once the file is created, the post and wait programs increment and decrement the counting semaphore (waiting and waking as required) even though they did not initialize the semaphore.

```

/* create.c */
#include "pthread.h"
#include "sem.h"

```

```
int
main(void)
{
    semaphore_t *semap;

    semap = semaphore_create("/tmp/semaphore");
    if (semap == NULL)
        exit(1);
    semaphore_close(semap);
    return (0);
}
```

```
/* post.c */
#include "pthread.h"
#include "sem.h"

int
main(void)
{
    semaphore_t *semap;

    semap = semaphore_open("/tmp/semaphore");
    if (semap == NULL)
        exit(1);
    semaphore_post(semap);
    semaphore_close(semap);
    return (0);
}
```

```
/* wait.c */
#include "pthread.h"
#include "sem.h"

int
main(void)
{
    semaphore_t *semap;

    semap = semaphore_open("/tmp/semaphore");
    if (semap == NULL)
        exit(1);
    semaphore_wait(semap);
    semaphore_close(semap);
    return (0);
}
```

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

`pthread_attr_destroy()`,  
`pthread_mutex_init()`

`pthread_mutex_destroy()`,

The Base Definitions volume of POSIX.1-2024, `<pthread.h>`

## CHANGE HISTORY

First released in Issue 5. Included from Issue 6.

Derived from the IEEE POSIX Pthreads Standard 1003.1c-1995 and the POSIX Threads Extension (1003.1j-2000).

## 1.159. pthread\_mutexattr\_getprioceiling, pthread\_mutexattr\_setprioceiling

### SYNOPSIS

```
#include <pthread.h>

int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t *restrict attr,
                                     int *restrict prioceiling);

int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
                                     int prioceiling);
```

### DESCRIPTION

The `pthread_mutexattr_getprioceiling()` and `pthread_mutexattr_setprioceiling()` functions, respectively, shall get and set the priority ceiling attribute of a mutex attributes object pointed to by `attr` which was previously created by the function `pthread_mutexattr_init()`.

The `prioceiling` attribute contains the priority ceiling of initialized mutexes. The values of `prioceiling` are within the maximum range of priorities defined by SCHED\_FIFO.

The `prioceiling` attribute defines the priority ceiling of initialized mutexes, which is the minimum priority level at which the critical section guarded by the mutex is executed. In order to avoid priority inversion, the priority ceiling of the mutex shall be set to a priority higher than or equal to the highest priority of all the threads that may lock that mutex. The values of `prioceiling` are within the maximum range of priorities defined under the SCHED\_FIFO scheduling policy.

The behavior is undefined if the value specified by the `attr` argument to `pthread_mutexattr_getprioceiling()` or `pthread_mutexattr_setprioceiling()` does not refer to an initialized mutex attributes object.

### RETURN VALUE

Upon successful completion, the `pthread_mutexattr_getprioceiling()` and `pthread_mutexattr_setprioceiling()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

# ERRORS

These functions may fail if:

- **[EINVAL]**

The value specified by `prioceiling` is invalid.

- **[EPERM]**

The caller does not have the privilege to perform the operation.

These functions shall not return an error code of [EINTR].

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

If an implementation detects that the value specified by the `attr` argument to `pthread_mutexattr_getprioceiling()` or `pthread_mutexattr_setprioceiling()` does not refer to an initialized mutex attributes object, it is recommended that the function should fail and report an [EINVAL] error.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_cond_destroy()`
- `pthread_create()`
- `pthread_mutex_destroy()`

- `<pthread.h>`

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Marked as part of the Realtime Threads Feature Group.

### Issue 6

The `pthread_mutexattr_getprioceiling()` and `pthread_mutexattr_setprioceiling()` functions are marked as part of the Threads and Thread Priority Protection options.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Thread Priority Protection option.

The [ENOTSUP] error condition has been removed since these functions do not have a `protocol` argument.

The `restrict` keyword is added to the `pthread_mutexattr_getprioceiling()` prototype for alignment with the ISO/IEC 9899:1999 standard.

### Issue 7

The `pthread_mutexattr_getprioceiling()` and `pthread_mutexattr_setprioceiling()` functions are moved from the Threads option to require support of either the Robust Mutex Priority Protection option or the Non-Robust Mutex Priority Protection option.

The [EINVAL] error for an uninitialized mutex attributes object is removed; this condition results in undefined behavior.

---

## 1.160. pthread\_mutexattr\_getprotocol

### Synopsis

```
#include <pthread.h>

int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *restrict
                                   attr,
                                   int *restrict protocol);

int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
                                   int protocol);
```

### Description

The `pthread_mutexattr_getprotocol()` and `pthread_mutexattr_setprotocol()` functions, respectively, shall get and set the protocol attribute of a mutex attributes object pointed to by `attr` which was previously created by the function `pthread_mutexattr_init()`.

The `protocol` attribute defines the protocol to be followed in utilizing mutexes. The value of `protocol` may be one of:

- `PTHREAD_PRIO_INHERIT`
- `PTHREAD_PRIO_NONE`
- `PTHREAD_PRIO_PROTECT`

which are defined in the `<pthread.h>` header. The default value of the attribute shall be `PTHREAD_PRIO_NONE`.

### Protocol Behavior

When a thread owns a mutex with the `PTHREAD_PRIO_NONE` protocol attribute, its priority and scheduling shall not be affected by its mutex ownership.

#### `PTHREAD_PRIO_INHERIT`

When a thread is blocking higher priority threads because of owning one or more mutexes with the `PTHREAD_PRIO_INHERIT` protocol attribute, it shall execute at the higher of its priority or the priority of the highest priority thread waiting on any of the mutexes owned by this thread and initialized with this protocol.

When a thread makes a call to `pthread_mutex_lock()`, the mutex was initialized with the protocol attribute having the value `PTHREAD_PRIO_INHERIT`, when the calling thread is blocked because the mutex is owned by another thread, that owner thread shall inherit the priority level of the calling thread as long as it continues to own the mutex. The implementation shall update its execution priority to the maximum of its assigned priority and all its inherited priorities. Furthermore, if this owner thread itself becomes blocked on another mutex with the protocol attribute having the value `PTHREAD_PRIO_INHERIT`, the same priority inheritance effect shall be propagated to this other owner thread, in a recursive manner.

### `PTHREAD_PRIO_PROTECT`

When a thread owns one or more mutexes initialized with the `PTHREAD_PRIO_PROTECT` protocol, it shall execute at the higher of its priority or the highest of the priority ceilings of all the mutexes owned by this thread and initialized with this attribute, regardless of whether other threads are blocked on any of these mutexes or not.

## Multiple Mutexes

If a thread simultaneously owns several mutexes initialized with different protocols, it shall execute at the highest of the priorities that it would have obtained by each of these protocols.

## Return Value

Upon successful completion, the `pthread_mutexattr_getprotocol()` and `pthread_mutexattr_setprotocol()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

## Errors

The `pthread_mutexattr_setprotocol()` function shall fail if:

- **ENOTSUP** - The value specified by `protocol` is an unsupported value.

The `pthread_mutexattr_getprotocol()` and `pthread_mutexattr_setprotocol()` functions may fail if:

- **EINVAL** - The value specified by `attr` or `protocol` is invalid.

# Examples

No examples are provided in the specification.

## Application Usage

The `pthread_mutexattr_getprotocol()` and `pthread_mutexattr_setprotocol()` functions are part of the Threads option and either the Thread Priority Protection or Thread Priority Inheritance options.

These functions require support of either the Non-Robust Mutex Priority Protection option or the Non-Robust Mutex Priority Inheritance option or the Robust Mutex Priority Protection option or the Robust Mutex Priority Inheritance option.

## Rationale

The protocol attribute allows applications to control the priority behavior of mutexes to prevent priority inversion scenarios where lower-priority threads hold mutexes needed by higher-priority threads.

## Default Value

The default value of the protocol attribute is `PTHREAD_PRIO_NONE`, which means that mutex ownership does not affect thread priority or scheduling.

## Priority Inheritance

Priority inheritance (`PTHREAD_PRIO_INHERIT`) temporarily raises the priority of a thread that holds a mutex when higher-priority threads are waiting for that mutex. This helps prevent unbounded priority inversion.

## Priority Protection

Priority protection (`PTHREAD_PRIO_PROTECT`) sets a priority ceiling for each mutex. A thread that owns such a mutex executes at the higher of its own priority or the highest priority ceiling of all mutexes it owns that were initialized with this protocol.

## Future Directions

None.

## See Also

- [pthread\\_mutexattr\\_init\(\)](#)
- [pthread\\_mutex\\_lock\(\)](#)
- [<pthread.h>](#)

## Copyright

Portions of this text are reprinted and reproduced in electronic form from IEEE Std 1003.1-2017, The Open Group Base Specifications Issue 7, Copyright (C) 2017 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group. In the event of any discrepancy between this version and the original IEEE and The Open Group Standard, the original IEEE and The Open Group Standard shall prevail. The original Standard can be obtained online at <http://www.opengroup.org/unix/online.html>.

Any typographical or formatting errors that appear in this page are most likely to have been introduced during the conversion of the source files to man page format. To report such errors, see [https://www.kernel.org/doc/man-pages/reporting\\_bugs.html](https://www.kernel.org/doc/man-pages/reporting_bugs.html).

## 1.161. `pthread_mutexattr_gettype`, `pthread_mutexattr_settype` — get and set the mutex type attribute

### SYNOPSIS

```
#include <pthread.h>

int pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict attr,
                               int *restrict type);

int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type)
```

### DESCRIPTION

The `pthread_mutexattr_gettype()` and `pthread_mutexattr_settype()` functions, respectively, shall get and set the mutex **type** attribute. This attribute is set in the **type** parameter to these functions. The default value of the **type** attribute is `PTHREAD_MUTEX_DEFAULT`.

The type of mutex is contained in the **type** attribute of the mutex attributes. Valid mutex types include:

- `PTHREAD_MUTEX_NORMAL`
- `PTHREAD_MUTEX_ERRORCHECK`
- `PTHREAD_MUTEX_RECURSIVE`
- `PTHREAD_MUTEX_DEFAULT`

The mutex type affects the behavior of calls which lock and unlock the mutex. See `pthread_mutex_lock()` for details. An implementation may map `PTHREAD_MUTEX_DEFAULT` to one of the other mutex types.

The behavior is undefined if the value specified by the **attr** argument to `pthread_mutexattr_gettype()` or `pthread_mutexattr_settype()` does not refer to an initialized mutex attributes object.

## RETURN VALUE

Upon successful completion, the `pthread_mutexattr_gettype()` function shall return zero and store the value of the **type** attribute of **attr** into the object referenced by the **type** parameter. Otherwise, an error shall be returned to indicate the error.

If successful, the `pthread_mutexattr_settype()` function shall return zero; otherwise, an error number shall be returned to indicate the error.

## ERRORS

The `pthread_mutexattr_settype()` function shall fail if:

**[EINVAL]**

The value **type** is invalid.

These functions shall not return an error code of **[EINTR]**.

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

It is advised that an application should not use a `PTHREAD_MUTEX_RECURSIVE` mutex with condition variables because the implicit unlock performed in a `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, or `pthread_cond_wait()` call may not actually release the mutex (if it had been locked multiple times). If this happens, no other thread can satisfy the condition of the predicate.

## RATIONALE

If an implementation detects that the value specified by the **attr** argument to `pthread_mutexattr_gettype()` or `pthread_mutexattr_settype()` does not

refer to an initialized mutex attributes object, it is recommended that the function should fail and report an `[EINVAL]` error.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_cond_clockwait()`
- `pthread_mutex_lock()`
- `<pthread.h>`

## CHANGE HISTORY

First released in Issue 5.

### Issue 6

The Open Group Corrigendum U033/3 is applied. The SYNOPSIS for `pthread_mutexattr_gettype()` is updated so that the first argument is of type `const pthread_mutexattr_t *`.

The **restrict** keyword is added to the `pthread_mutexattr_gettype()` prototype for alignment with the ISO/IEC 9899:1999 standard.

### Issue 7

The `pthread_mutexattr_gettype()` and `pthread_mutexattr_settype()` functions are moved from the XSI option to the Base.

The `[EINVAL]` error for an uninitialized mutex attributes object is removed; this condition results in undefined behavior.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0464 [121] is applied.

### Issue 8

Austin Group Defect 1216 is applied, adding `pthread_cond_clockwait()`.

---

## 1.162. `pthread_mutexattr_init`, `pthread_mutexattr_destroy` - destroy and initialize a mutex attributes object

---

### SYNOPSIS

```
#include <pthread.h>

int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

### DESCRIPTION

The `pthread_mutexattr_destroy()` function shall destroy a mutex attributes object; the object becomes, in effect, uninitialized. An implementation may cause `pthread_mutexattr_destroy()` to set the object referenced by `attr` to an invalid value.

A destroyed `attr` attributes object can be reinitialized using `pthread_mutexattr_init()`; the results of otherwise referencing the object after it has been destroyed are undefined.

The `pthread_mutexattr_init()` function shall initialize a mutex attributes object `attr` with the default value for all of the attributes defined by the implementation.

Results are undefined if `pthread_mutexattr_init()` is called specifying an already initialized `attr` attributes object.

After a mutex attributes object has been used to initialize one or more mutexes, any function affecting the attributes object (including destruction) shall not affect any previously initialized mutexes.

The behavior is undefined if the value specified by the `attr` argument to `pthread_mutexattr_destroy()` does not refer to an initialized mutex attributes object.

### RETURN VALUE

Upon successful completion, these functions shall return zero; otherwise, an error number shall be returned to indicate the error.

# ERRORS

These functions may fail if:

- **EINVAL** - The value specified by `attr` does not refer to an initialized mutex attributes object.

These functions shall not return an error code of **EINTR**.

## EXAMPLES

None.

## APPLICATION USAGE

If an implementation detects that the value specified by the `attr` argument to `pthread_mutexattr_destroy()` does not refer to an initialized mutex attributes object, it is recommended that the function should fail and report an **[EINVAL]** error.

See `pthread_attr_destroy()` for a general explanation of attributes. Attributes objects allow implementations to experiment with useful extensions and permit extension of this volume of POSIX.1-2024 without changing the existing functions. Thus, they provide for future extensibility of this volume of POSIX.1-2024 and reduce the temptation to standardize prematurely on semantics that are not yet widely implemented or understood.

Examples of possible additional mutex attributes that have been discussed are `spin_only`, `limited_spin`, `no_spin`, `recursive`, and `metered`. (To explain what the latter attributes might mean: recursive mutexes would allow for multiple re-locking by the current owner; metered mutexes would transparently keep records of queue length, wait time, and so on.) Since there is not yet wide agreement on the usefulness of these resulting from shared implementation and usage experience, they are not yet specified in this volume of POSIX.1-2024. Mutex attributes objects, however, make it possible to test out these concepts for possible standardization at a later time.

## Mutex Attributes and Performance

Care has been taken to ensure that the default values of the mutex attributes have been defined such that mutexes initialized with the defaults have simple enough semantics so that the locking and unlocking can be done with the equivalent of a test-and-set instruction (plus possibly a few other basic instructions).

There is at least one implementation method that can be used to reduce the cost of testing at lock-time if a mutex has non-default attributes. One such method that an implementation can employ (and this can be made fully transparent to fully conforming POSIX applications) is to secretly pre-lock any mutexes that are initialized to non-default attributes. Any later attempt to lock such a mutex causes the implementation to branch to the "slow path" as if the mutex were unavailable; then, on the slow path, the implementation can do the "real work" to lock a non-default mutex. The underlying unlock operation is more complicated since the implementation never really wants to release the pre-lock on this kind of mutex. This illustrates that, depending on the hardware, there may be certain optimizations that can be used so that whatever mutex attributes are considered "most frequently used" can be processed most efficiently.

## Process Shared Memory and Synchronization

The existence of memory mapping functions in this volume of POSIX.1-2024 leads to the possibility that an application may allocate the synchronization objects from this section in memory that is accessed by multiple processes (and therefore, by threads of multiple processes).

In order to permit such usage, while at the same time keeping the usual case (that is, usage within a single process) efficient, a **process-shared** option has been defined.

If an implementation supports the `_POSIX_THREAD_PROCESS_SHARED` option, then the **process-shared** attribute can be used to indicate that mutexes or condition variables may be accessed by threads of multiple processes.

The default setting of `PTHREAD_PROCESS_PRIVATE` has been chosen for the **process-shared** attribute so that the most efficient forms of these synchronization objects are created by default.

Synchronization variables that are initialized with the `PTHREAD_PROCESS_PRIVATE` **process-shared** attribute may only be operated on by threads in the process that initialized them. Synchronization variables that are initialized with the `PTHREAD_PROCESS_SHARED` **process-shared** attribute may be operated on by any thread in any process that has access to it. In particular, these processes may exist beyond the lifetime of the initializing process. For example, the following code implements a simple counting semaphore in a mapped file that may be used by many processes.

```
/* sem.h */
struct semaphore {
    pthread_mutex_t lock;
    pthread_cond_t nonzero;
    unsigned count;
```

```

};

typedef struct semaphore semaphore_t;

semaphore_t *semaphore_create(char *semaphore_name);
semaphore_t *semaphore_open(char *semaphore_name);
void semaphore_post(semaphore_t *semaphore);
void semaphore_wait(semaphore_t *semaphore);
void semaphore_close(semaphore_t *semaphore);

/* sem.c */
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <pthread.h>
#include "sem.h"

semaphore_t *
semaphore_create(char *semaphore_name)
{
    int fd;
    semaphore_t *semaphore;
    pthread_mutexattr_t psharedm;
    pthread_condattr_t psharedc;

    fd = open(semaphore_name, O_RDWR | O_CREAT | O_EXCL, 0666);
    if (fd < 0)
        return (NULL);
    (void) ftruncate(fd, sizeof(semaphore_t));
    (void) pthread_mutexattr_init(&psharedm);
    (void) pthread_mutexattr_setpshared(&psharedm,
        PTHREAD_PROCESS_SHARED);
    (void) pthread_condattr_init(&psharedc);
    (void) pthread_condattr_setpshared(&psharedc,
        PTHREAD_PROCESS_SHARED);
    semaphore = (semaphore_t *) mmap(NULL, sizeof(semaphore_t),
        PROT_READ | PROT_WRITE, MAP_SHARED,
        fd, 0);
    close (fd);
    (void) pthread_mutex_init(&semaphore->lock, &psharedm);
    (void) pthread_cond_init(&semaphore->nonzero, &psharedc);
    semaphore->count = 0;
    return (semaphore);
}

semaphore_t *
semaphore_open(char *semaphore_name)
{
    int fd;
    semaphore_t *semaphore;

```

```

fd = open(semaphore_name, O_RDWR, 0666);
if (fd < 0)
    return (NULL);
semap = (semaphore_t *) mmap(NULL, sizeof(semaphore_t),
    PROT_READ | PROT_WRITE, MAP_SHARED,
    fd, 0);
close (fd);
return (semaphore_t);
}

void
semaphore_post(semaphore_t *semaphore)
{
    pthread_mutex_lock(&semaphore->lock);
    if (semaphore->count == 0)
        pthread_cond_signal(&semaphore->nonzero);
    semaphore->count++;
    pthread_mutex_unlock(&semaphore->lock);
}

void
semaphore_wait(semaphore_t *semaphore)
{
    pthread_mutex_lock(&semaphore->lock);
    while (semaphore->count == 0)
        pthread_cond_wait(&semaphore->nonzero, &semaphore->lock);
    semaphore->count--;
    pthread_mutex_unlock(&semaphore->lock);
}

void
semaphore_close(semaphore_t *semaphore)
{
    munmap((void *) semaphore, sizeof(semaphore_t));
}

```

The following code is for three separate processes that create, post, and wait on a semaphore in the file **/tmp/semaphore**. Once the file is created, the post and wait programs increment and decrement the counting semaphore (waiting and waking as required) even though they did not initialize the semaphore.

```

/* create.c */
#include "pthread.h"
#include "sem.h"

int
main(void)
{
    semaphore_t *semaphore;

```

```

    semap = semaphore_create("/tmp/semaphore");
    if (semap == NULL)
        exit(1);
    semaphore_close(semap);
    return (0);
}

/* post.c */
#include "pthread.h"
#include "sem.h"

int
main(void)
{
    semaphore_t *semap;

    semap = semaphore_open("/tmp/semaphore");
    if (semap == NULL)
        exit(1);
    semaphore_post(semap);
    semaphore_close(semap);
    return (0);
}

/* wait.c */
#include "pthread.h"
#include "sem.h"

int
main(void)
{
    semaphore_t *semap;

    semap = semaphore_open("/tmp/semaphore");
    if (semap == NULL)
        exit(1);
    semaphore_wait(semap);
    semaphore_close(semap);
    return (0);
}

```

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [pthread\\_attr\\_destroy\(\)](#)
- [pthread\\_mutex\\_init\(\)](#)
- [pthread\\_mutexattr\\_getpshared\(\)](#)
- [pthread\\_mutexattr\\_setpshared\(\)](#)

The Base Definitions volume of POSIX.1-2024, [\<pthread.h>](#)

## COPYRIGHT

Portions of this text are reprinted and reproduced in electronic form from the IEEE Std 1003.1-2024, Standard for Information Technology -- Portable Operating System Interface (POSIX), The Open Group Base Specifications Issue 7, 2024 Edition (IEEE Std 1003.1-2024). Copyright © 2024 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group. This copyrighted material is made available under the terms of the BSD License, which is available at <https://opensource.org/licenses/BSD-3-Clause>.

---

## 1.163. pthread\_mutexattr\_getprioceiling, pthread\_mutexattr\_setprioceiling

### SYNOPSIS

```
#include <pthread.h>

int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t *restrict attr,
                                     int *restrict prioceiling);

int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
                                     int prioceiling);
```

### DESCRIPTION

The `pthread_mutexattr_getprioceiling()` and `pthread_mutexattr_setprioceiling()` functions, respectively, shall get and set the priority ceiling attribute of a mutex attributes object pointed to by `attr` which was previously created by the function `pthread_mutexattr_init()`.

The `prioceiling` attribute contains the priority ceiling of initialized mutexes. The values of `prioceiling` are within the maximum range of priorities defined by SCHED\_FIFO.

The `prioceiling` attribute defines the priority ceiling of initialized mutexes, which is the minimum priority level at which the critical section guarded by the mutex is executed. In order to avoid priority inversion, the priority ceiling of the mutex shall be set to a priority higher than or equal to the highest priority of all the threads that may lock that mutex. The values of `prioceiling` are within the maximum range of priorities defined under the SCHED\_FIFO scheduling policy.

The behavior is undefined if the value specified by the `attr` argument to `pthread_mutexattr_getprioceiling()` or `pthread_mutexattr_setprioceiling()` does not refer to an initialized mutex attributes object.

### RETURN VALUE

Upon successful completion, the `pthread_mutexattr_getprioceiling()` and `pthread_mutexattr_setprioceiling()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

# ERRORS

These functions may fail if:

- **[EINVAL]**

The value specified by `prioceiling` is invalid.

- **[EPERM]**

The caller does not have the privilege to perform the operation.

These functions shall not return an error code of [EINTR].

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

If an implementation detects that the value specified by the `attr` argument to `pthread_mutexattr_getprioceiling()` or `pthread_mutexattr_setprioceiling()` does not refer to an initialized mutex attributes object, it is recommended that the function should fail and report an [EINVAL] error.

## FUTURE DIRECTIONS

None.

## SEE ALSO

`pthread_cond_destroy()`,  
`pthread_mutex_destroy()`

`pthread_create()`,

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Marked as part of the Realtime Threads Feature Group.

### Issue 6

The `pthread_mutexattr_getprioceiling()` and `pthread_mutexattr_setprioceiling()` functions are marked as part of the Threads and Thread Priority Protection options.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Thread Priority Protection option.

The [ENOTSUP] error condition has been removed since these functions do not have a `protocol` argument.

The `restrict` keyword is added to the `pthread_mutexattr_getprioceiling()` prototype for alignment with the ISO/IEC 9899:1999 standard.

### Issue 7

The `pthread_mutexattr_getprioceiling()` and `pthread_mutexattr_setprioceiling()` functions are moved from the Threads option to require support of either the Robust Mutex Priority Protection option or the Non-Robust Mutex Priority Protection option.

The [EINVAL] error for an uninitialized mutex attributes object is removed; this condition results in undefined behavior.

## 1.164. `pthread_mutexattr_setprotocol`, `pthread_mutexattr_getprotocol`

获取和设置互斥锁属性对象的协议属性。

### 函数概要

```
#include <pthread.h>

int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int p
int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *restrict
```

### 参数

- `attr` - 指向先前由 `pthread_mutexattr_init()` 函数创建的互斥锁属性对象的指针
- `protocol` - 协议属性的值 (对于设置函数)
- `protocol` - 指向存储协议属性的整数的指针 (对于获取函数)

### 返回值

成功时，函数返回 0。失败时，返回错误编码以指示错误。

### 错误

- `ENOSYS` - 实现不支持指定的协议值
- `ENOTSUP` - 实现不支持指定的协议值
- `EINVAL` - `protocol` 的值无效

### 描述

`pthread_mutexattr_getprotocol()` 和 `pthread_mutexattr_setprotocol()` 函数分别获取和设置由 `attr` 指向的互斥锁属性对象的协议属性，该对象先前由 `pthread_mutexattr_init()` 函数创建。

协议属性定义了使用互斥锁时要遵循的协议。`protocol` 的值可以是以下之一：

- `PTHREAD_PRIO_INHERIT` - 优先级继承协议
- `PTHREAD_PRIO_NONE` - 无特殊协议 (默认值)
- `PTHREAD_PRIO_PROTECT` - 优先级保护协议

这些值在 `<pthread.h>` 头文件中定义。属性的默认值应为 `PTHREAD_PRIO_NONE`。

## 协议行为详解

### `PTHREAD_PRIO_NONE`

当线程拥有具有 `PTHREAD_PRIO_NONE` 协议属性的互斥锁时，其优先级和调度不应受到其互斥锁所有权的影响。

### `PTHREAD_PRIO_INHERIT`

当线程由于拥有一个或多个使用 `PTHREAD_PRIO_INHERIT` 协议属性初始化的互斥锁而阻塞更高优先级的线程时，它应以其优先级或等待该线程拥有的、使用此协议初始化的任何互斥锁的最高优先级线程的优先级中较高的那个优先级执行。

当线程调用 `pthread_mutex_lock()` 时，互斥锁使用 `PTHREAD_PRIO_INHERIT` 协议属性值初始化，当调用线程因互斥锁被另一个线程拥有而被阻塞时，该拥有者线程应继承调用线程的优先级级别，只要它继续拥有该互斥锁。实现应将其执行优先级更新为其分配的优先级和所有继承优先级的最大值。此外，如果此拥有者线程本身在另一个具有 `PTHREAD_PRIO_INHERIT` 协议属性值的互斥锁上被阻塞，相同的优先级继承效果应以递归方式传播到该其他拥有者线程。

### `PTHREAD_PRIO_PROTECT`

当线程拥有一个或多个使用 `PTHREAD_PRIO_PROTECT` 协议初始化的互斥锁时，它应以其优先级或该线程拥有的、使用此属性初始化的所有互斥锁的最高优先级上限中较高的那个优先级执行，无论是否有其他线程被这些互斥锁阻塞。

## 多协议场景

如果线程同时拥有几个使用不同协议初始化的互斥锁，它应以其通过这些协议各自获得的优先级中最高的优先级执行。

## 注意事项

如果 `pthread_mutexattr_getprotocol()` 或 `pthread_mutexattr_setprotocol()` 的 `attr` 参数指定的值不引用已初始化的互斥锁属性对象，则行为未定义。

## 一致性

这些函数属于 POSIX.1-2008 标准。

这些函数是线程选项以及线程优先级保护或线程优先级继承选项的一部分。

## 参见

- [pthread\\_mutexattr\\_init\(\)](#)
- [pthread\\_mutex\\_lock\(\)](#)
- [pthread\\_mutex\\_init\(\)](#)
- [<pthread.h>](#)

## 版权

© IEEE 2003, 2023 - All rights reserved.

---

## 1.165. `pthread_mutexattr_gettype`, `pthread_mutexattr_settype` - get and set the mutex type attribute

### SYNOPSIS

```
#include <pthread.h>

int pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict attr,
                               int *restrict type);

int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type)
```

### DESCRIPTION

The `pthread_mutexattr_gettype()` and `pthread_mutexattr_settype()` functions, respectively, shall get and set the mutex type attribute. This attribute is set in the type parameter to these functions. The default value of the type attribute is `PTHREAD_MUTEX_DEFAULT`.

The type of mutex is contained in the type attribute of the mutex attributes. Valid mutex types include:

- `PTHREAD_MUTEX_NORMAL`
- `PTHREAD_MUTEX_ERRORCHECK`
- `PTHREAD_MUTEX_RECURSIVE`
- `PTHREAD_MUTEX_DEFAULT`

The mutex type affects the behavior of calls which lock and unlock the mutex. See `pthread_mutex_lock()` for details. An implementation may map `PTHREAD_MUTEX_DEFAULT` to one of the other mutex types.

The behavior is undefined if the value specified by the attr argument to `pthread_mutexattr_gettype()` or `pthread_mutexattr_settype()` does not refer to an initialized mutex attributes object.

## RETURN VALUE

Upon successful completion, the `pthread_mutexattr_gettype()` function shall return zero and store the value of the type attribute of attr into the object referenced by the type parameter. Otherwise, an error shall be returned to indicate the error.

If successful, the `pthread_mutexattr_settype()` function shall return zero; otherwise, an error number shall be returned to indicate the error.

## ERRORS

The `pthread_mutexattr_settype()` function shall fail if:

- **[EINVAL]**

The value type is invalid.

These functions shall not return an error code of [EINTR].

## EXAMPLES

None.

## APPLICATION USAGE

It is advised that an application should not use a PTHREAD\_MUTEX\_RECURSIVE mutex with condition variables because the implicit unlock performed in a `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, or `pthread_cond_wait()` call may not actually release the mutex (if it had been locked multiple times). If this happens, no other thread can satisfy the condition of the predicate.

## RATIONALE

If an implementation detects that the value specified by the attr argument to `pthread_mutexattr_gettype()` or `pthread_mutexattr_settype()` does not refer to an initialized mutex attributes object, it is recommended that the function should fail and report an [EINVAL] error.

# FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_cond_clockwait()`
- `pthread_mutex_lock()`
- `<pthread.h>`

## CHANGE HISTORY

First released in Issue 5.

### Issue 6

The Open Group Corrigendum U033/3 is applied. The SYNOPSIS for `pthread_mutexattr_gettype()` is updated so that the first argument is of type `const pthread_mutexattr_t *`.

The `restrict` keyword is added to the `pthread_mutexattr_gettype()` prototype for alignment with the ISO/IEC 9899:1999 standard.

### Issue 7

The `pthread_mutexattr_gettype()` and `pthread_mutexattr_settype()` functions are moved from the XSI option to the Base.

The [EINVAL] error for an uninitialized mutex attributes object is removed; this condition results in undefined behavior.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0464 [121] is applied.

### Issue 8

Austin Group Defect 1216 is applied, adding `pthread_cond_clockwait()`.

---

## 1.166. `pthread_once` — dynamic package initialization

### SYNOPSIS

```
#include <pthread.h>

int pthread_once(pthread_once_t *once_control,
                 void (*init_routine)(void));

pthread_once_t once_control = PTHREAD_ONCE_INIT;
```

### DESCRIPTION

The first call to `pthread_once()` by any thread in a process, with a given `once_control`, shall call the `init_routine` with no arguments. Subsequent calls of `pthread_once()` with the same `once_control` shall not call the `init_routine`. On return from `pthread_once()`, `init_routine` shall have completed. The `once_control` parameter shall determine whether the associated initialization routine has been called.

The `pthread_once()` function is not a cancellation point. However, if `init_routine` is a cancellation point and is canceled, the effect on `once_control` shall be as if `pthread_once()` was never called.

If the call to `init_routine` is terminated by a call to `longjmp()` or `siglongjmp()`, the behavior is undefined.

The constant `PTHREAD_ONCE_INIT` is defined in the `<pthread.h>` header.

The behavior of `pthread_once()` is undefined if `once_control` has automatic storage duration or is not initialized by `PTHREAD_ONCE_INIT`.

### RETURN VALUE

Upon successful completion, `pthread_once()` shall return zero; otherwise, an error number shall be returned to indicate the error.

### ERRORS

The `pthread_once()` function shall not return an error code of [EINTR].

## APPLICATION USAGE

If `init_routine` recursively calls `pthread_once()` with the same `once_control`, the recursive call will not call the specified `init_routine`, and thus the specified `init_routine` will not complete, and thus the recursive call to `pthread_once()` will not return. Use of `longjmp()` or `siglongjmp()` within an `init_routine` to jump to a point outside of `init_routine` prevents `init_routine` from returning.

## RATIONALE

Some C libraries are designed for dynamic initialization. That is, the global initialization for the library is performed when the first procedure in the library is called. In a single-threaded program, this is normally implemented using a static variable whose value is checked on entry to a routine, as follows:

```
static int random_is_initialized = 0;
extern void initialize_random(void);

int random_function()
{
    if (random_is_initialized == 0) {
        initialize_random();
        random_is_initialized = 1;
    }
    ... /* Operations performed after initialization. */
}
```

To keep the same structure in a multi-threaded program, a new primitive is needed. Otherwise, library initialization has to be accomplished by an explicit call to a library-exported initialization function prior to any use of the library.

For dynamic library initialization in a multi-threaded process, if an initialization flag is used the flag needs to be protected against modification by multiple threads simultaneously calling into the library. This can be done by using a mutex (initialized by assigning `PTHREAD_MUTEX_INITIALIZER`). However, the better solution is to use `pthread_once()` which is designed for exactly this purpose, as follows:

```
#include <pthread.h>
static pthread_once_t random_is_initialized = PTHREAD_ONCE_INIT;
extern void initialize_random(void);

int random_function()
{
```

```
        (void) pthread_once(&random_is_initialized, initialize_random)
        ... /* Operations performed after initialization. */
    }
```

If an implementation detects that the value specified by the `once_control` argument to `pthread_once()` does not refer to a `pthread_once_t` object initialized by `PTHREAD_ONCE_INIT`, it is recommended that the function should fail and report an [EINVAL] error.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `<pthread.h>`

## CHANGE HISTORY

**First released in Issue 5.** Included for alignment with the POSIX Threads Extension.

### Issue 6

- The `pthread_once()` function is marked as part of the Threads option.
- The [EINVAL] error is added as a "may fail" case for if either argument is invalid.

### Issue 7

- The `pthread_once()` function is moved from the Threads option to the Base.
- The [EINVAL] error for an uninitialized `pthread_once_t` object is removed; this condition results in undefined behavior.
- POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0284 [863], XSH/TC2-2008/0285 [874], XSH/TC2-2008/0286 [874], and XSH/TC2-2008/0287 [747] are applied.

### Issue 8

- Austin Group Defect 1330 is applied, removing obsolescent interfaces.

## 1.167. `pthread_self` — get the calling thread ID

---

### SYNOPSIS

```
#include <pthread.h>

pthread_t pthread_self(void);
```

### DESCRIPTION

The `pthread_self()` function shall return the thread ID of the calling thread.

### RETURN VALUE

The `pthread_self()` function shall always be successful and no return value is reserved to indicate an error.

### ERRORS

No errors are defined.

*The following sections are informative.*

### EXAMPLES

None.

### APPLICATION USAGE

None.

### RATIONALE

The `pthread_self()` function provides a capability similar to the `getpid()` function for processes and the rationale is the same: the creation call does not provide the thread ID to the created thread.

# FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_create()`
- `pthread_equal()`
- `<pthread.h>`

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

### Issue 6

The `pthread_self()` function is marked as part of the Threads option.

### Issue 7

Austin Group Interpretation 1003.1-2001 #063 is applied, updating the RETURN VALUE section.

The `pthread_self()` function is moved from the Threads option to the Base.

## 1.168. `pthread_setcancelstate`, `pthread_setcanceltype`, `pthread_testcancel`

---

### SYNOPSIS

```
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
void pthread_testcancel(void);
```

### DESCRIPTION

The `pthread_setcancelstate()` function shall atomically both set the calling thread's cancelability state to the indicated `state` and return the previous cancelability state at the location referenced by `oldstate`. Legal values for `state` are `PTHREAD_CANCEL_ENABLE` and `PTHREAD_CANCEL_DISABLE`.

The `pthread_setcanceltype()` function shall atomically both set the calling thread's cancelability type to the indicated `type` and return the previous cancelability type at the location referenced by `oldtype`. Legal values for `type` are `PTHREAD_CANCEL_DEFERRED` and `PTHREAD_CANCEL_ASYNCHRONOUS`.

The cancelability state and type of any newly created threads, including the thread in which `main()` was first invoked, shall be `PTHREAD_CANCEL_ENABLE` and `PTHREAD_CANCEL_DEFERRED` respectively.

The `pthread_testcancel()` function shall create a cancellation point in the calling thread. The `pthread_testcancel()` function shall have no effect if cancelability is disabled.

The `pthread_setcancelstate()` function shall be async-signal-safe.

### RETURN VALUE

If successful, the `pthread_setcancelstate()` and `pthread_setcanceltype()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

# ERRORS

The `pthread_setcancelstate()` function may fail if:

- **[EINVAL]**
- The specified state is not `PTHREAD_CANCEL_ENABLE` or `PTHREAD_CANCEL_DISABLE`.

The `pthread_setcanceltype()` function may fail if:

- **[EINVAL]**
- The specified type is not `PTHREAD_CANCEL_DEFERRED` or `PTHREAD_CANCEL_ASYNCHRONOUS`.

These functions shall not return an error code of `[EINTR]`.

# EXAMPLES

None.

# APPLICATION USAGE

In order to write a signal handler for an asynchronous signal which can run safely in a cancellable thread, `pthread_setcancelstate()` must be used to disable cancellation for the duration of any calls that the signal handler makes which are cancellation points. However, earlier versions of the standard did not permit strictly conforming applications to call `pthread_setcancelstate()` from a signal handler since it was not required to be async-signal-safe. On non-conforming implementations where `pthread_setcancelstate()` is not async-signal-safe, alternatives are to ensure either that the corresponding signals are blocked during execution of functions that are not async-cancel-safe or that cancellation is disabled during times when those signals could be delivered.

# RATIONALE

The `pthread_setcancelstate()` and `pthread_setcanceltype()` functions control the points at which a thread may be asynchronously canceled. For cancellation control to be usable in modular fashion, some rules need to be followed.

An object can be considered to be a generalization of a procedure. It is a set of procedures and global variables written as a unit and called by clients not known

by the object. Objects may depend on other objects.

First, cancelability should only be disabled on entry to an object, never explicitly enabled. On exit from an object, the cancelability state should always be restored to its value on entry to the object.

This follows from a modularity argument: if the client of an object (or the client of an object that uses that object) has disabled cancelability, it is because the client does not want to be concerned about cleaning up if the thread is canceled while executing some sequence of actions. If an object is called in such a state and it enables cancelability and a cancellation request is pending for that thread, then the thread is canceled, contrary to the wish of the client that disabled.

Second, the cancelability type may be explicitly set to either *deferred* or *asynchronous* upon entry to an object. But as with the cancelability state, on exit from an object the cancelability type should always be restored to its value on entry to the object.

Finally, only functions that are cancel-safe may be called from a thread that is asynchronously cancelable.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_cancel()`
- XBD `<pthread.h>`

## CHANGE HISTORY

### First released in Issue 5

Included for alignment with the POSIX Threads Extension.

### Issue 6

The `pthread_setcancelstate()`, `pthread_setcanceltype()`, and `pthread_testcancel()` functions are marked as part of the Threads option.

## Issue 7

The `pthread_setcancelstate()`, `pthread_setcanceltype()`, and `pthread_testcancel()` functions are moved from the Threads option to the Base.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0294 [622] and XSH/TC2-2008/0295 [615] are applied.

## Issue 8

Austin Group Defect 841 is applied, requiring `pthread_setcancelstate()` to be async-signal-safe.

---

## 1.169. `pthread_setcancelstate`, `pthread_setcanceltype`, `pthread_testcancel` — set cancelability state

---

### SYNOPSIS

```
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
void pthread_testcancel(void);
```

### DESCRIPTION

The `pthread_setcancelstate()` function shall atomically both set the calling thread's cancelability state to the indicated `state` and return the previous cancelability state at the location referenced by `oldstate`. Legal values for `state` are `PTHREAD_CANCEL_ENABLE` and `PTHREAD_CANCEL_DISABLE`.

The `pthread_setcanceltype()` function shall atomically both set the calling thread's cancelability type to the indicated `type` and return the previous cancelability type at the location referenced by `oldtype`. Legal values for `type` are `PTHREAD_CANCEL_DEFERRED` and `PTHREAD_CANCEL_ASYNCHRONOUS`.

The cancelability state and type of any newly created threads, including the thread in which `main()` was first invoked, shall be `PTHREAD_CANCEL_ENABLE` and `PTHREAD_CANCEL_DEFERRED` respectively.

The `pthread_testcancel()` function shall create a cancellation point in the calling thread. The `pthread_testcancel()` function shall have no effect if cancelability is disabled.

The `pthread_setcancelstate()` function shall be async-signal-safe.

### RETURN VALUE

If successful, the `pthread_setcancelstate()` and `pthread_setcanceltype()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

# ERRORS

The `pthread_setcancelstate()` function may fail if:

- **[EINVAL]**
- The specified state is not `PTHREAD_CANCEL_ENABLE` or `PTHREAD_CANCEL_DISABLE`.

The `pthread_setcanceltype()` function may fail if:

- **[EINVAL]**
- The specified type is not `PTHREAD_CANCEL_DEFERRED` or `PTHREAD_CANCEL_ASYNCHRONOUS`.

These functions shall not return an error code of [EINTR].

## EXAMPLES

None.

## APPLICATION USAGE

In order to write a signal handler for an asynchronous signal which can run safely in a cancellable thread, `pthread_setcancelstate()` must be used to disable cancellation for the duration of any calls that the signal handler makes which are cancellation points. However, earlier versions of the standard did not permit strictly conforming applications to call `pthread_setcancelstate()` from a signal handler since it was not required to be async-signal-safe. On non-conforming implementations where `pthread_setcancelstate()` is not async-signal-safe, alternatives are to ensure either that the corresponding signals are blocked during execution of functions that are not async-cancel-safe or that cancellation is disabled during times when those signals could be delivered.

## RATIONALE

The `pthread_setcancelstate()` and `pthread_setcanceltype()` functions control the points at which a thread may be asynchronously canceled. For cancellation control to be usable in modular fashion, some rules need to be followed.

An object can be considered to be a generalization of a procedure. It is a set of procedures and global variables written as a unit and called by clients not known

by the object. Objects may depend on other objects.

First, cancelability should only be disabled on entry to an object, never explicitly enabled. On exit from an object, the cancelability state should always be restored to its value on entry to the object.

This follows from a modularity argument: if the client of an object (or the client of an object that uses that object) has disabled cancelability, it is because the client does not want to be concerned about cleaning up if the thread is canceled while executing some sequence of actions. If an object is called in such a state and it enables cancelability and a cancellation request is pending for that thread, then the thread is canceled, contrary to the wish of the client that disabled.

Second, the cancelability type may be explicitly set to either deferred or asynchronous upon entry to an object. But as with the cancelability state, on exit from an object the cancelability type should always be restored to its value on entry to the object.

Finally, only functions that are cancel-safe may be called from a thread that is asynchronously cancelable.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_cancel()`
- XBD `<pthread.h>`

## CHANGE HISTORY

### First released in Issue 5

Included for alignment with the POSIX Threads Extension.

### Issue 6

The `pthread_setcancelstate()`, `pthread_setcanceltype()`, and `pthread_testcancel()` functions are marked as part of the Threads option.

## Issue 7

The `pthread_setcancelstate()`, `pthread_setcanceltype()`, and `pthread_testcancel()` functions are moved from the Threads option to the Base.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0294 [622] and XSH/TC2-2008/0295 [615] are applied.

## Issue 8

Austin Group Defect 841 is applied, requiring `pthread_setcancelstate()` to be async-signal-safe.

---

## 1.170. `pthread_setconcurrency`, `pthread_getconcurrency`

---

### SYNOPSIS

```
#include <pthread.h>

int pthread_setconcurrency(int new_level);
int pthread_getconcurrency(void);
```

### DESCRIPTION

Unbound threads in a process may or may not be required to be simultaneously active. By default, the threads implementation ensures that a sufficient number of threads are active so that the process can continue to make progress. While this conserves system resources, it may not produce the most effective level of concurrency.

The `pthread_setconcurrency()` function allows an application to inform the threads implementation of its desired concurrency level, `new_level`. The actual level of concurrency provided by the implementation as a result of this function call is unspecified.

If `new_level` is zero, it causes the implementation to maintain the concurrency level at its discretion as if `pthread_setconcurrency()` had never been called.

The `pthread_getconcurrency()` function shall return the value set by a previous call to the `pthread_setconcurrency()` function. If the `pthread_setconcurrency()` function was not previously called, this function shall return zero to indicate that the implementation is maintaining the concurrency level.

A call to `pthread_setconcurrency()` shall inform the implementation of its desired concurrency level. The implementation shall use this as a hint, not a requirement.

If an implementation does not support multiplexing of user threads on top of several kernel-scheduled entities, the `pthread_setconcurrency()` and `pthread_getconcurrency()` functions are provided for source code compatibility but they shall have no effect when called. To maintain the function semantics, the `new_level` parameter is saved when `pthread_setconcurrency()` is called so that a subsequent call to `pthread_getconcurrency()` shall return the same value.

## RETURN VALUE

If successful, the `pthread_setconcurrency()` function shall return zero; otherwise, an error number shall be returned to indicate the error.

The `pthread_getconcurrency()` function shall always return the concurrency level set by a previous call to `pthread_setconcurrency()`. If the `pthread_setconcurrency()` function has never been called, `pthread_getconcurrency()` shall return zero.

## ERRORS

No errors are defined for these functions.

## EXAMPLES

None provided.

## APPLICATION USAGE

These functions allow applications to provide hints to the threads implementation about the desired level of concurrency. The implementation is not required to honor these hints, and the actual concurrency level may differ from the requested level.

## RATIONALE

The concurrency level functions provide a mechanism for applications to influence thread scheduling behavior when they have knowledge about the optimal number of simultaneously active threads. However, implementations are given flexibility in how they respond to these hints to accommodate different threading models and system architectures.

## FUTURE DIRECTIONS

None.

## SEE ALSO

None.

## COPYRIGHT

Portions of this text are reprinted and reproduced in electronic form from the IEEE Std 1003.1, 2013 Edition, Standard for Information Technology -- Portable Operating System Interface (POSIX), The Open Group Base Specifications Issue 7, Copyright (C) 2013 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group. (This is POSIX.1-2008 with the 2013 Technical Corrigendum 1 applied.) In the event of any discrepancy between this version and the original IEEE and The Open Group Standard, the original IEEE and The Open Group Standard shall be the referee. The original Standard can be obtained online at <http://www.opengroup.org/unix/online.html>.

*The following sections are informative.*

## 1.171. pthread\_getschedparam, pthread\_setschedparam

---

### SYNOPSIS

```
#include <pthread.h>

int pthread_getschedparam(pthread_t thread,
                          int *restrict policy,
                          struct sched_param *restrict param);

int pthread_setschedparam(pthread_t thread,
                          int policy,
                          const struct sched_param *param);
```

### DESCRIPTION

The `pthread_getschedparam()` and `pthread_setschedparam()` functions shall, respectively, get and set the scheduling policy and parameters of individual threads within a multi-threaded process to be retrieved and set. For SCHED\_FIFO and SCHED\_RR, the only required member of the `struct sched_param` structure is the priority `sched_priority`. For SCHED\_OTHER, the affected scheduling parameters are implementation-defined.

The `pthread_getschedparam()` function shall retrieve the scheduling policy and scheduling parameters for the thread whose thread ID is given by `thread` and shall store those values in `policy` and `param`, respectively. The priority value returned from `pthread_getschedparam()` shall be the value specified by the most recent `pthread_setschedparam()`, `pthread_setschedprio()`, or `pthread_create()` call affecting the target thread. It shall not reflect any temporary adjustments to its priority as a result of any priority inheritance or ceiling functions.

The `pthread_setschedparam()` function shall set the scheduling policy and associated scheduling parameters for the thread whose thread ID is given by `thread` to the policy and associated parameters provided in `policy` and `param`, respectively.

The `policy` parameter may have the value SCHED\_OTHER, SCHED\_FIFO, or SCHED\_RR. The scheduling parameters for the SCHED\_OTHER policy are implementation-defined. The SCHED\_FIFO and SCHED\_RR policies shall have a single scheduling parameter, `priority`.

## SCHED\_SPORADIC Server

If `_POSIX_THREAD_SPORADIC_SERVER` is defined, then the `policy` argument may have the value `SCHED_SPORADIC`, with the exception for the `pthread_setschedparam()` function that if the scheduling policy was not `SCHED_SPORADIC` at the time of the call, it is implementation-defined whether the function is supported; in other words, the implementation need not allow the application to dynamically change the scheduling policy to `SCHED_SPORADIC`.

The sporadic server scheduling policy has the associated parameters:

- `sched_ss_low_priority`
- `sched_ss_repl_period`
- `sched_ss_init_budget`
- `sched_priority`
- `sched_ss_max_repl`

The specified `sched_ss_repl_period` shall be greater than or equal to the specified `sched_ss_init_budget` for the function to succeed; if it is not, then the function shall fail. The value of `sched_ss_max_repl` shall be within the inclusive range `[1,{SS_REPL_MAX}]` for the function to succeed; if not, the function shall fail. It is unspecified whether the `sched_ss_repl_period` and `sched_ss_init_budget` values are stored as provided by this function or are rounded to align with the resolution of the clock being used.

If the `pthread_setschedparam()` function fails, the scheduling parameters shall not be changed for the target thread.

## RETURN VALUE

If successful, the `pthread_getschedparam()` and `pthread_setschedparam()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

## ERRORS

The `pthread_setschedparam()` function shall fail if:

- **ENOTSUP**: An attempt was made to set the policy or scheduling parameters to an unsupported value.
- **ENOTSUP**: An attempt was made to dynamically change the scheduling policy to `SCHED_SPORADIC`, and the implementation does not support this change.

The `pthread_setschedparam()` function may fail if:

- **EINVAL**: The value specified by `policy` or one of the scheduling parameters associated with the scheduling policy `policy` is invalid.
- **EPERM**: The caller does not have appropriate privileges to set either the scheduling parameters or the scheduling policy of the specified thread.
- **EPERM**: The implementation does not allow the application to modify one of the parameters to the value specified.

These functions shall not return an error code of `[EINTR]`.

---

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

If an implementation detects use of a thread ID after the end of its lifetime, it is recommended that the function should fail and report an `[ESRCH]` error.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_setschedprio()`
- `sched_getparam()`
- `sched_getscheduler()`
- XBD `<pthread.h>`
- XBD `<sched.h>`

# CHANGE HISTORY

## First released in Issue 5

Included for alignment with the POSIX Threads Extension.

## Issue 6

- The `pthread_getschedparam()` and `pthread_setschedparam()` functions are marked as part of the Threads and Thread Execution Scheduling options.
- The `[ENOSYS]` error condition has been removed as stubs need not be provided if an implementation does not support the Thread Execution Scheduling option.
- The Open Group Corrigendum U026/2 is applied, correcting the prototype for the `pthread_setschedparam()` function so that its second argument is of type `int`.
- The `SCHED_SPORADIC` scheduling policy is added for alignment with IEEE Std 1003.1d-1999.
- The `restrict` keyword is added to the `pthread_getschedparam()` prototype for alignment with the ISO/IEC 9899:1999 standard.
- The Open Group Corrigendum U047/1 is applied.
- IEEE PASC Interpretation 1003.1 #96 is applied, noting that priority values can also be set by a call to the `pthread_setschedprio()` function.

## Issue 7

- The `pthread_getschedparam()` and `pthread_setschedparam()` functions are marked only as part of the Thread Execution Scheduling option as the Threads option is now part of the Base.
- Austin Group Interpretation 1003.1-2001 #119 is applied, clarifying the accuracy requirements for the `sched_ss_repl_period` and `sched_ss_init_budget` values.
- Austin Group Interpretation 1003.1-2001 #142 is applied, removing the `[ESRCH]` error condition.
- POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0459 [314] is applied.

- POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0276 [757] is applied.
-

## 1.172. pthread\_setschedprio

---

### SYNOPSIS

```
#include <pthread.h>

int pthread_setschedprio(pthread_t thread, int prio);
```

### DESCRIPTION

The `pthread_setschedprio()` function shall set the scheduling priority for the thread whose thread ID is given by `thread` to the value given by `prio`. See Scheduling Policies for a description on how this function call affects the ordering of the thread in the thread list for its new priority.

If the `pthread_setschedprio()` function fails, the scheduling priority of the target thread shall not be changed.

### RETURN VALUE

If successful, the `pthread_setschedprio()` function shall return zero; otherwise, an error number shall be returned to indicate the error.

### ERRORS

The `pthread_setschedprio()` function may fail if:

- **[EINVAL]**

The value of `prio` is invalid for the scheduling policy of the specified thread.

- **[EPERM]**

The caller does not have appropriate privileges to set the scheduling priority of the specified thread.

The `pthread_setschedprio()` function shall not return an error code of [EINTR].

### EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

The `pthread_setschedprio()` function provides a way for an application to temporarily raise its priority and then lower it again, without having the undesired side-effect of yielding to other threads of the same priority. This is necessary if the application is to implement its own strategies for bounding priority inversion, such as priority inheritance or priority ceilings. This capability is especially important if the implementation does not support the Thread Priority Protection or Thread Priority Inheritance options, but even if those options are supported it is needed if the application is to bound priority inheritance for other resources, such as semaphores.

The standard developers considered that while it might be preferable conceptually to solve this problem by modifying the specification of `pthread_setschedparam()`, it was too late to make such a change, as there may be implementations that would need to be changed. Therefore, this new function was introduced.

If an implementation detects use of a thread ID after the end of its lifetime, it is recommended that the function should fail and report an [ESRCH] error.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [Scheduling Policies](#)
- `pthread_getschedparam()`
- `<pthread.h>`

## CHANGE HISTORY

First released in Issue 6. Included as a response to IEEE PASC Interpretation 1003.1 #96.

## Issue 7

The `pthread_setschedprio()` function is marked only as part of the Thread Execution Scheduling option as the Threads option is now part of the Base.

Austin Group Interpretation 1003.1-2001 #069 is applied, updating the [EPERM] error.

Austin Group Interpretation 1003.1-2001 #142 is applied, removing the [ESRCH] error condition.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0466 [314] is applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0296 [757] is applied.

## 1.173. `pthread_getspecific`, `pthread_setspecific` — thread-specific data management

### SYNOPSIS

```
#include <pthread.h>

void *pthread_getspecific(pthread_key_t key);

int pthread_setspecific(pthread_key_t key, const void *value);
```

### DESCRIPTION

The `pthread_getspecific()` function shall return the value currently bound to the specified `key` on behalf of the calling thread.

The `pthread_setspecific()` function shall associate a thread-specific `value` with a `key` obtained via a previous call to `pthread_key_create()`. Different threads may bind different values to the same key. These values are typically pointers to blocks of dynamically allocated memory that have been reserved for use by the calling thread.

The effect of calling `pthread_getspecific()` or `pthread_setspecific()` with a `key` value not obtained from `pthread_key_create()` or after `key` has been deleted with `pthread_key_delete()` is undefined.

Both `pthread_getspecific()` and `pthread_setspecific()` may be called from a thread-specific data destructor function. A call to `pthread_getspecific()` for the thread-specific data key being destroyed shall return the value `NULL`, unless the value is changed (after the destructor starts) by a call to `pthread_setspecific()`. Calling `pthread_setspecific()` from a thread-specific data destructor routine may result either in lost storage (after at least `PTHREAD_DESTRUCTOR_ITERATIONS` attempts at destruction) or in an infinite loop.

Both functions may be implemented as macros.

### RETURN VALUE

The `pthread_getspecific()` function shall return the thread-specific data value associated with the given `key`. If no thread-specific data value is associated with

`key` , then the value NULL shall be returned.

If successful, the `pthread_setspecific()` function shall return zero; otherwise, an error number shall be returned to indicate the error.

## ERRORS

No errors are returned from `pthread_getspecific()` .

The `pthread_setspecific()` function shall fail if:

- **[ENOMEM]**
- Insufficient memory exists to associate the non-NULL value with the key.

The `pthread_setspecific()` function shall not return an error code of [EINTR].

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

Performance and ease-of-use of `pthread_getspecific()` are critical for functions that rely on maintaining state in thread-specific data. Since no errors are required to be detected by it, and since the only error that could be detected is the use of an invalid key, the function to `pthread_getspecific()` has been designed to favor speed and simplicity over error reporting.

If an implementation detects that the value specified by the `key` argument to `pthread_setspecific()` does not refer to a key value obtained from `pthread_key_create()` or refers to a key that has been deleted with `pthread_key_delete()` , it is recommended that the function should fail and report an [EINVAL] error.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `pthread_key_create()`
- `<pthread.h>`

## CHANGE HISTORY

### First released in Issue 5

Included for alignment with the POSIX Threads Extension.

### Issue 6

- The `pthread_getspecific()` and `pthread_setspecific()` functions are marked as part of the Threads option.
- IEEE PASC Interpretation 1003.1c #3 (Part 6) is applied, updating the DESCRIPTION.
- IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/96 is applied, updating the ERRORS section so that the [ENOMEM] error case is changed from "to associate the value with the key" to "to associate the non-NUL value with the key".

### Issue 7

- Austin Group Interpretation 1003.1-2001 #063 is applied, updating the ERRORS section.
- The `pthread_getspecific()` and `pthread_setspecific()` functions are moved from the Threads option to the Base.
- The [EINVAL] error for a key value not obtained from `pthread_key_create()` or a key deleted with `pthread_key_delete()` is removed; this condition results in undefined behavior.

## 1.174. `pthread_sigmask`, `sigprocmask` — examine and change blocked signals

### SYNOPSIS

```
#include <signal.h>

int pthread_sigmask(int how, const sigset_t *restrict set,
                     sigset_t *restrict oset);
int sigprocmask(int how, const sigset_t *restrict set,
                 sigset_t *restrict oset);
```

### DESCRIPTION

The `pthread_sigmask()` function shall examine or change (or both) the calling thread's signal mask.

If the argument `set` is not a null pointer, it points to a set of signals to be used to change the currently blocked set.

The argument `how` indicates the way in which the set is changed, and the application shall ensure it consists of one of the following values:

#### `SIG_BLOCK`

- The resulting set shall be the union of the current set and the signal set pointed to by `set`.

#### `SIG_SETMASK`

- The resulting set shall be the signal set pointed to by `set`.

#### `SIG_UNBLOCK`

- The resulting set shall be the intersection of the current set and the complement of the signal set pointed to by `set`.

If the argument `oset` is not a null pointer, the previous mask shall be stored in the location pointed to by `oset`. If `set` is a null pointer, the value of the argument `how` is not significant and the thread's signal mask shall be unchanged; thus the call can be used to enquire about currently blocked signals.

If the argument `set` is not a null pointer, after `pthread_sigmask()` changes the currently blocked set of signals it shall determine whether there are any pending unblocked signals; if there are any, then at least one of those signals shall be delivered before the call to `pthread_sigmask()` returns.

It is not possible to block those signals which cannot be ignored. This shall be enforced by the system without causing an error to be indicated.

If any of the SIGFPE, SIGILL, SIGSEGV, or SIGBUS signals are generated while they are blocked, the result is undefined, unless the signal was generated by the action of another process, or by one of the functions `kill()`, `pthread_kill()`, `raise()`, or `sigqueue()`.

If `pthread_sigmask()` fails, the thread's signal mask shall not be changed.

The `sigprocmask()` function shall be equivalent to `pthread_sigmask()`, except that its behavior is unspecified if called from a multi-threaded process, and on error it returns -1 and sets `errno` to the error number instead of returning the error number directly.

## RETURN VALUE

Upon successful completion, `pthread_sigmask()` shall return 0; otherwise, it shall return the corresponding error number.

Upon successful completion, `sigprocmask()` shall return 0; otherwise, -1 shall be returned and `errno` shall be set to indicate the error.

## ERRORS

These functions shall fail if:

### [EINVAL]

- The `set` argument is not a null pointer and the value of the `how` argument is not equal to one of the defined values.

These functions shall not return an error code of [EINTR].

## EXAMPLES

### Signaling in a Multi-Threaded Process

This example shows the use of `pthread_sigmask()` in order to deal with signals in a multi-threaded process. It provides a fairly general framework that could be easily adapted/extended.

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

```

#include <signal.h>
#include <string.h>
#include <errno.h>
...
static sigset_t    signal_mask; /* signals to block */
```

```

int main (int argc, char *argv[])
{
    pthread_t    sig_thr_id;      /* signal handler thread ID */
    int          rc;             /* return code */
```

```

    sigemptyset (&signal_mask);
    sigaddset (&signal_mask, SIGINT);
    sigaddset (&signal_mask, SIGTERM);
    rc = pthread_sigmask (SIG_BLOCK, &signal_mask, NULL);
    if (rc != 0) {
        /* handle error */
        ...
    }
    /* any newly created threads inherit the signal mask */

    rc = pthread_create (&sig_thr_id, NULL, signal_thread, NULL);
    if (rc != 0) {
        /* handle error */
        ...
    }
    /* APPLICATION CODE */
    ...
}
```

```

void *signal_thread (void *arg)
{
    int      sig_caught;      /* signal caught */
    int      rc;              /* returned code */

    rc = sigwait (&signal_mask, &sig_caught);
    if (rc != 0) {
        /* handle error */
    }
    switch (sig_caught)
    {
        case SIGINT:      /* process SIGINT */
        ...
        break;
        case SIGTERM:     /* process SIGTERM */
        ...
        break;
        default:         /* should normally not happen */
            fprintf (stderr, "\nUnexpected signal %d\n", sig_caught);
```

```
        break;  
    }  
}
```

## APPLICATION USAGE

Although `pthread_sigmask()` has to deliver at least one of any pending unblocked signals that exist after it has changed the currently blocked set of signals, there is no requirement that the delivered signal(s) include any that were unblocked by the change. If one or more signals that were already unblocked become pending (see 2.4.1 Signal Generation and Delivery) during the period the `pthread_sigmask()` call is executing, the signal(s) delivered before the call returns might include only those signals.

## RATIONALE

When a thread's signal mask is changed in a signal-catching function that is installed by `sigaction()`, the restoration of the signal mask on return from the signal-catching function overrides that change (see `sigaction()`). If the signal-catching function was installed with `signal()`, it is unspecified whether this occurs.

See `kill()` for a discussion of the requirement on delivery of signals.

## FUTURE DIRECTIONS

None.

## SEE ALSO

`exec` , `kill()` , `sigaction()` , `sigaddset()` , `sigdelset()` ,  
`sigemptyset()` , `sigfillset()` , `sigismember()` , `sigpending()` ,  
`sigqueue()` , `sigsuspend()`

XBD `<signal.h>`

## CHANGE HISTORY

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

## Issue 5

- The DESCRIPTION is updated for alignment with the POSIX Threads Extension.
- The `pthread_sigmask()` function is added for alignment with the POSIX Threads Extension.

## Issue 6

- The `pthread_sigmask()` function is marked as part of the Threads option.
- The SYNOPSIS for `sigprocmask()` is marked as a CX extension to note that the presence of this function in the `<signal.h>` header is an extension to the ISO C standard.
- The following changes are made for alignment with the ISO POSIX-1:1996 standard:
  - The DESCRIPTION is updated to explicitly state the functions which may generate the signal.
  - The normative text is updated to avoid use of the term "must" for application requirements.
  - The `restrict` keyword is added to the `pthread_sigmask()` and `sigprocmask()` prototypes for alignment with the ISO/IEC 9899:1999 standard.
- IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/105 is applied, updating "process' signal mask" to "thread's signal mask" in the DESCRIPTION and RATIONALE sections.
- IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/106 is applied, adding the example to the EXAMPLES section.

## Issue 7

- The `pthread_sigmask()` function is moved from the Threads option to the Base.
- POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0467 [319] is applied.

## Issue 8

- Austin Group Defect 1132 is applied, clarifying the [EINVAL] error.

- Austin Group Defect 1636 is applied, clarifying the exceptions to the equivalence of `pthread_sigmask()` and `sigprocmask()`.
- Austin Group Defect 1731 is applied, clarifying that although `pthread_sigmask()` has to deliver at least one of any pending unblocked signals that exist after it has changed the currently blocked set of signals, there is no requirement that the delivered signal(s) include any that were unblocked by the change.

## 1.175. `pthread_setcancelstate`, `pthread_setcanceltype`, `pthread_testcancel`

---

### SYNOPSIS

```
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
void pthread_testcancel(void);
```

### DESCRIPTION

The `pthread_setcancelstate()` function shall atomically both set the calling thread's cancelability state to the indicated `state` and return the previous cancelability state at the location referenced by `oldstate`. Legal values for `state` are `PTHREAD_CANCEL_ENABLE` and `PTHREAD_CANCEL_DISABLE`.

The `pthread_setcanceltype()` function shall atomically both set the calling thread's cancelability type to the indicated `type` and return the previous cancelability type at the location referenced by `oldtype`. Legal values for `type` are `PTHREAD_CANCEL_DEFERRED` and `PTHREAD_CANCEL_ASYNCHRONOUS`.

The cancelability state and type of any newly created threads, including the thread in which `main()` was first invoked, shall be `PTHREAD_CANCEL_ENABLE` and `PTHREAD_CANCEL_DEFERRED` respectively.

The `pthread_testcancel()` function shall create a cancellation point in the calling thread. The `pthread_testcancel()` function shall have no effect if cancelability is disabled.

The `pthread_setcancelstate()` function shall be async-signal-safe.

### RETURN VALUE

If successful, the `pthread_setcancelstate()` and `pthread_setcanceltype()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

# ERRORS

The `pthread_setcancelstate()` function may fail if:

- **[EINVAL]**
- The specified state is not `PTHREAD_CANCEL_ENABLE` or `PTHREAD_CANCEL_DISABLE`.

The `pthread_setcanceltype()` function may fail if:

- **[EINVAL]**
- The specified type is not `PTHREAD_CANCEL_DEFERRED` or `PTHREAD_CANCEL_ASYNCHRONOUS`.

These functions shall not return an error code of `[EINTR]`.

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

In order to write a signal handler for an asynchronous signal which can run safely in a cancellable thread, `pthread_setcancelstate()` must be used to disable cancellation for the duration of any calls that the signal handler makes which are cancellation points. However, earlier versions of the standard did not permit strictly conforming applications to call `pthread_setcancelstate()` from a signal handler since it was not required to be async-signal-safe. On non-conforming implementations where `pthread_setcancelstate()` is not async-signal-safe, alternatives are to ensure either that the corresponding signals are blocked during execution of functions that are not async-cancel-safe or that cancellation is disabled during times when those signals could be delivered.

## RATIONALE

The `pthread_setcancelstate()` and `pthread_setcanceltype()` functions control the points at which a thread may be asynchronously canceled. For

cancellation control to be usable in modular fashion, some rules need to be followed.

An object can be considered to be a generalization of a procedure. It is a set of procedures and global variables written as a unit and called by clients not known by the object. Objects may depend on other objects.

First, cancelability should only be disabled on entry to an object, never explicitly enabled. On exit from an object, the cancelability state should always be restored to its value on entry to the object.

This follows from a modularity argument: if the client of an object (or the client of an object that uses that object) has disabled cancelability, it is because the client does not want to be concerned about cleaning up if the thread is canceled while executing some sequence of actions. If an object is called in such a state and it enables cancelability and a cancellation request is pending for that thread, then the thread is canceled, contrary to the wish of the client that disabled.

Second, the cancelability type may be explicitly set to either deferred or asynchronous upon entry to an object. But as with the cancelability state, on exit from an object the cancelability type should always be restored to its value on entry to the object.

Finally, only functions that are cancel-safe may be called from a thread that is asynchronously cancelable.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [pthread\\_cancel\(\)](#)
- XBD [<pthread.h>](#)

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

## Issue 6

The `pthread_setcancelstate()`, `pthread_setcanceltype()`, and `pthread_testcancel()` functions are marked as part of the Threads option.

## Issue 7

The `pthread_setcancelstate()`, `pthread_setcanceltype()`, and `pthread_testcancel()` functions are moved from the Threads option to the Base.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0294 [622] and XSH/TC2-2008/0295 [615] are applied.

## Issue 8

Austin Group Defect 841 is applied, requiring `pthread_setcancelstate()` to be async-signal-safe.

## 1.176. putc — put a byte on a stream

---

### SYNOPSIS

```
#include <stdio.h>

int putc(int c, FILE *stream);
```

### DESCRIPTION

The `putc()` function shall be equivalent to `fputc()`, except that if it is implemented as a macro it may evaluate `stream` more than once, so the argument should never be an expression with side-effects.

### RETURN VALUE

Refer to `fputc()`.

### ERRORS

Refer to `fputc()`.

### EXAMPLES

None.

### APPLICATION USAGE

Since it may be implemented as a macro, `putc()` may treat a `stream` argument with side-effects incorrectly. In particular, `putc(c,*f++)` does not necessarily work correctly. Therefore, use of this function is not recommended in such situations; `fputc()` should be used instead.

### RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

2.5 Standard I/O Streams, [fputc\(\)](#)

XBD [<stdio.h>](#)

## CHANGE HISTORY

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

### Issue 7

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0470 [14] is applied.

---

## 1.177. putc\_unlocked

---

### SYNOPSIS

```
#include <stdio.h>

int getc_unlocked(FILE *stream);
int getchar_unlocked(void);
int putc_unlocked(int c, FILE *stream);
int putchar_unlocked(int c);
```

### DESCRIPTION

Versions of the functions `getc()`, `getchar()`, `putc()`, and `putchar()` respectively named `getc_unlocked()`, `getchar_unlocked()`, `putc_unlocked()`, and `putchar_unlocked()` shall be provided which are functionally equivalent to the original versions, with the exception that they are not required to be implemented in a fully thread-safe manner. They shall be thread-safe when used within a scope protected by `flockfile()` (or `ftrylockfile()`) and `funlockfile()`. These functions can safely be used in a multi-threaded program if and only if they are called while the invoking thread owns the `(FILE *)` object, as is the case after a successful call to the `flockfile()` or `ftrylockfile()` functions.

If `getc_unlocked()` or `putc_unlocked()` are implemented as macros they may evaluate `stream` more than once, so the `stream` argument should never be an expression with side-effects.

### RETURN VALUE

See `getc()`, `getchar()`, `putc()`, and `putchar()`.

### ERRORS

See `getc()`, `getchar()`, `putc()`, and `putchar()`.

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

Since they may be implemented as macros, `getc_unlocked()` and `putc_unlocked()` may treat incorrectly a stream argument with side-effects. In particular, `getc_unlocked(f++)` and `putc_unlocked(c,f++)` do not necessarily work as expected. Therefore, use of these functions in such situations should be preceded by the following statement as appropriate:

```
#undef getc_unlocked
#undef putc_unlocked
```

## RATIONALE

Some I/O functions are typically implemented as macros for performance reasons (for example, `putc()` and `getc()`). For safety, they need to be synchronized, but it is often too expensive to synchronize on every character. Nevertheless, it was felt that the safety concerns were more important; consequently, the `getc()`, `getchar()`, `putc()`, and `putchar()` functions are required to be thread-safe. However, unlocked versions are also provided with names that clearly indicate the unsafe nature of their operation but can be used to exploit their higher performance. These unlocked versions can be safely used only within explicitly locked program regions, using exported locking primitives. In particular, a sequence such as:

```
flockfile(fileptr);
putc_unlocked('1', fileptr);
putc_unlocked('\n', fileptr);
fprintf(fileptr, "Line 2\n");
funlockfile(fileptr);
```

is permissible, and results in the text sequence:

```
1
Line 2
```

being printed without being interspersed with output from other threads.

It would be wrong to have the standard names such as `getc()`, `putc()`, and so on, map to the "faster, but unsafe" rather than the "slower, but safe" versions. In either case, you would still want to inspect all uses of `getc()`, `putc()`, and so on, by hand

when converting existing code. Choosing the safe bindings as the default, at least, results in correct code and maintains the "atomicity at the function" invariant. To do otherwise would introduce gratuitous synchronization errors into converted code. Other routines that modify the stdio (FILE \*) structures or buffers are also safely synchronized.

Note that there is no need for functions of the form getc\_locked(), putc\_locked(), and so on, since this is the functionality of getc(), putc(), et al. It would be inappropriate to use a feature test macro to switch a macro definition of getc() between getc\_locked() and getc\_unlocked(), since the ISO C standard requires an actual function to exist, a function whose behavior could not be changed by the feature test macro. Also, providing both the xxx\_locked() and xxx\_unlocked() forms leads to the confusion of whether the suffix describes the behavior of the function or the circumstances under which it should be used.

Three additional routines, flockfile(), ftrylockfile(), and funlockfile() (which may be macros), are provided to allow the user to delineate a sequence of I/O statements that are executed synchronously.

The ungetc() function is infrequently called relative to the other functions/macros so no unlocked variation is needed.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [2.5 Standard I/O Streams](#)
- [flockfile\(\)](#)
- [getc\(\)](#)
- [getchar\(\)](#)
- [putc\(\)](#)
- [putchar\(\)](#)
- XBD

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

## Issue 6

These functions are marked as part of the Thread-Safe Functions option.

The Open Group Corrigendum U030/2 is applied, adding APPLICATION USAGE describing how applications should be written to avoid the case when the functions are implemented as macros.

## Issue 7

The `getc_unlocked()`, `getchar_unlocked()`, `putc_unlocked()`, and `putchar_unlocked()` functions are moved from the Thread-Safe Functions option to the Base.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0232 [395], XSH/TC1-2008/0233 [395], XSH/TC1-2008/0234 [395], and XSH/TC1-2008/0235 [14] are applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0151 [826] is applied.

---

## 1.178. putchar

---

### SYNOPSIS

```
#include <stdio.h>

int putchar(int c);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The function call `putchar(c)` shall be equivalent to `putc(c, stdout)`.

### RETURN VALUE

Refer to `fputc()`.

### ERRORS

Refer to `fputc()`.

---

*The following sections are informative.*

### EXAMPLES

None.

### APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- 2.5 Standard I/O Streams
- `putc()`
- XBD `<stdio.h>`

## CHANGE HISTORY

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

### Issue 7

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0471 [14] is applied.

---

## 1.179. putchar\_unlocked

---

### SYNOPSIS

```
#include <stdio.h>

int getc_unlocked(FILE *stream);
int getchar_unlocked(void);
int putc_unlocked(int c, FILE *stream);
int putchar_unlocked(int c);
```

### DESCRIPTION

Versions of the functions `getc()`, `getchar()`, `putc()`, and `putchar()` respectively named `getc_unlocked()`, `getchar_unlocked()`, `putc_unlocked()`, and `putchar_unlocked()` shall be provided which are functionally equivalent to the original versions, with the exception that they are not required to be implemented in a fully thread-safe manner. They shall be thread-safe when used within a scope protected by `flockfile()` (or `ftrylockfile()`) and `funlockfile()`. These functions can safely be used in a multi-threaded program if and only if they are called while the invoking thread owns the `(FILE *)` object, as is the case after a successful call to the `flockfile()` or `ftrylockfile()` functions.

If `getc_unlocked()` or `putc_unlocked()` are implemented as macros they may evaluate stream more than once, so the stream argument should never be an expression with side-effects.

### RETURN VALUE

See `getc()`, `getchar()`, `putc()`, and `putchar()`.

### ERRORS

See `getc()`, `getchar()`, `putc()`, and `putchar()`.

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

Since they may be implemented as macros, `getc_unlocked()` and `putc_unlocked()` may treat incorrectly a stream argument with side-effects. In particular, `getc_unlocked(*f++)` and `putc_unlocked(c,*f++)` do not necessarily work as expected. Therefore, use of these functions in such situations should be preceded by the following statement as appropriate:

```
#undef getc_unlocked
#undef putc_unlocked
```

## RATIONALE

Some I/O functions are typically implemented as macros for performance reasons (for example, `putc()` and `getc()`). For safety, they need to be synchronized, but it is often too expensive to synchronize on every character. Nevertheless, it was felt that the safety concerns were more important; consequently, the `getc()`, `getchar()`, `putc()`, and `putchar()` functions are required to be thread-safe. However, unlocked versions are also provided with names that clearly indicate the unsafe nature of their operation but can be used to exploit their higher performance. These unlocked versions can be safely used only within explicitly locked program regions, using exported locking primitives. In particular, a sequence such as:

```
flockfile(fileptr);
putc_unlocked('1', fileptr);
putc_unlocked('\n', fileptr);
fprintf(fileptr, "Line 2\n");
funlockfile(fileptr);
```

is permissible, and results in the text sequence:

```
1
Line 2
```

being printed without being interspersed with output from other threads.

It would be wrong to have the standard names such as `getc()`, `putc()`, and so on, map to the "faster, but unsafe" rather than the "slower, but safe" versions. In

either case, you would still want to inspect all uses of `getc()`, `putc()`, and so on, by hand when converting existing code. Choosing the safe bindings as the default, at least, results in correct code and maintains the "atomicity at the function" invariant. To do otherwise would introduce gratuitous synchronization errors into converted code. Other routines that modify the stdio (FILE \*) structures or buffers are also safely synchronized.

Note that there is no need for functions of the form `getc_locked()`, `putc_locked()`, and so on, since this is the functionality of `getc()`, `putc()`, et al. It would be inappropriate to use a feature test macro to switch a macro definition of `getc()` between `getc_locked()` and `getc_unlocked()`, since the ISO C standard requires an actual function to exist, a function whose behavior could not be changed by the feature test macro. Also, providing both the `xxx_locked()` and `xxx_unlocked()` forms leads to the confusion of whether the suffix describes the behavior of the function or the circumstances under which it should be used.

Three additional routines, `flockfile()`, `ftrylockfile()`, and `funlockfile()` (which may be macros), are provided to allow the user to delineate a sequence of I/O statements that are executed synchronously.

The `ungetc()` function is infrequently called relative to the other functions/macros so no unlocked variation is needed.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [2.5 Standard I/O Streams](#)
- `flockfile()`
- `getc()`
- `getchar()`
- `putc()`
- `putchar()`

XBD

# CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

## Issue 6

These functions are marked as part of the Thread-Safe Functions option.

The Open Group Corrigendum U030/2 is applied, adding APPLICATION USAGE describing how applications should be written to avoid the case when the functions are implemented as macros.

## Issue 7

The `getc_unlocked()`, `getchar_unlocked()`, `putc_unlocked()`, and `putchar_unlocked()` functions are moved from the Thread-Safe Functions option to the Base.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0232 [395], XSH/TC1-2008/0233 [395], XSH/TC1-2008/0234 [395], and XSH/TC1-2008/0235 [14] are applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0151 [826] is applied.

## 1.180. puts

---

### SYNOPSIS

```
#include <stdio.h>

int puts(const char *s);
```

### DESCRIPTION

The `puts()` function shall write the string pointed to by `s`, followed by a newline, to the standard output stream `stdout`. The terminating null byte shall not be written.

The last data modification and last file status change timestamps of the file shall be marked for update between the successful execution of `puts()` and the next successful completion of a call to `fflush()` or `fclose()` on the same stream or a call to `exit()` or `abort()`.

### RETURN VALUE

Upon successful completion, `puts()` shall return a non-negative number. Otherwise, it shall return EOF, shall set an error indicator for the stream, and `errno` shall be set to indicate the error.

### ERRORS

Refer to `fputc()`.

---

*The following sections are informative.*

# EXAMPLES

## Printing to Standard Output

The following example gets the current time, converts it to a string using `localtime()` and `asctime()`, and prints it to standard output using `puts()`. It then prints the number of minutes to an event for which it is waiting.

```
#include <time.h>
#include <stdio.h>
...
time_t now;
int minutes_to_event;
...
time(&now);
printf("The time is ");
puts(asctime(localtime(&now)));
printf("There are %d minutes to the event.\n",
       minutes_to_event);
...
```

## APPLICATION USAGE

The `puts()` function appends a newline, while `fputs()` does not.

This volume of POSIX.1-2024 requires that successful completion simply return a non-negative integer. There are at least three known different implementation conventions for this requirement:

- Return a constant value.
- Return the last character written.
- Return the number of bytes written. Note that this implementation convention cannot be adhered to for strings longer than `{INT_MAX}` bytes as the value would not be representable in the return type of the function. For backwards compatibility, implementations can return the number of bytes for strings of up to `{INT_MAX}` bytes, and return `{INT_MAX}` for all longer strings.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

[2.5 Standard I/O Streams, fopen\(\), fputs\(\), putc\(\)](#)

XBD

## CHANGE HISTORY

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

### Issue 6

Extensions beyond the ISO C standard are marked.

### Issue 7

Changes are made related to support for finegrained timestamps.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0476 [174,412] and XSH/TC1-2008/0477 [14] are applied.

## 1.181. `qsort`, `qsort_r` — sort a table of data

## SYNOPSIS

```
#include <stdlib.h>

void qsort(void *base, size_t nel, size_t width,
           int (*compar)(const void *, const void *));

[CX] void qsort_r(void *base, size_t nel, size_t width,
                  int (*compar)(const void *, const void *, void *)
```

## DESCRIPTION

For `qsort()` : [CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `qsort()` function shall sort an array of `nel` objects, the initial element of which is pointed to by `base`. The size of each object, in bytes, is specified by the `width` argument. If the `nel` argument has the value zero, the comparison function pointed to by `compar` shall not be called and no rearrangement shall take place.

The application shall ensure that the comparison function pointed to by `compar` does not alter the contents of the array. The implementation may reorder elements of the array between calls to the comparison function, but shall not alter the contents of any individual element.

When the same objects (consisting of width bytes, irrespective of their current positions in the array) are passed more than once to the comparison function, the results shall be consistent with one another. That is, they shall define a total ordering on the array.

The contents of the array shall be sorted in ascending order according to a comparison function. The `compar` argument is a pointer to the comparison function, which is called with two arguments that point to the elements being compared. The application shall ensure that the function returns an integer less than, equal to, or greater than 0, if the first argument is considered respectively less than, equal to, or greater than the second. If two members compare as equal, their order in the sorted array is unspecified.

[CX] The `qsort_r()` function shall be identical to `qsort()` except that the comparison function `compar` takes a third argument. The `arg` opaque pointer passed to `qsort_r()` shall in turn be passed as the third argument to the comparison function.

## RETURN VALUE

These functions shall not return a value.

## ERRORS

No errors are defined.

## APPLICATION USAGE

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

If the `compar` callback function requires any additional state outside of the items being sorted, it can only access this state through global variables, making it potentially unsafe to use `qsort()` with the same `compar` function from separate threads at the same time. The `qsort_r()` function was added with the ability to pass through arbitrary arguments to the comparator, which avoids the need to access global variables and thus making it possible to safely share a stateful comparator across threads.

## RATIONALE

The requirement that each argument (hereafter referred to as `p`) to the comparison function is a pointer to elements of the array implies that for every call, for each argument separately, all of the following expressions are non-zero:

```
((char *)p - (char *)base) % width == 0  
(char *)p >= (char *)base  
(char *)p < (char *)base + nel * width
```

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `alphasort()`
- `<stdlib.h>`

## CHANGE HISTORY

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

### Issue 6

The normative text is updated to avoid use of the term "must" for application requirements.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/49 is applied, adding the last sentence to the first non-shaded paragraph in the DESCRIPTION, and the following two paragraphs. The RATIONALE is also updated. These changes are for alignment with the ISO C standard.

### Issue 8

Austin Group Defect 900 is applied, adding the `qsort_r()` function.

---

## 1.182. raise

---

### SYNOPSIS

```
#include <signal.h>

int raise(int sig);
```

### DESCRIPTION

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `raise()` function shall send the signal `sig` to the executing thread or process. If a signal handler is called, the `raise()` function shall not return until after the signal handler does.

The effect of the `raise()` function shall be equivalent to calling:

```
pthread_kill(pthread_self(), sig);
```

### RETURN VALUE

Upon successful completion, 0 shall be returned. Otherwise, a non-zero value shall be returned and `errno` shall be set to indicate the error.

### ERRORS

The `raise()` function shall fail if:

- **[EINVAL]**

The value of the `sig` argument is an invalid signal number.

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

The term "thread" is an extension to the ISO C standard.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `kill()`
- `sigaction()`
- XBD `<signal.h>`
- XBD `<sys/types.h>`

## CHANGE HISTORY

First released in Issue 4. Derived from the ANSI C standard.

### Issue 5

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

### Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE section, the requirement to set `errno` on error is added.
- The [EINVAL] error condition is added.

## Issue 7

Functionality relating to the Threads option is moved to the Base.

## 1.183. `rand`, `srand` — pseudo-random number generator

---

### SYNOPSIS

```
#include <stdlib.h>

int rand(void);
void srand(unsigned seed);
```

### DESCRIPTION

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `rand()` function shall compute a sequence of pseudo-random integers in the range  $[0, \{RAND\_MAX\}]$  with a period of at least  $2^{32}$ .

The `rand()` function need not be thread-safe; however, `rand()` shall avoid data races with all functions other than non-thread-safe pseudo-random sequence generation functions.

The `srand()` function uses the argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to `rand()`. If `srand()` is then called with the same seed value, the sequence of pseudo-random numbers shall be repeated. If `rand()` is called before any calls to `srand()` are made, the same sequence shall be generated as when `srand()` is first called with a seed value of 1.

The `srand()` function need not be thread-safe; however, `srand()` shall avoid data races with all functions other than non-thread-safe pseudo-random sequence generation functions.

The implementation shall behave as if no function defined in this volume of POSIX.1-2024 calls `rand()` or `srand()`.

### RETURN VALUE

The `rand()` function shall return the next pseudo-random number in the sequence.

The `srand()` function shall not return a value.

## ERRORS

No errors are defined.

---

## EXAMPLES

### Generating a Pseudo-Random Number Sequence

The following example demonstrates how to generate a sequence of pseudo-random numbers.

```
#include <stdio.h>
#include <stdlib.h>
...
long count, i;
char *keystr;
int elementlen, len;
char c;
...
/* Initial random number generator. */
srand(1);

/* Create keys using only lowercase characters */
len = 0;
for (i=0; i<count; i++) {
    while (len < elementlen) {
        c = (char) (rand() % 128);
        if (islower(c))
            keystr[len++] = c;
    }

    keystr[len] = '\0';
    printf("%s Element%0*ld\n", keystr, elementlen, i);
    len = 0;
}
```

### Generating the Same Sequence on Different Machines

The following code defines a pair of functions that could be incorporated into applications wishing to ensure that the same sequence of numbers is generated across different machines.

```

static unsigned long next = 1;

int myrand(void) /* RAND_MAX assumed to be 32767. */
{
    next = next * 1103515245 + 12345;
    return((unsigned)(next/65536) % 32768);
}

void mysrand(unsigned seed)
{
    next = seed;
}

```

---

## APPLICATION USAGE

These functions should be avoided whenever non-trivial requirements (including safety) have to be fulfilled, unless seeded using `getentropy()`.

The `drand48()` and `random()` functions provide much more elaborate pseudo-random number generators.

---

## RATIONALE

The ISO C standard `rand()` and `srand()` functions allow per-process pseudo-random streams shared by all threads. Those two functions need not change, but there has to be mutual-exclusion that prevents interference between two threads concurrently accessing the random number generator.

With regard to `rand()`, there are two different behaviors that may be wanted in a multi-threaded program:

1. A single per-process sequence of pseudo-random numbers that is shared by all threads that call `rand()`
2. A different sequence of pseudo-random numbers for each thread that calls `rand()`

This is provided by the modified thread-safe function based on whether the seed value is global to the entire process or local to each thread.

This does not address the known deficiencies of the `rand()` function implementations, which have been approached by maintaining more state. In effect, this specifies new thread-safe forms of a deficient function.

---

## FUTURE DIRECTIONS

None.

---

## SEE ALSO

- `drand48()`
  - `getentropy()`
  - `initstate()`
  - `<stdlib.h>`
- 

## CHANGE HISTORY

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

### Issue 5

The `rand_r()` function is included for alignment with the POSIX Threads Extension.

A note indicating that the `rand()` function need not be reentrant is added to the DESCRIPTION.

### Issue 6

Extensions beyond the ISO C standard are marked.

The `rand_r()` function is marked as part of the Thread-Safe Functions option.

### Issue 7

Austin Group Interpretation 1003.1-2001 #156 is applied.

The `rand_r()` function is marked obsolescent.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0301 [743] is applied.

## Issue 8

Austin Group Defect 1134 is applied, adding `getentropy()`.

Austin Group Defect 1302 is applied, aligning these functions with the ISO/IEC 9899:2018 standard.

Austin Group Defect 1330 is applied, removing obsolescent interfaces.

---

## 1.184. `rand`, `rand_r`, `srand` - pseudo-random number generator

### SYNOPSIS

```
#include <stdlib.h>

int rand(void);

[OB CX] int rand_r(unsigned *seed);

void srand(unsigned seed);
```

### DESCRIPTION

For `rand()` and `srand()`: [CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2017 defers to the ISO C standard.

The `rand()` function shall compute a sequence of pseudo-random integers in the range  $[0, \{RAND\_MAX\}]$  [XSI] with a period of at least  $2^{32}$ .

[CX] The `rand()` function need not be thread-safe.

[OB CX] The `rand_r()` function shall compute a sequence of pseudo-random integers in the range  $[0, \{RAND\_MAX\}]$ . (The value of the `\{RAND_MAX\}` macro shall be at least 32767.)

If `rand_r()` is called with the same initial value for the object pointed to by `seed` and that object is not modified between successive returns and calls to `rand_r()`, the same sequence shall be generated.

The `srand()` function uses the argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to `rand()`. If `srand()` is then called with the same seed value, the sequence of pseudo-random numbers shall be repeated. If `rand()` is called before any calls to `srand()` are made, the same sequence shall be generated as when `srand()` is first called with a seed value of 1.

The implementation shall behave as if no function defined in this volume of POSIX.1-2017 calls `rand()` or `srand()`.

# RETURN VALUE

The `rand()` function shall return the next pseudo-random number in the sequence.

[OB CX] The `rand_r()` function shall return a pseudo-random integer.

The `srand()` function shall not return a value.

# ERRORS

No errors are defined.

# EXAMPLES

## Generating a Pseudo-Random Number Sequence

The following example demonstrates how to generate a sequence of pseudo-random numbers.

```
#include <stdio.h>
#include <stdlib.h>
...
long count, i;
char *keystr;
int elementlen, len;
char c;
...
/* Initial random number generator. */
srand(1);

/* Create keys using only lowercase characters */
len = 0;
for (i=0; i<count; i++) {
    while (len < elementlen) {
        c = (char) (rand() % 128);
        if (islower(c))
            keystr[len++] = c;
    }

    keystr[len] = '\0';
    printf("%s Element%0*ld\n", keystr, elementlen, i);
    len = 0;
}
```

## Generating the Same Sequence on Different Machines

The following code defines a pair of functions that could be incorporated into applications wishing to ensure that the same sequence of numbers is generated across different machines.

```
static unsigned long next = 1;

int myrand(void) /* RAND_MAX assumed to be 32767. */
{
    next = next * 1103515245 + 12345;
    return((unsigned)(next/65536) % 32768);
}

void mysrand(unsigned seed)
{
    next = seed;
}
```

## APPLICATION USAGE

The `drand48()` and `random()` functions provide much more elaborate pseudo-random number generators.

The limitations on the amount of state that can be carried between one function call and another mean the `rand_r()` function can never be implemented in a way which satisfies all of the requirements on a pseudo-random number generator.

These functions should be avoided whenever non-trivial requirements (including safety) have to be fulfilled.

## RATIONALE

The ISO C standard `rand()` and `srand()` functions allow per-process pseudo-random streams shared by all threads. Those two functions need not change, but there has to be mutual-exclusion that prevents interference between two threads concurrently accessing the random number generator.

With regard to `rand()`, there are two different behaviors that may be wanted in a multi-threaded program:

1. A single per-process sequence of pseudo-random numbers that is shared by all threads that call `rand()`
2. A different sequence of pseudo-random numbers for each thread that calls `rand()`

This is provided by the modified thread-safe function based on whether the seed value is global to the entire process or local to each thread.

This does not address the known deficiencies of the `rand()` function implementations, which have been approached by maintaining more state. In effect, this specifies new thread-safe forms of a deficient function.

## FUTURE DIRECTIONS

The `rand_r()` function may be removed in a future version.

## SEE ALSO

`drand48()`, `initstate()`

XBD `<stdlib.h>`

## CHANGE HISTORY

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

### Issue 5

The `rand_r()` function is included for alignment with the POSIX Threads Extension.

A note indicating that the `rand()` function need not be reentrant is added to the DESCRIPTION.

### Issue 6

Extensions beyond the ISO C standard are marked.

The `rand_r()` function is marked as part of the Thread-Safe Functions option.

### Issue 7

Austin Group Interpretation 1003.1-2001 #156 is applied.

The `rand_r()` function is marked obsolescent.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0301 [743] is applied.

## 1.185. read

---

### SYNOPSIS

```
#include <unistd.h>

ssize_t pread(int fildes, void *buf, size_t nbyte, off_t offset);
ssize_t read(int fildes, void *buf, size_t nbyte);
```

### DESCRIPTION

The `read()` function shall attempt to read `nbyte` bytes from the file associated with the open file descriptor, `fildes`, into the buffer pointed to by `buf`. The behavior of multiple concurrent reads on the same pipe, FIFO, or terminal device is unspecified.

Before any action described below is taken, and if `nbyte` is zero, the `read()` function may detect and return errors as described below. In the absence of errors, or if error detection is not performed, the `read()` function shall return zero and have no other results.

On files that support seeking (for example, a regular file), the `read()` shall start at a position in the file given by the file offset associated with `fildes`. The file offset shall be incremented by the number of bytes actually read.

Files that do not support seeking—for example, terminals—always read from the current position. The value of a file offset associated with such a file is undefined.

No data transfer shall occur past the current end-of-file. If the starting position is at or after the end-of-file, 0 shall be returned. If the file refers to a device special file, the result of subsequent `read()` requests is implementation-defined.

If the value of `nbyte` is greater than `{SSIZE_MAX}`, the result is implementation-defined.

When attempting to read from an empty pipe or FIFO:

- If no process has the pipe open for writing, `read()` shall return 0 to indicate end-of-file.
- If some process has the pipe open for writing and `O_NONBLOCK` is set, `read()` shall return -1 and set `errno` to [EAGAIN].
- If some process has the pipe open for writing and `O_NONBLOCK` is clear, `read()` shall block the calling thread until some data is written or the pipe is

closed by all processes that had the pipe open for writing.

When attempting to read a file (other than a pipe or FIFO) that supports non-blocking reads and has no data currently available:

- If O\_NONBLOCK is set, `read()` shall return -1 and set `errno` to [EAGAIN].
- If O\_NONBLOCK is clear, `read()` shall block the calling thread until some data becomes available.
- The use of the O\_NONBLOCK flag has no effect if there is some data available.

The `read()` function reads data previously written to a file. If any portion of a regular file prior to the end-of-file has not been written, `read()` shall return bytes with value 0. For example, `lseek()` allows the file offset to be set beyond the end of existing data in the file. If data is later written at this point, subsequent reads in the gap between the previous end of data and the newly written data shall return bytes with value 0 until data is written into the gap.

Upon successful completion, where `nbyte` is greater than 0, `read()` shall mark for update the last data access timestamp of the file, and shall return the number of bytes read. This number shall never be greater than `nbyte`. The value returned may be less than `nbyte` if the number of bytes left in the file is less than `nbyte`, if the `read()` request was interrupted by a signal, or if the file is a pipe or FIFO or special file and has fewer than `nbyte` bytes immediately available for reading. For example, a `read()` from a file associated with a terminal may return one typed line of data.

If a `read()` is interrupted by a signal before it reads any data, it shall return -1 with `errno` set to [EINTR].

If a `read()` is interrupted by a signal after it has successfully read some data, it shall return the number of bytes read.

For regular files, no data transfer shall occur past the offset maximum established in the open file description associated with `fildes`.

If `fildes` refers to a socket, `read()` shall be equivalent to `recv()` with no flags set.

[SIO] If the O\_DSYNC and O\_RSYNC bits have been set, read I/O operations on the file descriptor shall complete as defined by synchronized I/O data integrity completion. If the O\_SYNC and O\_RSYNC bits have been set, read I/O operations on the file descriptor shall complete as defined by synchronized I/O file integrity completion.

[SHM] If `fildes` refers to a shared memory object, the result of the `read()` function is unspecified.

[TYM] If `fildes` refers to a typed memory object, the result of the `read()` function is unspecified.

The `pread()` function shall be equivalent to `read()`, except that it shall read from a given position in the file without changing the file offset. The first three arguments to `pread()` are the same as `read()` with the addition of a fourth argument `offset` for the desired position inside the file. An attempt to perform a `pread()` on a file that is incapable of seeking shall result in an error.

## RETURN VALUE

Upon successful completion, these functions shall return a non-negative integer indicating the number of bytes actually read. Otherwise, the functions shall return -1 and set `errno` to indicate the error.

## ERRORS

These functions shall fail if:

### [EAGAIN]

The file is neither a pipe, nor a FIFO, nor a socket, the `O_NONBLOCK` flag is set for the file descriptor, and the thread would be delayed in the read operation.

### [EBADF]

The `fildes` argument is not a valid file descriptor open for reading.

### [EINTR]

The read operation was terminated due to the receipt of a signal, and no data was transferred.

### [EIO]

The process is a member of a background process group attempting to read from its controlling terminal, and either the calling thread is blocking `SIGTTIN` or the process is ignoring `SIGTTIN` or the process group of the process is orphaned. This error may also be generated for implementation-defined reasons.

### [EISDIR]

[XSI] The `fildes` argument refers to a directory and the implementation does not allow the directory to be read using `read()` or `pread()`. The `readdir()` function should be used instead.

### [EOVERFLOW]

The file is a regular file, `nbyte` is greater than 0, the starting position is before the end-of-file, and the starting position is greater than or equal to the offset maximum established in the open file description associated with `fildes`.

The `pread()` function shall fail if:

**[EINVAL]**

The file is a regular file or block special file, and the `offset` argument is negative. The file offset shall remain unchanged.

**[ESPIPE]**

The file is incapable of seeking.

The `read()` function shall fail if:

**[EAGAIN]**

The file is a pipe or FIFO, the `O_NONBLOCK` flag is set for the file descriptor, and the thread would be delayed in the read operation.

**[EAGAIN] or [EWOULDBLOCK]**

The file is a socket, the `O_NONBLOCK` flag is set for the file descriptor, and the thread would be delayed in the read operation.

**[ECONNRESET]**

A read was attempted on a socket and the connection was forcibly closed by its peer.

**[ENOTCONN]**

A read was attempted on a socket that is not connected.

**[ETIMEDOUT]**

A read was attempted on a socket and a transmission timeout occurred.

These functions may fail if:

**[EIO]**

A physical I/O error has occurred.

**[ENOBUFS]**

Insufficient resources were available in the system to perform the operation.

**[ENOMEM]**

Insufficient memory was available to fulfill the request.

**[ENXIO]**

A request was made of a nonexistent device, or the request was outside the capabilities of the device.

## EXAMPLES

### Reading Data into a Buffer

The following example reads data from the file associated with the file descriptor `fd` into the buffer pointed to by `buf`.

```
#include <sys/types.h>
#include <unistd.h>
...
char buf[20];
size_t nbytes;
ssize_t bytes_read;
int fd;
...
nbytes = sizeof(buf);
bytes_read = read(fd, buf, nbytes);
...
```

## APPLICATION USAGE

None.

## RATIONALE

This volume of POSIX.1-2024 does not specify the value of the file offset after an error is returned; there are too many cases. For programming errors, such as [EBADF], the concept is meaningless since no file is involved. For errors that are detected immediately, such as [EAGAIN], clearly the offset should not change. After an interrupt or hardware error, however, an updated value would be very useful and is the behavior of many implementations.

Note that a `read()` of zero bytes does not modify the last data access timestamp. A `read()` that requests more than zero bytes, but returns zero, is required to modify the last data access timestamp.

Implementations are allowed, but not required, to perform error checking for `read()` requests of zero bytes.

## Input and Output

The use of I/O with large byte counts has always presented problems. Ideas such as `lread()` and `lwrite()` (using and returning `long`s) were considered at one time. The current solution is to use abstract types on the ISO C standard function to `read()` and `write()`. The abstract types can be declared so that existing functions work, but can also be declared so that larger types can be represented in future implementations. It is presumed that whatever constraints limit the maximum range of `size_t` also limit portable I/O requests to the same range. This volume of POSIX.1-2024 also limits the range further by requiring that the byte count be limited so that a signed return value remains meaningful. Since

the return type is also a (signed) abstract type, the byte count can be defined by the implementation to be larger than an `int` can hold.

The standard developers considered adding atomicity requirements to a pipe or FIFO, but recognized that due to the nature of pipes and FIFOs there could be no guarantee of atomicity of reads of {PIPE\_BUF} or any other size that would be an aid to applications portability.

This volume of POSIX.1-2024 requires that no action be taken for `read()` or `write()` when `nbyte` is zero. This is not intended to take precedence over detection of errors (such as invalid buffer pointers or file descriptors). This is consistent with the rest of this volume of POSIX.1-2024, but the phrasing here could be misread to require detection of the zero case before any other errors. A value of zero is to be considered a correct value, for which the semantics are a no-op.

I/O is intended to be atomic to ordinary files and pipes and FIFOs. Atomic means that all the bytes from a single operation that started out together end up together, without interleaving from other I/O operations. It is a known attribute of terminals that this is not honored, and terminals are explicitly (and implicitly permanently) excepted, making the behavior unspecified. The behavior for other device types is also left unspecified, but the wording is intended to imply that future standards might choose to specify atomicity (or not).

There were recommendations to add format parameters to `read()` and `write()` in order to handle networked transfers among heterogeneous file system and base hardware types. Such a facility may be required for support by the OSI presentation of layer services. However, it was determined that this should correspond with similar C-language facilities, and that is beyond the scope of this volume of POSIX.1-2024. The concept was suggested to the developers of the ISO C standard for their consideration as a possible area for future work.

In 4.3 BSD, a `read()` or `write()` that is interrupted by a signal before transferring any data does not by default return an [EINTR] error, but is restarted. In 4.2 BSD, 4.3 BSD, and the Eighth Edition, there is an additional function, `select()`, whose purpose is to pause until specified activity (data to read, space to write, and so on) is detected on specified file descriptors. It is common in applications written for those systems for `select()` to be used before `read()` in situations (such as keyboard input) where interruption of I/O due to a signal is desired.

The issue of which files or file types are interruptible is considered an implementation design issue. This is often affected primarily by hardware and reliability issues.

There are no references to actions taken following an "unrecoverable error". It is considered beyond the scope of this volume of POSIX.1-2024 to describe what

happens in the case of hardware errors.

Earlier versions of this standard allowed two very different behaviors with regard to the handling of interrupts. In order to minimize the resulting confusion, it was decided that POSIX.1-2024 should support only one of these behaviors. Historical practice on AT&T-derived systems was to have `read()` and `write()` return -1 and set `errno` to [EINTR] when interrupted after some, but not all, of the data requested had been transferred. However, the US Department of Commerce FIPS 151-1 and FIPS 151-2 require the historical BSD behavior, in which `read()` and `write()` return the number of bytes actually transferred before the interrupt. If -1 is returned when any data is transferred, it is difficult to recover from the error on a seekable device and impossible on a non-seekable device. Most new implementations support this behavior. The behavior required by POSIX.1-2024 is to return the number of bytes transferred.

POSIX.1-2024 does not specify when an implementation that buffers `read()`'s actually moves the data into the user-supplied buffer, so an implementation may choose to do this at the latest possible moment. Therefore, an interrupt arriving earlier may not cause `read()` to return a partial byte count, but rather to return -1 and set `errno` to [EINTR].

Consideration was also given to combining the two previous options, and setting `errno` to [EINTR] while returning a short count. However, not only is there no existing practice that implements this, it is also contradictory to the idea that when `errno` is set, the function responsible shall return -1.

This volume of POSIX.1-2024 intentionally does not specify any `pread()` errors related to pipes, FIFOs, and sockets other than [ESPIPE].

## FUTURE DIRECTIONS

None.

## SEE ALSO

`fcntl()`, `lseek()`, `open()`, `pipe()`, `readv()`

XBD 11. General Terminal Interface, `<sys/uio.h>`, `<unistd.h>`

## CHANGE HISTORY

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

## Issue 5

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

Large File Summit extensions are added.

The `pread()` function is added.

## Issue 6

The DESCRIPTION and ERRORS sections are updated so that references to STREAMS are marked as part of the XSI STREAMS Option Group.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION now states that if `read()` is interrupted by a signal after it has successfully read some data, it returns the number of bytes read. In Issue 3, it was optional whether `read()` returned the number of bytes read, or whether it returned -1 with `errno` set to [EINTR]. This is a FIPS requirement.
- In the DESCRIPTION, text is added to indicate that for regular files, no data transfer occurs past the offset maximum established in the open file description associated with `fildes`. This change is to support large files.
- The [EOVERFLOW] mandatory error condition is added.
- The [ENXIO] optional error condition is added.

Text referring to sockets is added to the DESCRIPTION.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The effect of reading zero bytes is clarified.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that `read()` results are unspecified for typed memory objects.

New RATIONALE is added to explain the atomicity requirements for input and output operations.

The following error conditions are added for operations on sockets: [EAGAIN], [ECONNRESET], [ENOTCONN], and [ETIMEDOUT].

The [EIO] error is made optional.

The following error conditions are added for operations on sockets: [ENOBUFS] and [ENOMEM].

The `readv()` function is split out into a separate reference page.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/108 is applied, updating the [EAGAIN] error in the ERRORS section from "the process would be delayed" to "the thread would be delayed".

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/109 is applied, making an editorial correction in the RATIONALE section.

## Issue 7

The `pread()` function is moved from the XSI option to the Base.

Functionality relating to the XSI STREAMS option is marked obsolescent.

Changes are made related to support for finegrained timestamps.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0480 [218], XSH/TC1-2008/0481 [79], XSH/TC1-2008/0482 [218], XSH/TC1-2008/0483 [218], XSH/TC1-2008/0484 [218], and XSH/TC1-2008/0485 [218,428] are applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0302 [710] and XSH/TC2-2008/0303 [676,710] are applied.

## Issue 8

Austin Group Defect 1330 is applied, removing obsolescent interfaces.

---

## 1.186. realloc, reallocarray — memory reallocators

---

### SYNOPSIS

```
#include <stdlib.h>

void *realloc(void *ptr, size_t size);

[CX] void *reallocarray(void *ptr, size_t nelem, size_t elsize);
```

### DESCRIPTION

For `realloc()`: [CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `realloc()` function shall deallocate the old object pointed to by `ptr` and return a pointer to a new object that has the size specified by `size`. The contents of the new object shall be the same as that of the old object prior to deallocation, up to the lesser of the new and old sizes. Any bytes in the new object beyond the size of the old object have indeterminate values.

[CX] The `reallocarray()` function shall be equivalent to the call `realloc(ptr, nelem * elsize)` except that overflow in the multiplication shall be an error.

If `ptr` is a null pointer, `realloc()` [CX] or `reallocarray()` shall be equivalent to `malloc()` for the specified size. Otherwise, if `ptr` does not match a pointer earlier returned by `aligned_alloc()`, `calloc()`, `malloc()`, [ADV] `posix_memalign()`, `realloc()`, [CX] `reallocarray()`, or a function in POSIX.1-2024 that allocates memory as if by `malloc()`, or if the space has been deallocated by a call to `free()`, [CX] `reallocarray()`, or `realloc()`, the behavior is undefined.

If `size` is non-zero and memory for the new object is not allocated, the old object shall not be deallocated.

The order and contiguity of storage allocated by successive calls to `realloc()` [CX] or `reallocarray()` is unspecified. The pointer returned if the allocation succeeds shall be suitably aligned so that it may be assigned to a pointer to any type of object with a fundamental alignment requirement and then used to access such an object in the space allocated (until the space is explicitly freed or

reallocated). Each such allocation shall yield a pointer to an object disjoint from any other object. The pointer returned shall point to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer shall be returned.

For purposes of determining the existence of a data race, `realloc()` [CX] and `reallocarray()` shall each behave as though it accessed only memory locations accessible through its argument and not other static duration storage. The function may, however, visibly modify the storage that it allocates. Calls to `aligned_alloc()`, `calloc()`, `free()`, `malloc()`, [ADV] `posix_memalign()`, [CX] `reallocarray()`, and `realloc()` that allocate or deallocate a particular region of memory shall occur in a single total order (see 4.15.1 Memory Ordering), and each such deallocation call shall synchronize with the next allocation (if any) in this order.

## RETURN VALUE

Upon successful completion, `realloc()` [CX] and `reallocarray()` shall return a pointer to the new object (which can have the same value as a pointer to the old object), or a null pointer if the new object has not been allocated.

[OB] If `size` is 0, [OB CX] or either `nelem` or `elsize` is 0, [OB] either:

If there is not enough available memory, `realloc()` [CX] and `reallocarray()` shall return a null pointer [CX] and set `errno` to [ENOMEM].

## ERRORS

The `realloc()` [CX] and `reallocarray()` functions shall fail if:

### [ENOMEM]

[CX] Insufficient memory is available.

[CX] The `reallocarray()` function shall fail if:

### [ENOMEM]

The calculation `nelem * elsize` would overflow.

The `realloc()` [CX] and `reallocarray()` functions may fail if:

### [EINVAL]

[CX] The requested allocation size is 0 and the implementation does not support 0 sized allocations.

## EXAMPLES

None.

## APPLICATION USAGE

The ISO C standard makes it implementation-defined whether a call to `realloc(p, 0)` frees the space pointed to by `p` if it returns a null pointer because memory for the new object was not allocated. POSIX.1 instead requires that implementations set `errno` if a null pointer is returned and the space has not been freed, and POSIX applications should only free the space if `errno` was changed.

## RATIONALE

See the RATIONALE for `malloc()`.

## FUTURE DIRECTIONS

The ISO C standard states that invoking `realloc()` with a `size` argument equal to zero is an obsolescent feature. This feature may be removed in a future version of this standard.

## SEE ALSO

- `aligned_alloc()`
- `calloc()`
- `free()`
- `malloc()`
- `<stdlib.h>`

## CHANGE HISTORY

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

## Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE section, if there is not enough available memory, the setting of `errno` to [ENOMEM] is added.
- The [ENOMEM] error condition is added.

## Issue 7

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0495 [400], XSH/TC1-2008/0496 [400], XSH/TC1-2008/0497 [400], and XSH/TC1-2008/0498 [400] are applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0309 [526] and XSH/TC2-2008/0310 [526,688] are applied.

## Issue 8

Austin Group Defect 374 is applied, adding the [EINVAL] error.

Austin Group Defect 1218 is applied, adding `reallocarray()`.

Austin Group Defect 1302 is applied, aligning the `realloc()` function with the ISO/IEC 9899:2018 standard.

Austin Group Defect 1387 is applied, changing the RATIONALE section.

---

## 1.187. `scanf`

---

### Synopsis

```
#include <stdio.h>

int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *s, const char *format, ...);
```

### Description

The `fscanf()` function shall read from the named input `stream`. The `scanf()` function shall read from the standard input stream `stdin`. The `sscanf()` function shall read from the string `s`. Each function reads bytes, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string `format` described below, and a set of `pointer` arguments indicating where the converted input should be stored. The result is undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments shall be evaluated but otherwise ignored.

Conversions can be applied to the  $n$ -th argument after the format in the argument list, rather than to the next unused argument. In this case, the conversion specifier character `%` (see below) is replaced by the sequence `"%n$"`, where  $n$  is a decimal integer in the range  $[1, \{NL_ARGMAX\}]$ . This feature provides for the definition of format strings that select arguments in an order appropriate to specific languages. In format strings containing the `"%n$"` form of conversion specifications, it is unspecified whether numbered arguments in the argument list can be referenced from the format string more than once.

The format can contain either form of a conversion specification—that is, `%` or `"%n$"`—but the two forms cannot be mixed within a single format string. The only exception to this is that `%%` or `%*` can be mixed with the `"%n$"` form. 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 pointers.

The `fscanf()` function in all its forms shall allow detection of a language-dependent radix character in the input string. The radix character is defined in the current locale (category `LC_NUMERIC`). In the `POSIX` locale, or in a locale where the radix character is not defined, the radix character shall default to a period ('.').

## Format String

The application shall ensure that the format is a character string, beginning and ending in its initial shift state, if any, composed of zero or more directives. Each directive is composed of one of the following: one or more white-space bytes; an ordinary character (neither '%' nor a white-space byte); or a conversion specification. Each conversion specification is introduced by the character '%' or the character sequence "%n\$", after which the following appear in sequence:

- An optional assignment-suppressing character '\*'.
- An optional non-zero decimal integer that specifies the maximum field width.
- An optional assignment-allocation character 'm'.
- An optional length modifier that specifies the size of the receiving object.
- A conversion specifier character that specifies the type of conversion to be applied. The valid conversion specifiers are described below.

## Execution Process

The `fscanf()` functions shall execute each directive of the format in turn. When all directives have been executed, or if a directive fails (as detailed below), the function shall return. Failures are described as input failures (due to the unavailability of input bytes) or matching failures (due to inappropriate input).

A directive composed of one or more white-space bytes shall be executed by reading input up to the first non-white-space byte, which shall remain unread, or until no more bytes can be read. The directive shall never fail.

A directive that is an ordinary character shall be executed as follows: the next byte shall be read from the input and compared with the byte that comprises the directive; if the comparison shows that they are not equivalent, the directive shall fail, and the differing and subsequent bytes shall remain unread. Similarly, if end-of-file, an encoding error, or a read error prevents a character from being read, the directive shall fail.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each conversion character. A conversion specification shall be executed in the following steps.

Input white-space bytes shall be skipped, unless the conversion specification includes a [, c, C, or n conversion specifier.

An item shall be read from the input, unless the conversion specification includes an n conversion specifier. An input item shall be 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 specifier) which is an initial

subsequence of a matching sequence. The first byte, if any, after the input item shall remain unread. If the length of the input item is 0, the execution of the conversion specification shall fail; 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 specifier, the input item (or, in the case of a %n conversion specification, the count of input bytes) shall be 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 shall be placed in the object pointed to by the first argument following the format argument that has not already received a conversion result if the conversion specification is introduced by %, or in the n-th argument if introduced by the character sequence "%n\$". If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

## Assignment Allocation

The c, s, and [ conversion specifiers shall accept an optional assignment-allocation character 'm', which shall cause a memory buffer to be allocated to hold the conversion results. If the conversion specifier is s or [, the allocated buffer shall include space for a terminating null character (or wide character). In such a case, the argument corresponding to the conversion specifier should be a reference to a pointer variable that will receive a pointer to the allocated buffer. The system shall allocate a buffer as if `malloc()` had been called. The application shall be responsible for freeing the memory after usage. If there is insufficient memory to allocate a buffer, the function shall set `errno` to [ENOMEM] and a conversion error shall result. If the function returns EOF, any memory successfully allocated for parameters using assignment-allocation character 'm' by this call shall be freed before the function returns.

## Length Modifiers

The length modifiers and their meanings are:

### **hh**

Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to **signed char** or **unsigned char**.

### **h**

Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to **short** or **unsigned short**.

## I (ell)

Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to **long** or **unsigned long**; that a following a, A, e, E, f, F, g, or G conversion specifier applies to an argument with type pointer to **double**; or that a following c, s, or [ conversion specifier applies to an argument with type pointer to **wchar\_t**. If the 'm' assignment-allocation character is specified, the conversion applies to an argument with the type pointer to a pointer to **wchar\_t**.

## II (ell-ell)

Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to **long long** or **unsigned long long**.

## j

Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to **intmax\_t** or **uintmax\_t**.

## z

Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to **size\_t** or the corresponding signed integer type.

## t

Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to **ptrdiff\_t** or the corresponding **unsigned** type.

## L

Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to an argument with type pointer to **long double**.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

# Conversion Specifiers

The following conversion specifiers are valid:

## d

Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of **strtol()** with the value 10 for the base argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to **int**.

## i

Matches an optionally signed integer, whose format is the same as expected for the subject sequence of **strtol()** with 0 for the base argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to **int**.

**o**

Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of `strtoul()` with the value 8 for the base argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to **unsigned**.

**u**

Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of `strtoul()` with the value 10 for the base argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to **unsigned**.

**x**

Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of `strtoul()` with the value 16 for the base argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to **unsigned**.

**a, e, f, g**

Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected for the subject sequence of `strtod()`. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to **float**.

If the `fprintf()` family of functions generates character string representations for infinity and NaN (a symbolic entity encoded in floating-point format) to support IEEE Std 754-1985, the `fscanf()` family of functions shall recognize them as input.

**s**

Matches a sequence of bytes that are not white-space bytes. If the 'm' assignment-allocation character is not specified, the application shall ensure that the corresponding argument is a pointer to the initial byte of an array of **char**, **signed char**, or **unsigned char** large enough to accept the sequence and a terminating null character code, which shall be added automatically. Otherwise, the application shall ensure that the corresponding argument is a pointer to a pointer to a **char**.

If an l (ell) qualifier is present, the input is a sequence of characters that begins in the initial shift state. Each character shall be converted to a wide character as if by a call to the `mbrtowc()` function, with the conversion state described by an **mbstate\_t** object initialized to zero before the first character is converted. If the 'm' assignment-allocation character is not specified, the application shall ensure that the corresponding argument is a pointer to an array of **wchar\_t** large enough to accept the sequence and the terminating null wide character, which shall be added automatically. Otherwise, the application shall ensure that the corresponding argument is a pointer to a pointer to a **wchar\_t**.

## [

Matches a non-empty sequence of bytes from a set of expected bytes (the scanset). The normal skip over white-space bytes shall be suppressed in this case. If the 'm' assignment-allocation character is not specified, the application shall ensure that the corresponding argument is a pointer to the initial byte of an array of **char**, **signed char**, or **unsigned char** large enough to accept the sequence and a terminating null byte, which shall be added automatically. Otherwise, the application shall ensure that the corresponding argument is a pointer to a pointer to a **char**.

If an l (ell) qualifier is present, the input is a sequence of characters that begins in the initial shift state. Each character in the sequence shall be converted to a wide character as if by a call to the **mbrtowc()** function, with the conversion state described by an **mbstate\_t** object initialized to zero before the first character is converted. If the 'm' assignment-allocation character is not specified, the application shall ensure that the corresponding argument is a pointer to an array of **wchar\_t** large enough to accept the sequence and the terminating null wide character, which shall be added automatically. Otherwise, the application shall ensure that the corresponding argument is a pointer to a pointer to a **wchar\_t**.

The conversion specification includes all subsequent bytes in the format string up to and including the matching right-square-bracket (']'). The bytes between the square brackets (the scanlist) comprise the scanset, unless the byte after the left-square-bracket is a circumflex (^), in which case the scanset contains all bytes that do not appear in the scanlist between the circumflex and the right-square-bracket. If the conversion specification begins with "[]" or "[^]", the right-square-bracket is included in the scanlist and the next right-square-bracket is the matching right-square-bracket that ends the conversion specification; otherwise, the first right-square-bracket is the one that ends the conversion specification. If a '-' is in the scanlist and is not the first character, nor the second where the first character is a '^', nor the last character, the behavior is implementation-defined.

## c

Matches a sequence of bytes of the number specified by the field width (1 if no field width is present in the conversion specification). No null byte is added. The normal skip over white-space bytes shall be suppressed in this case. If the 'm' assignment-allocation character is not specified, the application shall ensure that the corresponding argument is a pointer to the initial byte of an array of **char**, **signed char**, or **unsigned char** large enough to accept the sequence. Otherwise, the application shall ensure that the corresponding argument is a pointer to a pointer to a **char**.

If an l (ell) qualifier is present, the input shall be a sequence of characters that begins in the initial shift state. Each character in the sequence is converted to a wide character as if by a call to the **mbrtowc()** function, with the conversion state

described by an **mbstate\_t** object initialized to zero before the first character is converted. No null wide character is added. If the 'm' assignment-allocation character is not specified, the application shall ensure that the corresponding argument is a pointer to an array of **wchar\_t** large enough to accept the resulting sequence of wide characters. Otherwise, the application shall ensure that the corresponding argument is a pointer to a pointer to a **wchar\_t**.

### **p**

Matches an implementation-defined set of sequences, which shall be the same as the set of sequences that is produced by the %p conversion specification of the corresponding **fprintf()** functions. The application shall ensure that the corresponding argument is a pointer to a pointer to **void**. The interpretation of the input item is implementation-defined. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise, the behavior of the %p conversion specification is undefined.

### **n**

No input is consumed. The application shall ensure that the corresponding argument is a pointer to the integer into which shall be written the number of bytes read from the input so far by this call to the **fscanf()** functions. Execution of a %n conversion specification shall not increment the assignment count returned at the completion of execution of the function. No argument shall be converted, but one shall be consumed. If the conversion specification includes an assignment-suppressing character or a field width, the behavior is undefined.

### **C**

Equivalent to lc.

### **S**

Equivalent to ls.

### **%**

Matches a single '%' character; no conversion or assignment occurs. The complete conversion specification shall be %%.

If a conversion specification is invalid, the behavior is undefined.

The conversion specifiers A, E, F, G, and X are also valid and shall be equivalent to a, e, f, g, and x, respectively.

## Termination Conditions

If end-of-file is encountered during input, conversion shall be terminated. If end-of-file occurs before any bytes matching the current conversion specification (except for %n) have been read (other than leading white-space bytes, where permitted), execution of the current conversion specification shall terminate with an input

failure. Otherwise, unless execution of the current conversion specification is terminated with a matching failure, execution of the following conversion specification (if any) shall be terminated with an input failure.

Reaching the end of the string in `sscanf()` shall be equivalent to encountering end-of-file for `fscanf()`.

If conversion terminates on a conflicting input, the offending input is left unread in the input. Any trailing white-space bytes (including newline characters) shall be left unread unless matched by a conversion specification. The success of literal matches and suppressed assignments is only directly determinable via the `%n` conversion specification.

The `fscanf()` and `scanf()` functions may mark the last data access timestamp of the file associated with `stream` for update. The last data access timestamp shall be marked for update by the first successful execution of `fgetc()`, `fgets()`, `fread()`, `getc()`, `getchar()`, `getdelim()`, `getline()`, `fscanf()`, or `scanf()` using `stream` that returns data not supplied by a prior call to `ungetc()`.

## Return Value

Upon successful completion, these functions shall return the number of successfully matched and assigned input items; this number can be zero in the event of an early matching failure. If the input ends before the first matching failure or conversion, EOF shall be returned. If a read error occurs, the error indicator for the stream shall be set, EOF shall be returned, and `errno` shall be set to indicate the error.

## Errors

The functions shall fail if:

- **EAGAIN** - The `O_NONBLOCK` flag is set for the file descriptor underlying `stream` and the thread would be delayed in the `fscanf()` operation.
- **EBADF** - The file descriptor underlying `stream` is not a valid file descriptor opened for reading.
- **EINTR** - The read operation was terminated due to the receipt of a signal, and no data was transferred.
- **EIO** - A physical I/O error has occurred.
- **ENOMEM** - Insufficient storage space is available.

The `fscanf()` and `scanf()` functions may fail if:

- **OVERFLOW** - The file is a regular file and an attempt was made to read at or beyond the offset maximum associated with the corresponding stream.

The `sscanf()` function shall fail if:

- **ENAMETOOLONG** - The length of a component of a pathname exceeds `{NAME_MAX}`.

## Examples

```
#include <stdio.h>

int main() {
    int i, j;
    float x, y;
    char s[80];

    /* Basic usage */
    scanf("%d %f", &i, &x);

    /* Reading into multiple variables */
    scanf("%d %f %s", &j, &y, s);

    /* Using assignment suppression */
    scanf("%*d %d", &i); /* Skip first integer, read second */

    /* Using field width */
    scanf("%4d %4d", &i, &j); /* Read two 4-digit integers */

    return 0;
}
```

## Application Usage

Programmers are encouraged to use `fgets()` and `sscanf()` instead of `scanf()` to avoid issues with buffer overflows and to handle input errors more gracefully.

## Rationale

The `scanf()` family of functions provides formatted input capabilities that complement the `printf()` family of formatted output functions. The assignment-

allocation feature (the 'm' character) was added to simplify handling of variable-length string input.

## Future Directions

None.

## See Also

- [fgets\(\)](#)
- [fprintf\(\)](#)
- [getc\(\)](#)
- [getdelim\(\)](#)
- [getline\(\)](#)
- [malloc\(\)](#)
- [setlocale\(\)](#)
- [strtod\(\)](#)
- [strtol\(\)](#)
- [strtoul\(\)](#)

## Copyright

Portions of this text are reprinted and reproduced in electronic form from the IEEE Std 1003.1-2024, Standard for Information Technology -- Portable Operating System Interface (POSIX), The Open Group Base Specifications Issue 7, Copyright (C) 2024 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group. In the event of any discrepancy between this version and the original IEEE and The Open Group Standard, the original IEEE and The Open Group Standard is the referee document. The original Standard can be obtained online at <http://www.opengroup.org/unix/online.html>.

Any typographical or formatting errors that appear in this page are most likely the result of an error in transcribing the information from the original document and are not part of the standard itself.

## 1.188. `sched_get_priority_max`, `sched_get_priority_min`

---

### SYNOPSIS

```
[PS|TPS] #include <sched.h>

int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
```

### DESCRIPTION

The `sched_get_priority_max()` and `sched_get_priority_min()` functions shall return the appropriate maximum or minimum, respectively, for the scheduling policy specified by `policy`.

The value of `policy` shall be one of the scheduling policy values defined in `<sched.h>`.

### RETURN VALUE

If successful, the `sched_get_priority_max()` and `sched_get_priority_min()` functions shall return the appropriate maximum or minimum values, respectively. If unsuccessful, they shall return a value of -1 and set `errno` to indicate the error.

### ERRORS

The `sched_get_priority_max()` and `sched_get_priority_min()` functions shall fail if:

- `EINVAL`
- The value of the `policy` parameter does not represent a defined scheduling policy.

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `sched_getparam()`
- `sched_setparam()`
- `sched_getscheduler()`
- `sched_rr_get_interval()`
- `sched_setscheduler()`

XBD `<sched.h>`

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

### Issue 6

These functions are marked as part of the Process Scheduling option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Process Scheduling option.

The [ESRCH] error condition has been removed since these functions do not take a `pid` argument.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/52 is applied, changing the PS margin code in the SYNOPSIS to PS|TPS.

## 1.189. `sched_get_priority_max`, `sched_get_priority_min`

---

### SYNOPSIS

```
#include <sched.h>

int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
```

### DESCRIPTION

The `sched_get_priority_max()` and `sched_get_priority_min()` functions shall return the appropriate maximum or minimum, respectively, for the scheduling policy specified by `policy`.

The value of `policy` shall be one of the scheduling policy values defined in `<sched.h>`.

### RETURN VALUE

If successful, the `sched_get_priority_max()` and `sched_get_priority_min()` functions shall return the appropriate maximum or minimum values, respectively. If unsuccessful, they shall return a value of -1 and set `errno` to indicate the error.

### ERRORS

The `sched_get_priority_max()` and `sched_get_priority_min()` functions shall fail if:

- **EINVAL**
- The value of the `policy` parameter does not represent a defined scheduling policy.

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `sched_getparam()`
- `sched_setparam()`
- `sched_getscheduler()`
- `sched_rr_get_interval()`
- `sched_setscheduler()`

XBD `<sched.h>`

## CHANGE HISTORY

### First released in Issue 5

Included for alignment with the POSIX Realtime Extension.

### Issue 6

These functions are marked as part of the Process Scheduling option.

The `[ENOSYS]` error condition has been removed as stubs need not be provided if an implementation does not support the Process Scheduling option.

The **[ESRCH]** error condition has been removed since these functions do not take a **pid** argument.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/52 is applied, changing the PS margin code in the SYNOPSIS to PS|TPS.

---

## 1.190. sched\_rr\_get\_interval — get execution time limits (REALTIME)

---

### SYNOPSIS

```
#include <sched.h>

int sched_rr_get_interval(pid_t pid, struct timespec *interval);
```

### DESCRIPTION

The `sched_rr_get_interval()` function shall update the `timespec` structure referenced by the `interval` argument to contain the current execution time limit (that is, time quantum) for the process specified by `pid`. If `pid` is zero, the current execution time limit for the calling process shall be returned.

### RETURN VALUE

If successful, the `sched_rr_get_interval()` function shall return zero. Otherwise, it shall return a value of -1 and set `errno` to indicate the error.

### ERRORS

The `sched_rr_get_interval()` function shall fail if:

- **[ESRCH]** - No process can be found corresponding to that specified by `pid`.

### EXAMPLES

None.

### APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `sched_getparam()`
- `sched_get_priority_max()`
- `sched_getscheduler()`
- `sched_setparam()`
- `sched_setscheduler()`

XBD `<sched.h>`

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

### Issue 6

The `sched_rr_get_interval()` function is marked as part of the Process Scheduling option.

The `[ENOSYS]` error condition has been removed as stubs need not be provided if an implementation does not support the Process Scheduling option.

IEEE Std 1003.1-2001/Cor 1-2002, XSH/TC1/D6/53 is applied, changing the PS margin code in the SYNOPSIS to PS|TPS.

---

## 1.191. `sem_close` — close a named semaphore

---

### SYNOPSIS

```
#include <semaphore.h>

int sem_close(sem_t *sem);
```

### DESCRIPTION

The `sem_close()` function shall indicate that the calling process is finished using the named semaphore indicated by `sem`. The effects of calling `sem_close()` for an unnamed semaphore (one created by `sem_init()`) are undefined. The `sem_close()` function shall deallocate (that is, make available for reuse by a subsequent `sem_open()` by this process) any system resources allocated by the system for use by this process for this semaphore. If the semaphore indicated by `sem` is implemented using a file descriptor, the file descriptor shall be closed. The effect of subsequent use of the semaphore indicated by `sem` by this process is undefined. If any threads in the calling process are currently blocked on the semaphore, the behavior is undefined. If the semaphore has not been removed with a successful call to `sem_unlink()`, then `sem_close()` has no effect on the state of the semaphore. If the `sem_unlink()` function has been successfully invoked for `name` after the most recent call to `sem_open()` with `O_CREAT` for this semaphore, then when all processes that have opened the semaphore close all semaphore handles to it, the semaphore is no longer accessible.

### RETURN VALUE

Upon successful completion, a value of zero shall be returned. Otherwise, a value of -1 shall be returned and `errno` set to indicate the error.

### ERRORS

The `sem_close()` function may fail if:

- `[EINVAL]`

The `sem` argument is not a valid semaphore descriptor.

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

`semctl()` , `semget()` , `semop()` , `sem_init()` , `sem_open()` ,  
`sem_unlink()`

XBD `<semaphore.h>`

## CHANGE HISTORY

### Issue 6

The `sem_close()` function is marked as part of the Semaphores option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Semaphores option.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/113 is applied, updating the ERRORS section so that the [EINVAL] error becomes optional.

### Issue 7

The `sem_close()` function is moved from the Semaphores option to the Base.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0523 [37] is applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0317 [870] is applied.

## Issue 8

Austin Group Defect 368 is applied, adding a requirement that if `sem` is implemented using a file descriptor, `sem_close()` closes the file descriptor.

Austin Group Defect 1324 is applied, clarifying the circumstances under which an unlinked semaphore is no longer accessible.

---

## 1.192. `sem_destroy` — destroy an unnamed semaphore

---

### SYNOPSIS

```
#include <semaphore.h>

int sem_destroy(sem_t *sem);
```

### DESCRIPTION

The `sem_destroy()` function shall destroy the unnamed semaphore indicated by `sem`. If an unnamed semaphore is implemented using a file descriptor, the file descriptor shall be closed. Only a semaphore that was created using `sem_init()` can be destroyed using `sem_destroy()`; the effect of calling `sem_destroy()` with a named semaphore is undefined. The effect of subsequent use of the semaphore `sem` is undefined until `sem` is reinitialized by another call to `sem_init()`.

It is safe to destroy an initialized semaphore upon which no threads are currently blocked. The effect of destroying a semaphore upon which other threads are currently blocked is undefined.

### RETURN VALUE

Upon successful completion, a value of zero shall be returned. Otherwise, a value of -1 shall be returned and `errno` set to indicate the error.

### ERRORS

The `sem_destroy()` function may fail if:

- **[EINVAL]**  
The `sem` argument is not a valid semaphore.
- **[EBUSY]**  
There are currently processes blocked on the semaphore.

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `semctl()`
- `semget()`
- `semop()`
- `sem_init()`
- `sem_open()`

XBD `<semaphore.h>`

## CHANGE HISTORY

### First released in Issue 5

Included for alignment with the POSIX Realtime Extension.

### Issue 6

- The `sem_destroy()` function is marked as part of the Semaphores option.
- The `[ENOSYS]` error condition has been removed as stubs need not be provided if an implementation does not support the Semaphores option.

- IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/114 is applied, updating the ERRORS section so that the `[EINVAL]` error becomes optional.

## Issue 7

- The `sem_destroy()` function is moved from the Semaphores option to the Base.
- POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0524 [37] is applied.

## Issue 8

- Austin Group Defect 368 is applied, adding a requirement that if an unnamed semaphore is implemented using a file descriptor, `sem_destroy()` closes the file descriptor.
-

## 1.193. sem\_getvalue

---

### SYNOPSIS

```
#include <semaphore.h>

int sem_getvalue(sem_t *restrict sem, int *restrict sval);
```

### DESCRIPTION

The `sem_getvalue()` function shall update the location referenced by the `sval` argument to have the value of the semaphore referenced by `sem` without affecting the state of the semaphore. The updated value represents an actual semaphore value that occurred at some unspecified time during the call, but it need not be the actual value of the semaphore when it is returned to the calling process.

If `sem` is locked, then the object to which `sval` points shall either be set to zero or to a negative number whose absolute value represents the number of processes waiting for the semaphore at some unspecified time during the call.

### RETURN VALUE

Upon successful completion, the `sem_getvalue()` function shall return a value of zero. Otherwise, it shall return a value of -1 and set `errno` to indicate the error.

### ERRORS

The `sem_getvalue()` function may fail if:

- **[EINVAL]** - The `sem` argument does not refer to a valid semaphore.

---

*The following sections are informative.*

### EXAMPLES

None.

# APPLICATION USAGE

None.

# RATIONALE

None.

# FUTURE DIRECTIONS

None.

# SEE ALSO

- `semctl()`
- `semget()`
- `semop()`
- `sem_clockwait()`
- `sem_post()`
- `sem_trywait()`

XBD `<semaphore.h>`

# CHANGE HISTORY

## First released in Issue 5.

Included for alignment with the POSIX Realtime Extension.

## Issue 6

The `sem_getvalue()` function is marked as part of the Semaphores option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Semaphores option.

The `sem_timedwait()` function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

The **restrict** keyword is added to the `sem_getvalue()` prototype for alignment with the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/54 is applied.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/115 is applied, updating the ERRORS section so that the [EINVAL] error becomes optional.

## Issue 7

The `sem_getvalue()` function is moved from the Semaphores option to the Base.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0525 [37] is applied.

---

## 1.194. `sem_init` — initialize an unnamed semaphore

---

### SYNOPSIS

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned value);
```

### DESCRIPTION

The `sem_init()` function shall initialize the unnamed semaphore referred to by `sem`. The value of the initialized semaphore shall be `value`. Following a successful call to `sem_init()`, the semaphore can be used in subsequent calls to `sem_clockwait()`, `sem_destroy()`, `sem_post()`, `sem_timedwait()`, `sem_trywait()`, and `sem_wait()`. This semaphore shall remain usable until the semaphore is destroyed. An unnamed semaphore may be implemented using a file descriptor.

If the `pshared` argument has a non-zero value, then the semaphore is shared between processes; in this case, any process that can access the semaphore `sem` can use `sem` for performing `sem_clockwait()`, `sem_destroy()`, `sem_post()`, `sem_timedwait()`, `sem_trywait()`, and `sem_wait()` operations.

If the `pshared` argument is zero, then the semaphore is shared between threads of the process; any thread in this process can use `sem` for performing `sem_clockwait()`, `sem_destroy()`, `sem_post()`, `sem_timedwait()`, `sem_trywait()`, and `sem_wait()` operations.

See [2.9.9 Synchronization Object Copies and Alternative Mappings](#) for further requirements.

Attempting to initialize an already initialized semaphore results in undefined behavior.

### RETURN VALUE

Upon successful completion, the `sem_init()` function shall initialize the semaphore in `sem` and return 0. Otherwise, it shall return -1 and set `errno` to indicate the error.

# ERRORS

The `sem_init()` function shall fail if:

- **[EINVAL]**

The `value` argument exceeds `{SEM_VALUE_MAX}`.

- **[ENOSPC]**

A resource required to initialize the semaphore has been exhausted, or the limit on semaphores (`{SEM_NSEMS_MAX}`) has been reached.

- **[EPERM]**

The process lacks appropriate privileges to initialize the semaphore.

The `sem_init()` function may fail if:

- **[EMFILE]**

All file descriptors available to the process are currently open.

- **[ENFILE]**

The maximum allowable number of files is currently open in the system.

# EXAMPLES

None.

# APPLICATION USAGE

None.

# RATIONALE

None.

# FUTURE DIRECTIONS

None.

# SEE ALSO

- `sem_clockwait()`

- `sem_destroy()`
- `sem_post()`
- `sem_trywait()`
- `<semaphore.h>`

## CHANGE HISTORY

### First released in Issue 5

Included for alignment with the POSIX Realtime Extension.

### Issue 6

- The `sem_init()` function is marked as part of the Semaphores option.
- The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Semaphores option.
- The `sem_timedwait()` function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.
- IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/116 is applied, updating the DESCRIPTION to add the `sem_timedwait()` function for alignment with IEEE Std 1003.1d-1999.

### Issue 7

- SD5-XSH-ERN-176 is applied.
- The `sem_init()` function is moved from the Semaphores option to the Base.
- POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0526 [37] is applied.
- POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0318 [972] is applied.

### Issue 8

- Austin Group Defect 368 is applied, adding a statement that an unnamed semaphore may be implemented using a file descriptor and adding the [EMFILE] and [ENFILE] errors.
- Austin Group Defect 1216 is applied, adding `sem_clockwait()`.



## 1.195. `sem_open` — initialize and open a named semaphore

---

### SYNOPSIS

```
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag, ...);
```

### DESCRIPTION

The `sem_open()` function shall establish a connection between a named semaphore and a process. A named semaphore may be implemented using a file descriptor. Following a call to `sem_open()` with semaphore name `name`, the process may reference the semaphore associated with `name` using the semaphore handle returned from the call. This semaphore can be used in subsequent calls to `sem_clockwait()`, `sem_close()`, `sem_post()`, `sem_timedwait()`, `sem_trywait()`, and `sem_wait()`. The semaphore remains usable by this process until the semaphore is closed by a successful call to `sem_close()`, `_exit()`, or one of the `exec` functions.

The `oflag` argument controls whether the semaphore is created or merely accessed by the call to `sem_open()`. The following flag bits may be set in `oflag`:

#### O\_CREAT

This flag is used to create a semaphore if it does not already exist. If O\_CREAT is set and the semaphore already exists, then O\_CREAT has no effect, except as noted under O\_EXCL. Otherwise, `sem_open()` creates a named semaphore. The O\_CREAT flag requires a third and a fourth argument: `mode`, which is of type `mode_t`, and `value`, which is of type `unsigned`. The semaphore is created with an initial value of `value`. Valid initial values for semaphores are less than or equal to {SEM\_VALUE\_MAX}.

The user ID of the semaphore shall be set to the effective user ID of the process. The group ID of the semaphore shall be set to the effective group ID of the process; however, if the `name` argument is visible in the file system, the group ID may be set to the group ID of the containing directory. The permission bits of the semaphore are set to the value of the `mode` argument except those set in the file

mode creation mask of the process. When bits in `mode` other than the file permission bits are specified, the effect is unspecified.

After the semaphore named `name` has been created by `sem_open()` with the `O_CREAT` flag, other processes can connect to the semaphore by calling `sem_open()` with the same value of `name`.

## O\_EXCL

If `O_EXCL` and `O_CREAT` are set, `sem_open()` fails if the semaphore `name` exists. The check for the existence of the semaphore and the creation of the semaphore if it does not exist are atomic with respect to other processes executing `sem_open()` with `O_EXCL` and `O_CREAT` set. If `O_EXCL` is set and `O_CREAT` is not set, the effect is undefined.

If flags other than `O_CREAT` and `O_EXCL` are specified in the `oflag` parameter, the effect is unspecified.

The `name` argument points to a string naming a semaphore object. It is unspecified whether the name appears in the file system and is visible to functions that take pathnames as arguments. The `name` argument conforms to the construction rules for a pathname, except that the interpretation of `<slash>` characters other than the leading `<slash>` character in `name` is implementation-defined, and that the length limits for the `name` argument are implementation-defined and need not be the same as the pathname limits `{PATH_MAX}` and `{NAME_MAX}`. If `name` begins with the `<slash>` character, then processes calling `sem_open()` with the same value of `name` shall refer to the same semaphore object, as long as that name has not been removed. If `name` does not begin with the `<slash>` character, the effect is implementation-defined.

If a process makes multiple successful calls to `sem_open()` with the same value for `name`, there have been no intervening calls to `sem_unlink()` for `name`, and at least one open handle for this semaphore has not been closed with a `sem_close()` call, it is implementation-defined whether the same handle or a unique handle is returned for each such successful call.

References to copies of the semaphore produce undefined results.

## RETURN VALUE

Upon successful completion, the `sem_open()` function shall return the address of the semaphore. Otherwise, it shall return a value of `SEM_FAILED` and set `errno` to indicate the error. The symbol `SEM_FAILED` is defined in the `<semaphore.h>`

header. No successful return from `sem_open()` shall return the value `SEM_FAILED`.

## ERRORS

If any of the following conditions occur, the `sem_open()` function shall return `SEM_FAILED` and set `errno` to the corresponding value:

- **[EACCES]** The named semaphore exists and the permissions specified by `oflag` are denied, or the named semaphore does not exist and permission to create the named semaphore is denied.
- **[EEXIST]** `O_CREAT` and `O_EXCL` are set and the named semaphore already exists.
- **[EINTR]** The `sem_open()` operation was interrupted by a signal.
- **[EINVAL]** The `sem_open()` operation is not supported for the given name, or `O_CREAT` was specified in `oflag` and `value` was greater than `{SEM_VALUE_MAX}`.
- **[ENOENT]** `O_CREAT` is not set and the named semaphore does not exist.
- **[ENOMEM]** There is insufficient memory for the creation of the new named semaphore.
- **[ENOSPC]** There is insufficient space on a storage device for the creation of the new named semaphore.

If any of the following conditions occur, the `sem_open()` function may return `SEM_FAILED` and set `errno` to the corresponding value:

- **[EMFILE]** Too many semaphore descriptors or file descriptors are currently in use by this process.
- **[ENAMETOOLONG]** The length of the `name` argument exceeds `{_POSIX_PATH_MAX}` on systems that do not support the XSI option [XSI] or exceeds `{_XOPEN_PATH_MAX}` on XSI systems, or has a pathname component that is longer than `{_POSIX_NAME_MAX}` on systems that do not support the XSI option [XSI] or longer than `{_XOPEN_NAME_MAX}` on XSI systems.
- **[ENFILE]** Too many semaphore descriptors or file descriptors are currently open in the system.

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

Early drafts required an error return value of -1 with the type `sem_t *` for the `sem_open()` function, which is not guaranteed to be portable across implementations. The revised text provides the symbolic error code `SEM_FAILED` to eliminate the type conflict.

## FUTURE DIRECTIONS

A future version might require the `sem_open()` and `sem_unlink()` functions to have semantics similar to normal file system operations.

## SEE ALSO

`semctl()`, `semget()`, `semop()`, `sem_clockwait()`, `sem_close()`,  
`sem_post()`, `sem_trywait()`, `sem_unlink()`

XBD `<semaphore.h>`

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

### Issue 6

The `sem_open()` function is marked as part of the Semaphores option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Semaphores option.

The `sem_timedwait()` function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/117 is applied, updating the DESCRIPTION to add the `sem_timedwait()` function for alignment with IEEE Std 1003.1d-1999.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/118 is applied, updating the DESCRIPTION to describe the conditions to return the same semaphore address on a call to `sem_open()`. The words "and at least one previous successful `sem_open()` call for this semaphore has not been matched with a `sem_close()` call" are added.

## Issue 7

Austin Group Interpretation 1003.1-2001 #066 is applied, updating the [ENOSPC] error case and adding the [ENOMEM] error case.

Austin Group Interpretation 1003.1-2001 #077 is applied, clarifying the `name` argument and adding [ENAMETOOLONG] as a "may fail" error.

Austin Group Interpretation 1003.1-2001 #141 is applied, adding FUTURE DIRECTIONS.

SD5-XSH-ERN-170 is applied, updating the DESCRIPTION to clarify the wording for setting the user ID and group ID of the semaphore.

The `sem_open()` function is moved from the Semaphores option to the Base.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0527 [37] is applied.

## Issue 8

Austin Group Defect 368 is applied, adding a statement that a named semaphore may be implemented using a file descriptor and changing the ERRORS section.

Austin Group Defect 1216 is applied, adding `sem_clockwait()`.

Austin Group Defect 1324 is applied, making it implementation-defined whether the same handle or a unique handle is returned when multiple successful calls to `sem_open()` are made with the same value for `name`.

## 1.196. `sem_post` — unlock a semaphore

---

### SYNOPSIS

```
#include <semaphore.h>

int sem_post(sem_t *sem);
```

### DESCRIPTION

The `sem_post()` function shall unlock the semaphore referenced by `sem` by performing a semaphore unlock operation on that semaphore.

If the semaphore value resulting from this operation is positive, then no threads were blocked waiting for the semaphore to become unlocked; the semaphore value is simply incremented.

If the value of the semaphore resulting from this operation is zero, then one of the threads blocked waiting for the semaphore shall be allowed to return successfully from its call to `sem_wait()`. [PS] If the Process Scheduling option is supported, the thread to be unblocked shall be chosen in a manner appropriate to the scheduling policies and parameters in effect for the blocked threads. In the case of the schedulers `SCHED_FIFO` and `SCHED_RR`, the highest priority waiting thread shall be unblocked, and if there is more than one highest priority thread blocked waiting for the semaphore, then the highest priority thread that has been waiting the longest shall be unblocked. If the Process Scheduling option is not defined, the choice of a thread to unblock is unspecified.

[SS] If the Process Sporadic Server option is supported, and the scheduling policy is `SCHED_SPORADIC`, the semantics are as per `SCHED_FIFO` above.

The `sem_post()` function shall be async-signal-safe and may be invoked from a signal-catching function.

### RETURN VALUE

If successful, the `sem_post()` function shall return zero; otherwise, the function shall return -1 and set `errno` to indicate the error.

# ERRORS

The `sem_post()` function shall fail if:

- **[EOVERFLOW]** The maximum allowable value of the semaphore would be exceeded.

The `sem_post()` function may fail if:

- **[EINVAL]** The `sem` argument does not refer to a valid semaphore.

# EXAMPLES

See `sem_clockwait()`.

# APPLICATION USAGE

None.

# RATIONALE

None.

# FUTURE DIRECTIONS

None.

# SEE ALSO

- `semctl()`
- `semget()`
- `semop()`
- `sem_clockwait()`
- `sem_trywait()`

# CHANGE HISTORY

## First released in Issue 5

Included for alignment with the POSIX Realtime Extension.

## Issue 6

- The `sem_post()` function is marked as part of the Semaphores option.
- The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Semaphores option.
- The `sem_timedwait()` function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.
- SCHED\_SPORADIC is added to the list of scheduling policies for which the thread that is to be unblocked is specified for alignment with IEEE Std 1003.1d-1999.
- IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/119 is applied, updating the ERRORS section so that the [EINVAL] error becomes optional.

## Issue 7

- Austin Group Interpretation 1003.1-2001 #156 is applied.
- The `sem_post()` function is moved from the Semaphores option to the Base.
- POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0528 [37] is applied.

## Issue 8

- Austin Group Defect 315 is applied, adding the [EOVERFLOW] error.
-

## 1.197. `sem_clockwait`, `sem_timedwait` — lock a semaphore

### SYNOPSIS

```
#include <semaphore.h>

int sem_clockwait(sem_t *restrict sem,
                  clockid_t clock_id,
                  const struct timespec *restrict abstime);

int sem_timedwait(sem_t *restrict sem,
                  const struct timespec *restrict abstime);
```

### DESCRIPTION

The `sem_clockwait()` and `sem_timedwait()` functions shall lock the semaphore referenced by `sem` as in the `sem_wait()` function. However, if the semaphore cannot be locked without waiting for another process or thread to unlock the semaphore by performing a `sem_post()` function, this wait shall be terminated when the specified timeout expires.

The timeout shall expire when the absolute time specified by `abstime` passes, as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds `abstime`), or if the absolute time specified by `abstime` has already been passed at the time of the call.

For `sem_timedwait()`, the timeout shall be based on the `CLOCK_REALTIME` clock. For `sem_clockwait()`, the timeout shall be based on the clock specified by the `clock_id` argument. The resolution of the timeout shall be the resolution of the clock on which it is based. Implementations shall support passing `CLOCK_REALTIME` and `CLOCK_MONOTONIC` to `sem_clockwait()` as the `clock_id` argument.

Under no circumstance shall the function fail with a timeout if the semaphore can be locked immediately. The validity of the `abstime` need not be checked if the semaphore can be locked immediately.

## RETURN VALUE

The `sem_clockwait()` and `sem_timedwait()` functions shall return zero if the calling process successfully performed the semaphore lock operation on the semaphore designated by `sem`. If the call was unsuccessful, the state of the semaphore shall be unchanged, and the functions shall return a value of -1 and set `errno` to indicate the error.

## ERRORS

The `sem_clockwait()` and `sem_timedwait()` functions shall fail if:

- **[EINVAL]**
- The process or thread would have blocked, and either the `abstime` parameter specified a nanoseconds field value less than zero or greater than or equal to 1000 million, or the `sem_clockwait()` function was passed an invalid or unsupported `clock_id` value.
- **[ETIMEDOUT]**
- The semaphore could not be locked before the specified timeout expired.

The `sem_clockwait()` and `sem_timedwait()` functions may fail if:

- **[EDEADLK]**
- A deadlock condition was detected.
- **[EINTR]**
- A signal interrupted the function.
- **[EINVAL]**
- The `sem` argument does not refer to a valid semaphore.

## EXAMPLES

The program shown below operates on an unnamed semaphore. The program expects two command-line arguments. The first argument specifies a seconds value that is used to set an alarm timer to generate a SIGALRM signal. This handler performs a `sem_post()` to increment the semaphore that is being waited on in `main()` using `sem_clockwait()`. The second command-line argument specifies the length of the timeout, in seconds, for `sem_clockwait()`.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <time.h>
#include <assert.h>
#include <errno.h>
#include <signal.h>

sem_t sem;

static void
handler(int sig)
{
    int sav_errno = errno;
    static const char info_msg[] = "sem_post() from handler\n";
    write(STDOUT_FILENO, info_msg, sizeof info_msg - 1);
    if (sem_post(&sem) == -1) {
        static const char err_msg[] = "sem_post() failed\n";
        write(STDERR_FILENO, err_msg, sizeof err_msg - 1);
        _exit(EXIT_FAILURE);
    }
    errno = sav_errno;
}

int
main(int argc, char *argv[])
{
    struct sigaction sa;
    struct timespec ts;
    int s;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s <alarm-secs> <wait-secs>\n",
                argv[0]);
        exit(EXIT_FAILURE);
    }

    if (sem_init(&sem, 0, 0) == -1) {
        perror("sem_init");
        exit(EXIT_FAILURE);
    }

    /* Establish SIGALRM handler; set alarm timer using argv[1] */

    sa.sa_handler = handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    if (sigaction(SIGALRM, &sa, NULL) == -1) {
        perror("sigaction");
```

```
        exit(EXIT_FAILURE);
    }

    alarm(atoi(argv[1]));

    /* Calculate relative interval as current time plus
       number of seconds given argv[2] */

    if (clock_gettime(CLOCK_MONOTONIC, &ts) == -1) {
        perror("clock_gettime");
        exit(EXIT_FAILURE);
    }
    ts.tv_sec += atoi(argv[2]);

    printf("main() about to call sem_clockwait()\n");
    while ((s = sem_clockwait(&sem, CLOCK_MONOTONIC, &ts)) == -1 &
           errno == EINTR)
        continue;          /* Restart if interrupted by handler */

    /* Check what happened */

    if (s == -1) {
        if (errno == ETIMEDOUT)
            printf("sem_clockwait() timed out\n");
        else
            perror("sem_clockwait");
    } else
        printf("sem_clockwait() succeeded\n");

    exit((s == 0) ? EXIT_SUCCESS : EXIT_FAILURE);
}
```

## APPLICATION USAGE

Applications using these functions may be subject to priority inversion, as discussed in XBD 3.275 Priority Inversion.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `sem_post()`
- `sem_trywait()`
- `semctl()`
- `semget()`
- `semop()`
- `time()`

XBD 3.275 Priority Inversion, `<semaphore.h>`, `<time.h>`

## CHANGE HISTORY

**First released in Issue 6. Derived from IEEE Std 1003.1d-1999.**

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/120 is applied, updating the ERRORS section so that the [EINVAL] error becomes optional.

### Issue 7

The `sem_timedwait()` function is moved from the Semaphores option to the Base.

Functionality relating to the Timers option is moved to the Base.

An example is added.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0529 [138] is applied.

### Issue 8

Austin Group Defect 592 is applied, removing text relating to `<time.h>` from the SYNOPSIS and DESCRIPTION sections.

Austin Group Defect 1216 is applied, adding `sem_clockwait()`.

---

## 1.198. `sem_trywait`, `sem_wait` — lock a semaphore

---

### SYNOPSIS

```
#include <semaphore.h>

int sem_trywait(sem_t *sem);
int sem_wait(sem_t *sem);
```

### DESCRIPTION

The `sem_trywait()` function shall lock the semaphore referenced by `sem` only if the semaphore is currently not locked; that is, if the semaphore value is currently positive. Otherwise, it shall not lock the semaphore.

The `sem_wait()` function shall lock the semaphore referenced by `sem` by performing a semaphore lock operation on that semaphore. If the semaphore value is currently zero, then the calling thread shall not return from the call to `sem_wait()` until it either locks the semaphore or the call is interrupted by a signal.

Upon successful return, the state of the semaphore shall be locked and shall remain locked until the `sem_post()` function is executed and returns successfully.

The `sem_wait()` function is interruptible by the delivery of a signal.

### RETURN VALUE

The `sem_trywait()` and `sem_wait()` functions shall return zero if the calling process successfully performed the semaphore lock operation on the semaphore designated by `sem`. If the call was unsuccessful, the state of the semaphore shall be unchanged, and the function shall return a value of -1 and set `errno` to indicate the error.

### ERRORS

The `sem_trywait()` function shall fail if:

- **[EAGAIN]**

The semaphore was already locked, so it cannot be immediately locked by the

`sem_trywait()` operation.

The `sem_trywait()` and `sem_wait()` functions may fail if:

- **[EDEADLK]**  
A deadlock condition was detected.
- **[EINTR]**  
A signal interrupted this function.
- **[EINVAL]**  
The `sem` argument does not refer to a valid semaphore.

## APPLICATION USAGE

Applications using these functions may be subject to priority inversion, as discussed in XBD 3.275 Priority Inversion.

## SEE ALSO

- `semctl()`
- `semget()`
- `semop()`
- `sem_clockwait()`
- `sem_post()`

XBD:

- [3.275 Priority Inversion](#)
- [4.15.2 Memory Synchronization](#)
- 

## CHANGE HISTORY

### Issue 6

- The `sem_trywait()` and `sem_wait()` functions are marked as part of the Semaphores option.
- The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Semaphores option.

- The `sem_timedwait()` function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.
- IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/121 is applied, updating the ERRORS section so that the [EINVAL] error becomes optional.

## Issue 7

- SD5-XSH-ERN-54 is applied, removing the `sem_wait()` function from the "shall fail" error cases.
  - The `sem_trywait()` and `sem_wait()` functions are moved from the Semaphores option to the Base.
  - POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0530 [37] is applied.
-

## 1.199. `sem_unlink` - remove a named semaphore

---

### SYNOPSIS

```
#include <semaphore.h>

int sem_unlink(const char *name);
```

### DESCRIPTION

The `sem_unlink()` function shall remove the semaphore named by the string `name`. If the semaphore named by `name` is currently referenced by other processes, then `sem_unlink()` shall have no effect on the state of the semaphore. If one or more processes have the semaphore open when `sem_unlink()` is called, destruction of the semaphore is postponed until all references to the semaphore have been destroyed by calls to `sem_close()`, `_exit()`, or `exec`. Calls to `sem_open()` to recreate or reconnect to the semaphore refer to a new semaphore after `sem_unlink()` is called. The `sem_unlink()` call shall not block until all references have been destroyed; it shall return immediately.

### RETURN VALUE

Upon successful completion, the `sem_unlink()` function shall return a value of 0. Otherwise, the semaphore shall not be changed and the function shall return a value of -1 and set `errno` to indicate the error.

### ERRORS

The `sem_unlink()` function shall fail if:

- **[EACCES]** - Permission is denied to unlink the named semaphore.
- **[ENOENT]** - The named semaphore does not exist.

The `sem_unlink()` function may fail if:

- **[ENAMETOOLONG]** - The length of the `name` argument exceeds `{_POSIX_PATH_MAX}` on systems that do not support the XSI option or exceeds `{_XOPEN_PATH_MAX}` on XSI systems, or has a pathname

component that is longer than `{_POSIX_NAME_MAX}` on systems that do not support the XSI option or longer than `{_XOPEN_NAME_MAX}` on XSI systems. A call to `sem_unlink()` with a `name` argument that contains the same semaphore name as was previously used in a successful `sem_open()` call shall not give an [ENAMETOOLONG] error.

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

A future version might require the `sem_open()` and `sem_unlink()` functions to have semantics similar to normal file system operations.

## SEE ALSO

- `semctl()`
- `semget()`
- `semop()`
- `sem_close()`
- `sem_open()`
- XBD `<semaphore.h>`

# CHANGE HISTORY

## First released in Issue 5

Included for alignment with the POSIX Realtime Extension.

## Issue 6

- The `sem_unlink()` function is marked as part of the Semaphores option.
- The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Semaphores option.

## Issue 7

- Austin Group Interpretation 1003.1-2001 #077 is applied, changing [ENAMETOOLONG] from a "shall fail" to a "may fail" error.
  - Austin Group Interpretation 1003.1-2001 #141 is applied, adding FUTURE DIRECTIONS.
  - The `sem_unlink()` function is moved from the Semaphores option to the Base.
  - POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0531 [37] is applied.
-

## 1.200. `sem_trywait`, `sem_wait` — lock a semaphore

---

### SYNOPSIS

```
#include <semaphore.h>

int sem_trywait(sem_t *sem);
int sem_wait(sem_t *sem);
```

### DESCRIPTION

The `sem_trywait()` function shall lock the semaphore referenced by `sem` only if the semaphore is currently not locked; that is, if the semaphore value is currently positive. Otherwise, it shall not lock the semaphore.

The `sem_wait()` function shall lock the semaphore referenced by `sem` by performing a semaphore lock operation on that semaphore. If the semaphore value is currently zero, then the calling thread shall not return from the call to `sem_wait()` until it either locks the semaphore or the call is interrupted by a signal.

Upon successful return, the state of the semaphore shall be locked and shall remain locked until the `sem_post()` function is executed and returns successfully.

The `sem_wait()` function is interruptible by the delivery of a signal.

### RETURN VALUE

The `sem_trywait()` and `sem_wait()` functions shall return zero if the calling process successfully performed the semaphore lock operation on the semaphore designated by `sem`. If the call was unsuccessful, the state of the semaphore shall be unchanged, and the function shall return a value of -1 and set `errno` to indicate the error.

### ERRORS

The `sem_trywait()` function shall fail if:

- **[EAGAIN]**

The semaphore was already locked, so it cannot be immediately locked by the

`sem_trywait()` operation.

The `sem_trywait()` and `sem_wait()` functions may fail if:

- **[EDEADLK]**  
A deadlock condition was detected.
  - **[EINTR]**  
A signal interrupted this function.
  - **[EINVAL]**  
The `sem` argument does not refer to a valid semaphore.
- 

## APPLICATION USAGE

Applications using these functions may be subject to priority inversion, as discussed in XBD 3.275 Priority Inversion.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `semctl()`
- `semget()`
- `semop()`
- `sem_clockwait()`
- `sem_post()`

XBD 3.275 Priority Inversion, 4.15.2 Memory Synchronization, `<semaphore.h>`

# CHANGE HISTORY

## First released in Issue 5

Included for alignment with the POSIX Realtime Extension.

## Issue 6

- The `sem_trywait()` and `sem_wait()` functions are marked as part of the Semaphores option.
- The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Semaphores option.
- The `sem_timedwait()` function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.
- IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/121 is applied, updating the ERRORS section so that the [EINVAL] error becomes optional.

## Issue 7

- SD5-XSH-ERN-54 is applied, removing the `sem_wait()` function from the "shall fail" error cases.
  - The `sem_trywait()` and `sem_wait()` functions are moved from the Semaphores option to the Base.
  - POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0530 [37] is applied.
-

## 1.201. setbuf

---

### SYNOPSIS

```
#include <stdio.h>

void setbuf(FILE *restrict stream, char *restrict buf);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

Except that it returns no value, the function call:

```
setbuf(stream, buf)
```

shall be equivalent to:

```
setvbuf(stream, buf, _I0FBF, BUFSIZ)
```

if `buf` is not a null pointer, or to:

```
setvbuf(stream, buf, _IONBF, BUFSIZ)
```

if `buf` is a null pointer.

### RETURN VALUE

The `setbuf()` function shall not return a value.

### ERRORS

Although the `setvbuf()` interface may set `errno` in defined ways, the value of `errno` after a call to `setbuf()` is unspecified.

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

A common source of error is allocating buffer space as an "automatic" variable in a code block, and then failing to close the stream in the same block.

With `setbuf()`, allocating a buffer of BUFSIZ bytes does not necessarily imply that all of BUFSIZ bytes are used for the buffer area.

Since `errno` is not required to be unchanged on success, in order to correctly detect and possibly recover from errors, applications should use `setvbuf()` instead of `setbuf()`.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [2.5 Standard I/O Streams](#)
- [fopen\(\)](#)
- [setvbuf\(\)](#)

XBD

## CHANGE HISTORY

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

## Issue 6

The prototype for `setbuf()` is updated for alignment with the ISO/IEC 9899:1999 standard.

## Issue 7

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0546 [397], XSH/TC1-2008/0547 [397], and XSH/TC1-2008/0548 [14] are applied.

## 1.202. `setenv` — add or change environment variable

### SYNOPSIS

```
#include <stdlib.h>

int setenv(const char *envname, const char *envval, int overwrite)
```

### DESCRIPTION

The `setenv()` function shall update or add a variable in the environment of the calling process. The `envname` argument points to a string containing the name of an environment variable to be added or altered. The environment variable shall be set to the value to which `envval` points. The function shall fail if `envname` points to a string which contains an '=' character.

If the environment variable named by `envname` already exists and the value of `overwrite` is non-zero, the function shall return success and the environment shall be updated. If the environment variable named by `envname` already exists and the value of `overwrite` is zero, the function shall return success and the environment shall remain unchanged.

The `setenv()` function shall update the list of pointers to which `environ` points.

The strings described by `envname` and `envval` are copied by this function.

The `setenv()` function need not be thread-safe.

### RETURN VALUE

Upon successful completion, zero shall be returned. Otherwise, -1 shall be returned, `errno` set to indicate the error, and the environment shall be unchanged.

### ERRORS

The `setenv()` function shall fail if:

- **[EINVAL]**

- The `envname` argument points to an empty string or points to a string containing an '=' character.
- **[ENOMEM]**
- Insufficient memory was available to add a variable or its value to the environment.

## EXAMPLES

None.

## APPLICATION USAGE

See `exec()` for restrictions on changing the environment in multi-threaded applications.

## RATIONALE

Unanticipated results may occur if `setenv()` changes the external variable `environ`. In particular, if the optional `envp` argument to `main()` is present, it is not changed, and thus may point to an obsolete copy of the environment (as may any other copy of `environ`). However, other than the aforementioned restriction, the standard developers intended that the traditional method of walking through the environment by way of the `environ` pointer must be supported.

It was decided that `setenv()` should be required by this version because it addresses a piece of missing functionality, and does not impose a significant burden on the implementor.

There was considerable debate as to whether the System V `putenv()` function or the BSD `setenv()` function should be required as a mandatory function. The `setenv()` function was chosen because it permitted the implementation of the `unsetenv()` function to delete environmental variables, without specifying an additional interface. The `putenv()` function is available as part of the XSI option.

The standard developers considered requiring that `setenv()` indicate an error when a call to it would result in exceeding `{ARG_MAX}`. The requirement was rejected since the condition might be temporary, with the application eventually reducing the environment size. The ultimate success or failure depends on the size at the time of a call to `exec`, which returns an indication of this error condition.

See also the RATIONALE section in `getenv()`.

## FUTURE DIRECTIONS

None.

## SEE ALSO

[exec](#),  
[getenv\(\)](#),  
[putenv\(\)](#),  
[unsetenv\(\)](#)

XBD ,

,

## CHANGE HISTORY

First released in Issue 6. Derived from the IEEE P1003.1a draft standard.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/55 is applied, adding references to [exec](#) in the APPLICATION USAGE and SEE ALSO sections.

### Issue 7

Austin Group Interpretation 1003.1-2001 #156 is applied.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0549 [167], XSH/TC1-2008/0550 [185], XSH/TC1-2008/0551 [167], and XSH/TC1-2008/0552 [38] are applied.

---

## 1.203. setjmp - set jump point for a non-local goto

---

### Synopsis

```
#include <setjmp.h>
void setjmp(jmp_buf env);
```

### Description

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

A call to `setjmp()` shall save the calling environment in its `env` argument for later use by `longjmp()`.

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 behavior is undefined.

An application shall ensure that an invocation of `setjmp()` appears in one of the following contexts only:

- The entire controlling expression of a selection or iteration statement
- One operand of a relational or equality operator with the other operand an integral constant expression, with the resulting expression being the entire controlling expression of a selection or iteration statement
- The operand of a unary '!' operator with the resulting expression being the entire controlling expression of a selection or iteration
- The entire expression of an expression statement (possibly cast to `void`)

If the invocation appears in any other context, the behavior is undefined.

*The following sections are informative.*

## 1.204. `setlocale` — set program locale

---

### SYNOPSIS

```
#include <locale.h>

char *setlocale(int category, const char *locale);
```

### DESCRIPTION

The `setlocale()` function selects the appropriate piece of the global locale, as specified by the `category` and `locale` arguments, and can be used to change or query the entire global locale or portions thereof. The value `LC_ALL` for `category` names the entire global locale; other values for `category` name only a part of the global locale:

- **LC\_COLLATE**

Affects the behavior of regular expressions and the collation functions.

- **LC\_CTYPE**

Affects the behavior of regular expressions, character classification, character conversion functions, and wide-character functions.

- **LC\_MESSAGES**

Affects the affirmative and negative response expressions returned by `nl_langinfo()` and the way message catalogs are located. It may also affect the behavior of functions that return or write message strings.

- **LC\_MONETARY**

Affects the behavior of functions that handle monetary values.

- **LC\_NUMERIC**

Affects the behavior of functions that handle numeric values.

- **LC\_TIME**

Affects the behavior of the time conversion functions.

The `locale` argument is a pointer to a character string containing the required setting of `category`. The contents of this string are implementation-defined. In addition, the following preset values of `locale` are defined for all settings of `category`:

- **"POSIX"**

Specifies the minimal environment for C-language translation called the

POSIX locale. The POSIX locale is the default global locale at entry to `main()`.

- "C"

Equivalent to "POSIX".

- ""

Specifies an implementation-defined native environment. The determination of the name of the new locale for the specified category depends on the value of the associated environment variables, `LC_*` and `LANG`; see the Base Definitions volume of POSIX.1-2024.

- **A null pointer**

Directs `setlocale()` to query the current global locale setting and return the name of the locale if `category` is not `LC_ALL`, or a string which encodes the locale name(s) for all of the individual categories if `category` is `LC_ALL`.

Setting all of the categories of the global locale is similar to successively setting each individual category of the global locale, except that all error checking is done before any actions are performed. To set all the categories of the global locale, `setlocale()` can be invoked as:

```
setlocale(LC_ALL, "");
```

In this case, `setlocale()` shall first verify that the values of all the environment variables it needs according to the precedence rules indicate supported locales. If the value of any of these environment variable searches yields a locale that is not supported (and non-null), `setlocale()` shall return a null pointer and the global locale shall not be changed. If all environment variables name supported locales, `setlocale()` shall proceed as if it had been called for each category, using the appropriate value from the associated environment variable or from the implementation-defined default if there is no such value.

The global locale established using `setlocale()` shall only be used in threads for which no current locale has been set using `uselocale()` or whose current locale has been set to the global locale using `uselocale(LC_GLOBAL_LOCALE)`.

The implementation shall behave as if no function defined in this volume of POSIX.1-2024 calls `setlocale()`.

The `setlocale()` function need not be thread-safe; however, it shall avoid data races with all function calls that do not affect and are not affected by the global locale.

## RETURN VALUE

Upon successful completion, `setlocale()` shall return the string associated with the specified category for the new locale. Otherwise, `setlocale()` shall return a null pointer and the global locale shall not be changed.

A null pointer for `locale` shall cause `setlocale()` to return a pointer to the string associated with the specified `category` for the current global locale. The global locale shall not be changed.

The string returned by `setlocale()` is such that a subsequent call with that string and its associated `category` shall restore that part of the global locale. The application shall not modify the string returned. The returned string pointer might be invalidated or the string content might be overwritten by a subsequent call to `setlocale()`. The returned pointer might also be invalidated if the calling thread is terminated.

## ERRORS

No errors are defined.

## APPLICATION USAGE

The following code illustrates how a program can initialize the international environment for one language, while selectively modifying the global locale such that regular expressions and string operations can be applied to text recorded in a different language:

```
setlocale(LC_ALL, "De");
setlocale(LC_COLLATE, "Fr@dict");
```

Internationalized programs can initiate language operation according to environment variable settings by calling `setlocale()` as follows:

```
setlocale(LC_ALL, "");
```

Changing the setting of `LC_MESSAGES` has no effect on catalogs that have already been opened by calls to `catopen()`.

In order to make use of different locale settings while multiple threads are running, applications should use `uselocale()` in preference to `setlocale()`.

# RATIONALE

References to the international environment or locale in the following text relate to the global locale for the process. This can be overridden for individual threads using `uselocale()`.

The ISO C standard defines a collection of functions to support internationalization. One of the most significant aspects of these functions is a facility to set and query the *international environment*. The international environment is a repository of information that affects the behavior of certain functionality, namely:

1. Character handling
2. Collating
3. Date/time formatting
4. Numeric editing
5. Monetary formatting
6. Messaging

The `setlocale()` function provides the application developer with the ability to set all or portions, called *categories*, of the international environment. These categories correspond to the areas of functionality mentioned above.

There are two primary uses of `setlocale()`:

1. Querying the international environment to find out what it is set to
2. Setting the international environment, or *locale*, to a specific value

To query the international environment, `setlocale()` is invoked with a specific category and the null pointer as the locale. The null pointer is a special directive to `setlocale()` that tells it to query rather than set the international environment.

There are three ways to set the international environment with `setlocale()`:

- `setlocale(category, string)`

This usage sets a specific `category` in the international environment to a specific value corresponding to the value of the `string`. For example:

```
setlocale(LC_ALL, "fr_FR.ISO-8859-1");
```

- `setlocale(category, "C")`

The ISO C standard states that one locale must exist on all conforming implementations. The name of the locale is C and corresponds to a minimal international environment needed to support the C programming language.

- `setlocale(category, "")`

This sets a specific category to an implementation-defined default. This corresponds to the value of the environment variables.

## SEE ALSO

<code>catopen()</code>	<code>exec</code>	<code>fprintf()</code>	<code>fscanf()</code>	<code>getlocalename_l()</code>
<code>isalnum()</code>	<code>isalpha()</code>	<code>isblank()</code>	<code>iscntrl()</code>	<code>isdigit()</code>
<code>isgraph()</code>	<code>islower()</code>	<code>isprint()</code>	<code>ispunct()</code>	<code>isspace()</code>
<code>isupper()</code>	<code>iswalnum()</code>	<code>iswalpha()</code>	<code>iswblank()</code>	<code>iswcntrl()</code>
<code>iswctype()</code>	<code>iswdigit()</code>	<code>iswgraph()</code>	<code>iswlower()</code>	<code>iswprint()</code>
<code>iswpunct()</code>	<code>iswspace()</code>	<code>iswupper()</code>	<code>iswxdigit()</code>	<code>isxdigit()</code>
<code>localeconv()</code>	<code>mblen()</code>	<code>mbstowcs()</code>	<code>mbtowc()</code>	<code>newlocale()</code>
<code>nl_langinfo()</code>	<code>perror()</code>	<code>psiginfo()</code>	<code>strcoll()</code>	<code>strerror()</code>
<code>strfmon()</code>	<code>strsignal()</code>	<code>strtod()</code>	<code>strxfrm()</code>	<code>tolower()</code>
<code>toupper()</code>	<code>towlower()</code>	<code>towupper()</code>	<code>use locale()</code>	<code>wcs coll()</code>
<code>wcstod()</code>	<code>wcstombs()</code>	<code>wcsxfrm()</code>	<code>wctomb()</code>	

XBD 7. Locale, 8. Environment Variables, `<langinfo.h>`, `<locale.h>`

## CHANGE HISTORY

First released in Issue 3. Updated in subsequent issues to align with the POSIX Threads Extension, mark extensions beyond the ISO C standard, and apply various technical corrigenda.

## 1.205. setvbuf — assign buffering to a stream

### SYNOPSIS

```
#include <stdio.h>

int setvbuf(FILE *restrict stream, char *restrict buf, int type, size_t size);
```

### DESCRIPTION

The `setvbuf()` function may be used after the stream pointed to by `stream` is associated with an open file but before any other operation (other than an unsuccessful call to `setvbuf()`) is performed on the stream. The argument `type` determines how `stream` shall be buffered, as follows:

- `_IOFBF` shall cause input/output to be fully buffered.
- `_IOLBF` shall cause input/output to be line buffered.
- `_IONBF` shall cause input/output to be unbuffered.

If `buf` is not a null pointer, the array it points to may be used instead of a buffer allocated by `setvbuf()` and the argument `size` specifies the size of the array; otherwise, `size` may determine the size of a buffer allocated by the `setvbuf()` function. The contents of the array at any time are unspecified.

For information about streams, see [2.5 Standard I/O Streams](#).

### RETURN VALUE

Upon successful completion, `setvbuf()` shall return 0. Otherwise, it shall return a non-zero value if an invalid value is given for `type` or if the request cannot be honored, and may set `errno` to indicate the error.

### ERRORS

The `setvbuf()` function may fail if:

- **EBADF**: The stream is not a memory stream and the file descriptor underlying the stream is not valid.

## EXAMPLES

None.

## APPLICATION USAGE

A common source of error is allocating buffer space as an "automatic" variable in a code block, and then failing to close the stream in the same block.

With `setvbuf()`, allocating a buffer of `size` bytes does not necessarily imply that all of `size` bytes are used for the buffer area.

Applications should note that many implementations only provide line buffering on input from terminal devices.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [2.5 Standard I/O Streams](#)
- [fopen\(\)](#)
- [setbuf\(\)](#)
- 

## CHANGE HISTORY

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

### Issue 6

Extensions beyond the ISO C standard are marked.

The `setvbuf()` prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

## Issue 8

Austin Group Defect 1144 is applied, changing the [EBADF] error condition.

---

## 1.206. `shm_open` — open a shared memory object (REALTIME)

---

### SYNOPSIS

```
[SHM] #include <sys/mman.h>

int shm_open(const char *name, int oflag, mode_t mode);
```

### DESCRIPTION

The `shm_open()` function shall establish a connection between a shared memory object and a file descriptor. It shall create an open file description that refers to the shared memory object and a file descriptor that refers to that open file description. The file descriptor shall be allocated as described in [2.6 File Descriptor Allocation](#), and can be used by other functions to refer to that shared memory object.

The `name` argument points to a string naming a shared memory object. It is unspecified whether the name appears in the file system and is visible to other functions that take pathnames as arguments. The `name` argument conforms to the construction rules for a pathname, except that the interpretation of `<slash>` characters other than the leading `<slash>` character in `name` is implementation-defined, and that the length limits for the `name` argument are implementation-defined and need not be the same as the pathname limits `{PATH_MAX}` and `{NAME_MAX}`.

If `name` begins with the `<slash>` character, then processes calling `shm_open()` with the same value of `name` refer to the same shared memory object, as long as that name has not been removed. If `name` does not begin with the `<slash>` character, the effect is implementation-defined.

If successful, `shm_open()` shall return a file descriptor for the shared memory object. The open file description is new, and therefore the file descriptor does not share it with any other processes. It is unspecified whether the file offset is set. The `FD_CLOEXEC` file descriptor flag associated with the new file descriptor shall be set.

The file status flags and file access modes of the open file description shall be set according to the value of `oflag`. The `oflag` argument is the bitwise-inclusive OR of the following flags. Applications specify exactly one of the first two values (access modes) below in the value of `oflag`:

## Access Modes

- `O_RDONLY` - Open for read access only.
- `O_RDWR` - Open for read or write access.

## Additional Flags

Any combination of the remaining flags can be specified in the value of `oflag` :

- `O_CREAT` - If the shared memory object exists, this flag has no effect, except as noted under `O_EXCL` below. Otherwise, the shared memory object is created. The user ID of the shared memory object shall be set to the effective user ID of the process. The group ID of the shared memory object shall be set to the effective group ID of the process; however, if the `name` argument is visible in the file system, the group ID may be set to the group ID of the containing directory. The permission bits of the shared memory object shall be set to the value of the `mode` argument except those set in the file mode creation mask of the process. When bits in `mode` other than the file permission bits are set, the effect is unspecified. The `mode` argument does not affect whether the shared memory object is opened for reading, for writing, or for both. The shared memory object has a size of zero.
- `O_EXCL` - If `O_EXCL` and `O_CREAT` are set, `shm_open()` fails if the shared memory object exists. The check for the existence of the shared memory object and the creation of the object if it does not exist is atomic with respect to other processes executing `shm_open()` naming the same shared memory object with `O_EXCL` and `O_CREAT` set. If `O_EXCL` is set and `O_CREAT` is not set, the result is undefined.
- `O_TRUNC` - If the shared memory object exists, and it is successfully opened `O_RDWR`, the object shall be truncated to zero length and the mode and owner shall be unchanged by this function call. The result of using `O_TRUNC` with `O_RDONLY` is undefined.

## Atomic Operations

The following functions shall be atomic with respect to each other in the effects specified in POSIX.1-2024 when they operate on shared memory objects:

- `close()`
- `ftruncate()`
- `mmap()`

- `shm_open()`
- `shm_unlink()`

If two threads each call one of these functions, each call shall either see all of the specified effects of the other call, or none of them. The requirement on the `close()` function shall also apply whenever a file descriptor is successfully closed, however caused (for example, as a consequence of calling `close()`, calling `dup2()`, or of process termination).

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.

## RETURN VALUE

Upon successful completion, the `shm_open()` function shall return a non-negative integer representing the file descriptor. Otherwise, it shall return -1 and set `errno` to indicate the error.

## ERRORS

The `shm_open()` function shall fail if:

- **[EACCES]** - The shared memory object exists and the permissions specified by `oflag` 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.
- **[EEXIST]** - `O_CREAT` and `O_EXCL` are set and the named shared memory object already exists.
- **[EINTR]** - The `shm_open()` operation was interrupted by a signal.
- **[EINVAL]** - The `shm_open()` operation is not supported for the given name.
- **[EMFILE]** - All file descriptors available to the process are currently open.
- **[ENFILE]** - Too many shared memory objects are currently open in the system.
- **[ENOENT]** - `O_CREAT` is not set and the named shared memory object does not exist.

- **[ENOSPC]** - There is insufficient space for the creation of the new shared memory object.

The `shm_open()` function may fail if:

- **[ENAMETOOLONG]** - The length of the `name` argument exceeds `{_POSIX_PATH_MAX}` on systems that do not support the XSI option [XSI] or exceeds `{_XOPEN_PATH_MAX}` on XSI systems, or has a pathname component that is longer than `{_POSIX_NAME_MAX}` on systems that do not support the XSI option [XSI] or longer than `{_XOPEN_NAME_MAX}` on XSI systems.

## EXAMPLES

### Creating and Mapping a Shared Memory Object

The following code segment demonstrates the use of `shm_open()` to create a shared memory object which is then sized using `ftruncate()` before being mapped into the process address space using `mmap()`:

```
#include <unistd.h>
#include <sys/mman.h>
...

#define MAX_LEN 10000
struct region {           /* Defines "structure" of shared memory */
    int len;
    char buf[MAX_LEN];
};
struct region *rptr;
int fd;

/* Create shared memory object and set its size */

fd = shm_open("/myregion", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
if (fd == -1)
    /* Handle error */;

if (ftruncate(fd, sizeof(struct region)) == -1)
    /* Handle error */;

/* Map shared memory object */

rptr = mmap(NULL, sizeof(struct region),
            PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
if (rptr == MAP_FAILED)
    /* Handle error */;
```

```
/* Now we can refer to mapped region using fields of rptr;
   for example, rptr->len */
...
```

## APPLICATION USAGE

None.

## RATIONALE

When the Memory Mapped Files option is supported, the normal `open()` call is used to obtain a descriptor to a file to be mapped according to existing practice with `mmap()`. When the Shared Memory Objects option is supported, the `shm_open()` function shall obtain a descriptor to the shared memory object to be mapped.

There is ample precedent for having a file descriptor represent several types of objects. In the POSIX.1-1990 standard, a file descriptor can represent a file, a pipe, a FIFO, a tty, or a directory. Many implementations simply have an operations vector, which is indexed by the file descriptor type and does very different operations. Note that in some cases the file descriptor passed to generic operations on file descriptors is returned by `open()` or `creat()` and in some cases returned by alternate functions, such as `pipe()`. The latter technique is used by `shm_open()`.

Note that such shared memory objects can actually be implemented as mapped files. In both cases, the size can be set after the open using `ftruncate()`. The `shm_open()` function itself does not create a shared object of a specified size because this would duplicate an extant function that set the size of an object referenced by a file descriptor.

On implementations where memory objects are implemented using the existing file system, the `shm_open()` function may be implemented using a macro that invokes `open()`, and the `shm_unlink()` function may be implemented using a macro that invokes `unlink()`.

For implementations without a permanent file system, the definition of the name of the memory objects is allowed not to survive a system reboot. Note that this allows systems with a permanent file system to implement memory objects as data structures internal to the implementation as well.

On implementations that choose to implement memory objects using memory directly, a `shm_open()` followed by an `ftruncate()` and `close()` can be

used to preallocate a shared memory area and to set the size of that preallocation. This may be necessary for systems without virtual memory hardware support in order to ensure that the memory is contiguous.

The set of valid open flags to `shm_open()` was restricted to `O_RDONLY`, `O_RDWR`, `O_CREAT`, and `O_TRUNC` because these could be easily implemented on most memory mapping systems. This volume of POSIX.1-2024 is silent on the results if the implementation cannot supply the requested file access because of implementation-defined reasons, including hardware ones. The `O_CLOEXEC` open flag is not listed, because all shared memory objects are created with the `FD_CLOEXEC` flag already set; an application can later use `fcntl()` to clear that flag to allow the shared memory file descriptor to be preserved across the `exec` family of functions.

The error conditions `[EACCES]` and `[ENOTSUP]` are provided to inform the application that the implementation cannot complete a request.

`[EACCES]` indicates for implementation-defined reasons, probably hardware-related, that the implementation cannot comply with a requested mode because it conflicts with another requested mode. An example might be that an application desires to open a memory object two times, mapping different areas with different access modes. If the implementation cannot map a single area into a process space in two places, which would be required if different access modes were required for the two areas, then the implementation may inform the application at the time of the second open.

`[ENOTSUP]` indicates for implementation-defined reasons, probably hardware-related, that the implementation cannot comply with a requested mode at all. An example would be that the hardware of the implementation cannot support write-only shared memory areas.

On all implementations, it may be desirable to restrict the location of the memory objects to specific file systems for performance (such as a RAM disk) or implementation-defined reasons (shared memory supported directly only on certain file systems). The `shm_open()` function may be used to enforce these restrictions. There are a number of methods available to the application to determine an appropriate name of the file or the location of an appropriate directory. One way is from the environment via `getenv()`. Another would be from a configuration file.

This volume of POSIX.1-2024 specifies that memory objects have initial contents of zero when created. This is consistent with current behavior for both files and newly allocated memory. For those implementations that use physical memory, it would be possible that such implementations could simply use available memory and give it to the process uninitialized. This, however, is not consistent with standard behavior for the uninitialized data area, the stack, and of course, files.

Finally, it is highly desirable to set the allocated memory to zero for security reasons. Thus, initializing memory objects to zero is required.

## FUTURE DIRECTIONS

A future version might require the `shm_open()` and `shm_unlink()` functions to have semantics similar to normal file system operations.

## SEE ALSO

- [2.6 File Descriptor Allocation](#)
- `close()`
- `dup()`
- `exec`
- `fcntl()`
- `mmap()`
- `shmat()`
- `shmctl()`
- `shmdt()`
- `shm_unlink()`
- `umask()`
- XBD `<fcntl.h>`
- XBD `<sys/mman.h>`

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

### Issue 6

The `shm_open()` function is marked as part of the Shared Memory Objects option.

The `[ENOSYS]` error condition has been removed as stubs need not be provided if an implementation does not support the Shared Memory Objects option.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/126 is applied, adding the example to the EXAMPLES section.

## Issue 7

Austin Group Interpretation 1003.1-2001 #077 is applied, clarifying the `name` argument and changing `[ENAMETOOLONG]` from a "shall fail" to a "may fail" error.

Austin Group Interpretation 1003.1-2001 #141 is applied, adding FUTURE DIRECTIONS.

SD5-XSH-ERN-170 is applied, updating the DESCRIPTION to clarify the wording for setting the user ID and group ID of the shared memory object.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0324 [835], XSH/TC2-2008/0325 [835], and XSH/TC2-2008/0326 [835] are applied.

## Issue 8

Austin Group Defect 411 is applied, adding a sentence about `O_CLOEXEC` to the RATIONALE section.

Austin Group Defect 593 is applied, removing a reference to `<fcntl.h>` from the DESCRIPTION section.

Austin Group Defect 695 is applied, adding atomicity requirements to operations on shared memory objects.

---

## 1.207. `shm_unlink` — remove a shared memory object (REALTIME)

---

### SYNOPSIS

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

int shm_unlink(const char *name);
```

### DESCRIPTION

The `shm_unlink()` function shall remove the name of the shared memory object named by the string pointed to by `name`.

If one or more references to the shared memory object exist when the object is unlinked, the name shall be removed before `shm_unlink()` returns, but the removal of the memory object contents shall be postponed until all open and map references to the shared memory object have been removed.

Even if the object continues to exist after the last `shm_unlink()`, reuse of the name shall subsequently cause `shm_open()` to behave as if no shared memory object of this name exists (that is, `shm_open()` shall fail if `O_CREAT` is not set, or shall create a new shared memory object if `O_CREAT` is set).

### RETURN VALUE

Upon successful completion, a value of zero shall be returned. Otherwise, a value of -1 shall be returned and `errno` set to indicate the error. If -1 is returned, the named shared memory object shall not be changed by this function call.

### ERRORS

The `shm_unlink()` function shall fail if:

- **[EACCES]** - Permission is denied to unlink the named shared memory object.
- **[ENOENT]** - The named shared memory object does not exist.

The `shm_unlink()` function may fail if:

- **[ENAMETOOLONG]** - The length of the `name` argument exceeds `{_POSIX_PATH_MAX}` on systems that do not support the XSI option or exceeds `{_XOPEN_PATH_MAX}` on XSI systems, or has a pathname component that is longer than `{_POSIX_NAME_MAX}` on systems that do not support the XSI option or longer than `{_XOPEN_NAME_MAX}` on XSI systems. A call to `shm_unlink()` with a `name` argument that contains the same shared memory object name as was previously used in a successful `shm_open()` call shall not give an [ENAMETOOLONG] error.

## EXAMPLES

None.

## APPLICATION USAGE

Names of memory objects that were allocated with `open()` are deleted with `unlink()` in the usual fashion. Names of memory objects that were allocated with `shm_open()` are deleted with `shm_unlink()`. Note that the actual memory object is not destroyed until the last close and unmap on it have occurred if it was already in use.

## RATIONALE

None.

## FUTURE DIRECTIONS

A future version might require the `shm_open()` and `shm_unlink()` functions to have semantics similar to normal file system operations.

## SEE ALSO

- `close()`
- `mmap()`
- `munmap()`
- `shmat()`
- `shmctl()`

- `shmrdt()`
- `shm_open()`
- `<sys/mman.h>`

## CHANGE HISTORY

### First released in Issue 5

Included for alignment with the POSIX Realtime Extension.

### Issue 6

The `shm_unlink()` function is marked as part of the Shared Memory Objects option.

In the DESCRIPTION, text is added to clarify that reusing the same name after a `shm_unlink()` will not attach to the old shared memory object.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Shared Memory Objects option.

### Issue 7

Austin Group Interpretation 1003.1-2001 #077 is applied, changing [ENAMETOOLONG] from a "shall fail" to a "may fail" error.

Austin Group Interpretation 1003.1-2001 #141 is applied, adding FUTURE DIRECTIONS.

---

## 1.208. sigaction

### SYNOPSIS

```
#include <signal.h>

int sigaction(int sig, const struct sigaction *restrict act,
              struct sigaction *restrict oact);
```

### DESCRIPTION

The `sigaction()` function allows the calling process to examine and/or specify the action to be associated with a specific signal. The argument `sig` specifies the signal; acceptable values are defined in `<signal.h>`.

The structure `sigaction`, used to describe an action to be taken, is defined in the `<signal.h>` header to include at least the following members:

Member Type	Member Name	Description
<code>void (*)(int)</code>	<code>sa_handler</code>	Pointer to a signal-catching function or one of the macros <code>SIG_IGN</code> or <code>SIG_DFL</code> .
<code>sigset_t</code>	<code>sa_mask</code>	Additional set of signals to be blocked during execution of signal-catching function.
<code>int</code>	<code>sa_flags</code>	Special flags to affect behavior of signal.
<code>void (*)(int,            siginfo_t *, void            *)</code>	<code>sa_sigaction</code>	Pointer to a signal-catching function.

The storage occupied by `sa_handler` and `sa_sigaction` may overlap, and a conforming application shall not use both simultaneously.

If the argument `act` is not a null pointer, it points to a structure specifying the action to be associated with the specified signal. If the argument `oact` is not a null pointer, the action previously associated with the signal is stored in the location pointed to by the argument `oact`. If the argument `act` is a null pointer, signal

handling is unchanged; thus, the call can be used to enquire about the current handling of a given signal. The SIGKILL and SIGSTOP signals shall not be added to the signal mask using this mechanism; this restriction shall be enforced by the system without causing an error to be indicated.

If the `SA_SIGINFO` flag (see below) is cleared in the `sa_flags` field of the `sigaction` structure, the `sa_handler` field identifies the action to be associated with the specified signal. If the `SA_SIGINFO` flag is set in the `sa_flags` field, the `sa_sigaction` field specifies a signal-catching function.

The `sa_flags` field can be used to modify the behavior of the specified signal.

The following flags, defined in the `<signal.h>` header, can be set in `sa_flags`:

## SA\_NOCLDSTOP

Do not generate SIGCHLD when children stop [XSI] or stopped children continue.

If `sig` is SIGCHLD and the `SA_NOCLDSTOP` flag is not set in `sa_flags`, and the implementation supports the SIGCHLD signal, then a SIGCHLD signal shall be generated for the calling process whenever any of its child processes stop [XSI] and a SIGCHLD signal may be generated for the calling process whenever any of its stopped child processes are continued. If `sig` is SIGCHLD and the `SA_NOCLDSTOP` flag is set in `sa_flags`, then the implementation shall not generate a SIGCHLD signal in this way.

## SA\_ONSTACK

[XSI] If set and an alternate signal stack has been declared with `sigaltstack()`, the signal shall be delivered to the calling process on that stack. Otherwise, the signal shall be delivered on the current stack.

## SA\_RESETHAND

If set, the disposition of the signal shall be reset to SIG\_DFL and the `SA_SIGINFO` flag shall be cleared on entry to the signal handler.

**Note:** SIGILL and SIGTRAP cannot be automatically reset when delivered; the system silently enforces this restriction.

Otherwise, the disposition of the signal shall not be modified on entry to the signal handler.

In addition, if this flag is set, `sigaction()` may behave as if the `SA_NODEFER` flag were also set.

## SA\_RESTART

This flag affects the behavior of interruptible functions; that is, those specified to fail with `errno` set to [EINTR]. If set, and a function specified as interruptible is interrupted by this signal, the function shall restart and shall not fail with [EINTR] unless otherwise specified. If an interruptible function which uses a timeout is restarted, the duration of the timeout following the restart is set to an unspecified value that does not exceed the original timeout value. If the flag is not set, interruptible functions interrupted by this signal shall fail with `errno` set to [EINTR].

## SA\_SIGINFO

If cleared and the signal is caught, the signal-catching function shall be entered as:

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

where `signo` is the only argument to the signal-catching function. In this case, the application shall use the `sa_handler` member to describe the signal-catching function and the application shall not modify the `sa_sigaction` member.

If SA\_SIGINFO is set and the signal is caught, the signal-catching function shall be entered as:

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

where two additional arguments are passed to the signal-catching function. The second argument shall point to an object of type `siginfo_t` explaining the reason why the signal was generated; the third argument can be cast to a pointer to an object of type `ucontext_t` to refer to the receiving thread's context that was interrupted when the signal was delivered. In this case, the application shall use the `sa_sigaction` member to describe the signal-catching function and the application shall not modify the `sa_handler` member.

The `si_signo` member contains the system-generated signal number.

[XSI] The `si_errno` member may contain implementation-defined additional error information; if non-zero, it contains an error number identifying the condition that caused the signal to be generated.

The `si_code` member contains a code identifying the cause of the signal, as described in [2.4.3 Signal Actions](#).

## SA\_NOCLDWAIT

[XSI] If `sig` does not equal SIGCHLD, the behavior is unspecified. Otherwise, the behavior of the SA\_NOCLDWAIT flag is as specified in *Consequences of Process Termination*.

## SA\_NODEFER

If set and `sig` is caught, `sig` shall not be added to the thread's signal mask on entry to the signal handler unless it is included in `sa_mask`. Otherwise, `sig` shall always be added to the thread's signal mask on entry to the signal handler.

When a signal is caught by a signal-catching function installed by `sigaction()`, a new signal mask is calculated and installed for the duration of the signal-catching function (or until a call to either `sigprocmask()` or `sigsuspend()` is made). This mask is formed by taking the union of the current signal mask and the value of the `sa_mask` for the signal being delivered, and unless SA\_NODEFER or SA\_RESETHAND is set, then including the signal being delivered. If and when the user's signal handler returns normally, the original signal mask is restored.

Once an action is installed for a specific signal, it shall remain installed until another action is explicitly requested (by another call to `sigaction()`), until the SA\_RESETHAND flag causes resetting of the handler, or until one of the `exec` functions is called.

If the previous action for `sig` had been established by `signal()`, the values of the fields returned in the structure pointed to by `oact` are unspecified, and in particular `oact->sa_handler` is not necessarily the same value passed to `signal()`. However, if a pointer to the same structure or a copy thereof is passed to a subsequent call to `sigaction()` via the `act` argument, handling of the signal shall be as if the original call to `signal()` were repeated.

If `sigaction()` fails, no new signal handler is installed.

It is unspecified whether an attempt to set the action for a signal that cannot be caught or ignored to SIG\_DFL is ignored or causes an error to be returned with `errno` set to [EINVAL].

If SA\_SIGINFO is not set in `sa_flags`, then the disposition of subsequent occurrences of `sig` when it is already pending is implementation-defined; the signal-catching function shall be invoked with a single argument. If SA\_SIGINFO is set in `sa_flags`, then subsequent occurrences of `sig` generated by `sigqueue()` or as a result of any signal-generating function that supports the specification of an application-defined value (when `sig` is already pending) shall be queued in FIFO order until delivered or accepted; the signal-catching function shall be invoked with three arguments. The application specified value is passed to the signal-catching function as the `si_value` member of the `siginfo_t` structure.

The result of the use of `sigaction()` and a `sigwait()` function concurrently within a process on the same signal is unspecified.

## RETURN VALUE

Upon successful completion, `sigaction()` shall return 0; otherwise, -1 shall be returned, `errno` shall be set to indicate the error, and no new signal-catching function shall be installed.

## ERRORS

The `sigaction()` function shall fail if:

- **[EINVAL]** - The `sig` argument is not a valid signal number or an attempt is made to catch a signal that cannot be caught or ignore a signal that cannot be ignored.

The `sigaction()` function may fail if:

- **[EINVAL]** - An attempt was made to set the action to SIG\_DFL for a signal that cannot be caught or ignored (or both).

In addition, on systems that do not support the XSI option, the `sigaction()` function may fail if the `SA_SIGINFO` flag is set in the `sa_flags` field of the `sigaction` structure for a signal not in the range `SIGRTMIN` to `SIGRTMAX`.

*The following sections are informative.*

## EXAMPLES

### Establishing a Signal Handler

The following example demonstrates the use of `sigaction()` to establish a handler for the `SIGINT` signal.

```
#include <signal.h>

static void handler(int signum)
{
    /* Take appropriate actions for signal delivery */
}

int main(void)
{
    struct sigaction sa;
```

```

    sa.sa_handler = handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART; /* Restart functions if
                               interrupted by handler */
    if (sigaction(SIGINT, &sa, NULL) == -1)
        /* Handle error */;

    /* Further code */
}

```

## APPLICATION USAGE

The `sigaction()` function supersedes the `signal()` function, and should be used in preference. In particular, `sigaction()` and `signal()` should not be used in the same process to control the same signal. The behavior of async-signal-safe functions, as defined in their respective DESCRIPTION sections, is as specified by this volume of POSIX.1-2024, regardless of invocation from a signal-catching function. This is the only intended meaning of the statement that async-signal-safe functions may be used in signal-catching functions without restrictions. Applications must still consider all effects of such functions on such things as data structures, files, and process state. In particular, application developers need to consider the restrictions on interactions when interrupting `sleep()` and interactions among multiple handles for a file description. The fact that any specific function is listed as async-signal-safe does not necessarily mean that invocation of that function from a signal-catching function is recommended.

In order to prevent errors arising from interrupting non-async-signal-safe 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, should save and restore its value in order to avoid the possibility that delivery of a signal in between an error return from a function that sets `errno` and the subsequent examination of `errno` could result in the signal-catching function changing the value of `errno`. Naturally, the same principles apply to the async-signal-safety of application routines and asynchronous data access. Note that `longjmp()` and `siglongjmp()` are not in the list of async-signal-safe functions. This is because the code executing after `longjmp()` and `siglongjmp()` can call any unsafe functions with the same danger as calling those unsafe functions directly from the signal handler. Applications that use `longjmp()` and `siglongjmp()` from within signal handlers require rigorous protection in order to be portable. Many of the other functions that are excluded from the list are traditionally implemented using either

`malloc()` or `free()` functions or the standard I/O library, both of which traditionally use data structures in a non-async-signal-safe manner. Since any combination of different functions using a common data structure can cause async-signal-safety problems, this volume of POSIX.1-2024 does not define the behavior when any unsafe function is called in a signal handler that interrupts an unsafe function.

Usually, the signal is executed on the stack that was in effect before the signal was delivered. An alternate stack may be specified to receive a subset of the signals being caught.

When the signal handler returns, the receiving thread resumes execution at the point it was interrupted unless the signal handler makes other arrangements. If `longjmp()` is used to leave the signal handler, then the signal mask must be explicitly restored.

This volume of POSIX.1-2024 defines the third argument of a signal handling function when `SA_SIGINFO` is set as a `void *` instead of a `ucontext_t *`, but without requiring type checking. New applications should explicitly cast the third argument of the signal handling function to `ucontext_t *`.

The BSD optional four argument signal handling function is not supported by this volume of POSIX.1-2024. 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.

Since the `sigaction()` function is allowed but not required to set `SA_NODEFER` when the application sets the `SA_RESETHAND` flag, applications which depend on the `SA_RESETHAND` functionality for the newly installed signal handler must always explicitly set `SA_NODEFER` when they set `SA_RESETHAND` in order to be portable.

See also the rationale for Realtime Signal Generation and Delivery in XRAT [B.2.4.2 Realtime Signal Generation and Delivery](#).

## RATIONALE

Although this volume of POSIX.1-2024 requires that signals that cannot be ignored shall not be added to the signal mask when a signal-catching function is entered,

there is no explicit requirement that subsequent calls to `sigaction()` reflect this in the information returned in the `oact` argument. In other words, if SIGKILL is included in the `sa_mask` field of `act`, it is unspecified whether or not a subsequent call to `sigaction()` returns with SIGKILL included in the `sa_mask` field of `oact`.

The `SA_NOCLDSTOP` flag, when supplied in the `act->sa_flags` parameter, allows overloading `SIGCHLD` with the System V semantics that each `SIGCLD` signal indicates a single terminated child. Most conforming applications that catch `SIGCHLD` are expected to install signal-catching functions that repeatedly call the `waitpid()` function with the `WNOHANG` flag set, acting on each child for which status is returned, until `waitpid()` returns zero. If stopped children are not of interest, the use of the `SA_NOCLDSTOP` flag can prevent the overhead from invoking the signal-catching routine when they stop.

Some historical implementations also define other mechanisms for stopping processes, such as the `ptrace()` function. These implementations usually do...

## 1.209. sigaddset

---

### SYNOPSIS

```
#include <signal.h>

int sigaddset(sigset_t *set, int signo);
```

### DESCRIPTION

The `sigaddset()` function adds the individual signal specified by the `signo` to the signal set pointed to by `set`.

Applications shall call either `sigemptyset()` or `sigfillset()` at least once for each object of type `sigset_t` prior to any other use of that object. If such an object is not initialized in this way, but is nonetheless supplied as an argument to any of `pthread_sigmask()`, `sigaction()`, `sigaddset()`, `sigdelset()`, `sigismember()`, `sigpending()`, `sigprocmask()`, `sigsuspend()`, `sigtimedwait()`, `sigwait()`, or `sigwaitinfo()`, the results are undefined.

### RETURN VALUE

Upon successful completion, `sigaddset()` shall return 0; otherwise, it shall return -1 and set `errno` to indicate the error.

### ERRORS

The `sigaddset()` function may fail if:

- **[EINVAL]**

The value of the `signo` argument is an invalid or unsupported signal number.

*The following sections are informative.*

### EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [2.4 Signal Concepts](#)
- [pthread\\_sigmask\(\)](#)
- [sigaction\(\)](#)
- [sigdelset\(\)](#)
- [sigemptyset\(\)](#)
- [sigfillset\(\)](#)
- [sigismember\(\)](#)
- [sigpending\(\)](#)
- [sigsuspend\(\)](#)
- XBD

## CHANGE HISTORY

First released in Issue 3.

Included for alignment with the POSIX.1-1988 standard.

### Issue 5

The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in previous issues.

## Issue 6

The normative text is updated to avoid use of the term "must" for application requirements.

The SYNOPSIS is marked CX since the presence of this function in the header is an extension over the ISO C standard.

---

## 1.210. `sigdelset` - delete a signal from a signal set

---

### SYNOPSIS

```
#include <signal.h>

int sigdelset(sigset_t *set, int signo);
```

### DESCRIPTION

The `sigdelset()` function deletes the individual signal specified by `signo` from the signal set pointed to by `set`.

Applications should call either `sigemptyset()` or `sigfillset()` at least once for each object of type `sigset_t` prior to any other use of that object. If such an object is not initialized in this way, but is nonetheless supplied as an argument to any of `pthread_sigmask()`, `sigaction()`, `sigaddset()`, `sigdelset()`, `sigismember()`, `sigpending()`, `sigprocmask()`, `sigsuspend()`, `sigtimedwait()`, `sigwait()`, or `sigwaitinfo()`, the results are undefined.

### RETURN VALUE

Upon successful completion, `sigdelset()` shall return 0; otherwise, it shall return -1 and set `errno` to indicate the error.

### ERRORS

The `sigdelset()` function may fail if:

**[EINVAL]**

The `signo` argument is not a valid signal number, or is an unsupported signal number.

### EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [Signal Concepts](#)
- [pthread\\_sigmask\(\)](#)
- [sigaction\(\)](#)
- [sigaddset\(\)](#)
- [sigemptyset\(\)](#)
- [sigfillset\(\)](#)
- [sigismember\(\)](#)
- [sigpending\(\)](#)
- [sigsuspend\(\)](#)
- 

## CHANGE HISTORY

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

### Issue 5

The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in previous issues.

## Issue 6

The SYNOPSIS is marked CX since the presence of this function in the `<signal.h>` header is an extension over the ISO C standard.

---

## 1.211. sigemptyset

---

### SYNOPSIS

```
[CX] #include <signal.h>

int sigemptyset(sigset_t *set);
```

### DESCRIPTION

The `sigemptyset()` function initializes the signal set pointed to by `set`, such that all signals defined in POSIX.1-2024 are excluded.

### RETURN VALUE

Upon successful completion, `sigemptyset()` shall return 0; otherwise, it shall return -1 and set `errno` to indicate the error.

### ERRORS

No errors are defined.

---

*The following sections are informative.*

### EXAMPLES

None.

### APPLICATION USAGE

None.

## RATIONALE

The implementation of the `sigemptyset()` (or `sigfillset()`) function could quite trivially clear (or set) all the bits in the signal set. Alternatively, it would be reasonable to initialize part of the structure, such as a version field, to permit binary-compatibility between releases where the size of the set varies. For such reasons, either `sigemptyset()` or `sigfillset()` must be called prior to any other use of the signal set, even if such use is read-only (for example, as an argument to `sigpending()`). This function is not intended for dynamic allocation.

The `sigfillset()` and `sigemptyset()` functions require that the resulting signal set include (or exclude) all the signals defined in this volume of POSIX.1-2024. Although it is outside the scope of this volume of POSIX.1-2024 to place this requirement on signals that are implemented as extensions, it is recommended that implementation-defined signals also be affected by these functions. However, there may be a good reason for a particular signal not to be affected. For example, blocking or ignoring an implementation-defined signal may have undesirable side-effects, whereas the default action for that signal is harmless. In such a case, it would be preferable for such a signal to be excluded from the signal set returned by `sigfillset()`.

In early proposals there was no distinction between invalid and unsupported signals (the names of optional signals that were not supported by an implementation were not defined by that implementation). The [EINVAL] error was thus specified as a required error for invalid signals. With that distinction, it is not necessary to require implementations of these functions to determine whether an optional signal is actually supported, as that could have a significant performance impact for little value. The error could have been required for invalid signals and optional for unsupported signals, but this seemed unnecessarily complex. Thus, the error is optional in both cases.

## FUTURE DIRECTIONS

None.

## SEE ALSO

2.4 Signal Concepts, `pthread_sigmask()`, `sigaction()`, `sigaddset()`,  
`sigdelset()`, `sigfillset()`, `sigismember()`, `sigpending()`,  
`sigsuspend()`

XBD `<signal.h>`

# CHANGE HISTORY

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

## Issue 6

The SYNOPSIS is marked CX since the presence of this function in the `<signal.h>` header is an extension over the ISO C standard.

---

## 1.212. `sigfillset` - initialize and fill a signal set

---

### SYNOPSIS

```
#include <signal.h>

int sigfillset(sigset_t *set);
```

### DESCRIPTION

The `sigfillset()` function shall initialize the signal set pointed to by `set`, such that all signals defined in this volume of POSIX.1-2024 are included.

### RETURN VALUE

Upon successful completion, `sigfillset()` shall return 0; otherwise, it shall return -1 and set `errno` to indicate the error.

### ERRORS

No errors are defined.

*The following sections are informative.*

### EXAMPLES

None.

### APPLICATION USAGE

None.

### RATIONALE

Refer to `sigemptyset()`.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- 2.4 Signal Concepts
- `pthread_sigmask()`
- `sigaction()`
- `sigaddset()`
- `sigdelset()`
- `sigemptyset()`
- `sigismember()`
- `sigpending()`
- `sigsuspend()`
- XBD `<signal.h>`

## CHANGE HISTORY

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

### Issue 6

The SYNOPSIS is marked CX since the presence of this function in the `<signal.h>` header is an extension over the ISO C standard.

## 1.213. sigismember

---

### SYNOPSIS

```
#include <signal.h>

int sigismember(const sigset_t *set, int signo);
```

### DESCRIPTION

The `sigismember()` function shall test whether the signal specified by `signo` is a member of the set pointed to by `set`.

Applications should call either `sigemptyset()` or `sigfillset()` at least once for each object of type `sigset_t` prior to any other use of that object. If such an object is not initialized in this way, but is nonetheless supplied as an argument to any of `pthread_sigmask()`, `sigaction()`, `sigaddset()`, `sigdelset()`, `sigismember()`, `sigpending()`, `sigprocmask()`, `sigsuspend()`, `sigtimedwait()`, `sigwait()`, or `sigwaitinfo()`, the results are undefined.

### RETURN VALUE

Upon successful completion, `sigismember()` shall return 1 if the specified signal is a member of the specified set, or 0 if it is not. Otherwise, it shall return -1 and set `errno` to indicate the error.

### ERRORS

The `sigismember()` function may fail if:

- **[EINVAL]**

The `signo` argument is not a valid signal number, or is an unsupported signal number.

### EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [2.4 Signal Concepts](#)
- [pthread\\_sigmask\(\)](#)
- [sigaction\(\)](#)
- [sigaddset\(\)](#)
- [sigdelset\(\)](#)
- [sigfillset\(\)](#)
- [sigemptyset\(\)](#)
- [sigpending\(\)](#)
- [sigsuspend\(\)](#)
- 

## CHANGE HISTORY

### First released in Issue 3

Included for alignment with the POSIX.1-1988 standard.

### Issue 5

The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in previous issues.

## Issue 6

The SYNOPSIS is marked CX since the presence of this function in the `<signal.h>` header is an extension over the ISO C standard.

---

## 1.214. signal — signal management

---

### SYNOPSIS

```
#include <signal.h>

void (*signal(int sig, void (*func)(int)))(int);
```

### DESCRIPTION

The `signal()` function chooses one of three ways in which receipt of the signal number `sig` is to be subsequently handled. If the value of `func` is `SIG_DFL`, default handling for that signal shall occur. If the value of `func` is `SIG_IGN`, the signal shall be ignored. Otherwise, the application shall ensure that `func` points to a function to be called when that signal occurs. An invocation of such a function because of a signal, or (recursively) of any further functions called by that invocation (other than functions in the standard library), is called a "signal handler".

When a signal occurs, and `func` points to a function, it is implementation-defined whether the equivalent of a:

```
signal(sig, SIG_DFL);
```

is executed or the implementation prevents some implementation-defined set of signals (at least including `sig`) from occurring until the current signal handling has completed. (If the value of `sig` is `SIGILL`, the implementation may alternatively define that no action is taken.) Next the equivalent of:

```
(*func)(sig);
```

is executed. If and when the function returns, if the value of `sig` was `SIGFPE`, `SIGILL`, or `SIGSEGV` or any other implementation-defined value corresponding to a computational exception, the behavior is undefined. Otherwise, the program shall resume execution at the point it was interrupted.

If the process is multi-threaded, or if the process is single-threaded and a signal handler is executed other than as the result of:

- The process calling `abort()`, `raise()`, `kill()`, `pthread_kill()`, or `sigqueue()` to generate a signal that is not blocked
- A pending signal being unblocked and being delivered before the call that unblocked it returns

the behavior is undefined if:

[Additional behavior conditions continue in the full specification...]

At program start-up, the equivalent of:

```
signal(sig, SIG_IGN);
```

is executed for some signals, and the equivalent of:

```
signal(sig, SIG_DFL);
```

is executed for all other signals (see `exec`).

The `signal()` function shall not change the setting of `errno` if successful.

The `signal()` function is required to be thread-safe.

## RETURN VALUE

If the request can be honored, `signal()` shall return the value of `func` for the most recent call to `signal()` for the specified signal `sig`. Otherwise, `SIG_ERR` shall be returned and a positive value shall be stored in `errno`.

## ERRORS

The `signal()` function shall fail if:

- **[EINVAL]** The `sig` argument is not a valid signal number or an attempt is made to catch a signal that cannot be caught or ignore a signal that cannot be ignored.

The `signal()` function may fail if:

- **[EINVAL]** An attempt was made to set the action to `SIG_DFL` for a signal that cannot be caught or ignored (or both).

## APPLICATION USAGE

The `sigaction()` function provides a more comprehensive and reliable mechanism for controlling signals; new applications should use `sigaction()` rather than `signal()`.

## RATIONALE

The ISO C standard says that the use of `signal()` in a multi-threaded program results in undefined behavior. However, POSIX.1 has required `signal()` to be thread-safe since before threads were added to the ISO C standard.

## SEE ALSO

- [2.4 Signal Concepts](#)
- [exec](#)
- [pause\(\)](#)
- [raise\(\)](#)
- [sigaction\(\)](#)
- [sigsuspend\(\)](#)
- [waitid\(\)](#)
- 

## CHANGE HISTORY

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

### Issue 5

Moved from X/OPEN UNIX extension to BASE.

### Issue 6

Extensions beyond the ISO C standard are marked.

### Issue 7

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0580 [275], XSH/TC1-2008/0581 [66], and XSH/TC1-2008/0582 [105] are applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0331 [785] is applied.

## Issue 8

Austin Group Defect 728 is applied, reducing the set of circumstances in which undefined behavior results when a signal handler refers to an object with static or thread storage duration.

Austin Group Defect 1302 is applied, aligning this function with the ISO/IEC 9899:2018 standard.

---

## 1.215. sigpending

---

### SYNOPSIS

```
#include <signal.h>

int sigpending(sigset_t *set);
```

### DESCRIPTION

The `sigpending()` function shall store, in the location referenced by the `set` argument, the set of signals that are blocked from delivery to the calling thread and that are pending on the process or the calling thread.

### RETURN VALUE

Upon successful completion, `sigpending()` shall return 0; otherwise, -1 shall be returned and `errno` set to indicate the error.

### ERRORS

No errors are defined.

*The following sections are informative.*

### EXAMPLES

None.

### APPLICATION USAGE

None.

### RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

[exec](#),  
[pthread\\_sigmask\(\)](#),  
[sigaddset\(\)](#),  
[sigdelset\(\)](#),  
[sigemptyset\(\)](#),  
[sigfillset\(\)](#),  
[sigismember\(\)](#)

XBD [`<signal.h>`](#)

## CHANGE HISTORY

First released in Issue 3.

### Issue 5

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

### Issue 6

The SYNOPSIS is marked CX since the presence of this function in the [`<signal.h>`](#) header is an extension over the ISO C standard.

## 1.216. sigprocmask

---

### SYNOPSIS

```
#include <signal.h>

int pthread_sigmask(int how, const sigset_t *restrict set,
                     sigset_t *restrict oset);
int sigprocmask(int how, const sigset_t *restrict set,
                 sigset_t *restrict oset);
```

### DESCRIPTION

The `pthread_sigmask()` function shall examine or change (or both) the calling thread's signal mask.

If the argument `set` is not a null pointer, it points to a set of signals to be used to change the currently blocked set.

The argument `how` indicates the way in which the set is changed, and the application shall ensure it consists of one of the following values:

#### **SIG\_BLOCK**

The resulting set shall be the union of the current set and the signal set pointed to by `set`.

#### **SIG\_SETMASK**

The resulting set shall be the signal set pointed to by `set`.

#### **SIG\_UNBLOCK**

The resulting set shall be the intersection of the current set and the complement of the signal set pointed to by `set`.

If the argument `oset` is not a null pointer, the previous mask shall be stored in the location pointed to by `oset`. If `set` is a null pointer, the value of the argument `how` is not significant and the thread's signal mask shall be unchanged; thus the call can be used to enquire about currently blocked signals.

If the argument `set` is not a null pointer, after `pthread_sigmask()` changes the currently blocked set of signals it shall determine whether there are any pending unblocked signals; if there are any, then at least one of those signals shall be delivered before the call to `pthread_sigmask()` returns.

It is not possible to block those signals which cannot be ignored. This shall be enforced by the system without causing an error to be indicated.

If any of the SIGFPE, SIGILL, SIGSEGV, or SIGBUS signals are generated while they are blocked, the result is undefined, unless the signal was generated by the action of another process, or by one of the functions `kill()`, `pthread_kill()`, `raise()`, or `sigqueue()`.

If `pthread_sigmask()` fails, the thread's signal mask shall not be changed.

The `sigprocmask()` function shall be equivalent to `pthread_sigmask()`, except that its behavior is unspecified if called from a multi-threaded process, and on error it returns -1 and sets `errno` to the error number instead of returning the error number directly.

## RETURN VALUE

Upon successful completion, `pthread_sigmask()` shall return 0; otherwise, it shall return the corresponding error number.

Upon successful completion, `sigprocmask()` shall return 0; otherwise, -1 shall be returned and `errno` shall be set to indicate the error.

## ERRORS

These functions shall fail if:

### [EINVAL]

The `set` argument is not a null pointer and the value of the `how` argument is not equal to one of the defined values.

These functions shall not return an error code of [EINTR].

## EXAMPLES

### Signaling in a Multi-Threaded Process

This example shows the use of `pthread_sigmask()` in order to deal with signals in a multi-threaded process. It provides a fairly general framework that could be easily adapted/extended.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <signal.h>
#include <string.h>
#include <errno.h>
```

```

...
static sigset_t    signal_mask; /* signals to block           */

int main (int argc, char *argv[])
{
    pthread_t  sig_thr_id;      /* signal handler thread ID */
    int         rc;             /* return code               */

    sigemptyset (&signal_mask);
    sigaddset (&signal_mask, SIGINT);
    sigaddset (&signal_mask, SIGTERM);
    rc = pthread_sigmask (SIG_BLOCK, &signal_mask, NULL);
    if (rc != 0) {
        /* handle error */
        ...
    }
    /* any newly created threads inherit the signal mask */

    rc = pthread_create (&sig_thr_id, NULL, signal_thread, NULL);
    if (rc != 0) {
        /* handle error */
        ...
    }
    /* APPLICATION CODE */
    ...
}

void *signal_thread (void *arg)
{
    int      sig_caught;      /* signal caught           */
    int      rc;               /* returned code           */

    rc = sigwait (&signal_mask, &sig_caught);
    if (rc != 0) {
        /* handle error */
    }
    switch (sig_caught)
    {
    case SIGINT:    /* process SIGINT   */
        ...
        break;
    case SIGTERM:   /* process SIGTERM  */
        ...
        break;
    default:        /* should normally not happen */
        fprintf (stderr, "\nUnexpected signal %d\n", sig_caught);
        break;
    }
}

```

```
    }  
}
```

## APPLICATION USAGE

Although `pthread_sigmask()` has to deliver at least one of any pending unblocked signals that exist after it has changed the currently blocked set of signals, there is no requirement that the delivered signal(s) include any that were unblocked by the change. If one or more signals that were already unblocked become pending (see [2.4.1 Signal Generation and Delivery](#)) during the period the `pthread_sigmask()` call is executing, the signal(s) delivered before the call returns might include only those signals.

## RATIONALE

When a thread's signal mask is changed in a signal-catching function that is installed by `sigaction()`, the restoration of the signal mask on return from the signal-catching function overrides that change (see [sigaction\(\)](#)). If the signal-catching function was installed with `signal()`, it is unspecified whether this occurs.

See `kill()` for a discussion of the requirement on delivery of signals.

## FUTURE DIRECTIONS

None.

## SEE ALSO

`exec` , `kill()` , `sigaction()` , `sigaddset()` , `sigdelset()` ,  
`sigemptyset()` , `sigfillset()` , `sigismember()` , `sigpending()` ,  
`sigqueue()` , `sigsuspend()`

XBD `<signal.h>`

## CHANGE HISTORY

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

## Issue 5

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

The `pthread_sigmask()` function is added for alignment with the POSIX Threads Extension.

## Issue 6

The `pthread_sigmask()` function is marked as part of the Threads option.

The SYNOPSIS for `sigprocmask()` is marked as a CX extension to note that the presence of this function in the `<signal.h>` header is an extension to the ISO C standard.

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

- The DESCRIPTION is updated to explicitly state the functions which may generate the signal.

The normative text is updated to avoid use of the term "must" for application requirements.

The `restrict` keyword is added to the `pthread_sigmask()` and `sigprocmask()` prototypes for alignment with the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/105 is applied, updating "process' signal mask" to "thread's signal mask" in the DESCRIPTION and RATIONALE sections.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/106 is applied, adding the example to the EXAMPLES section.

## Issue 7

The `pthread_sigmask()` function is moved from the Threads option to the Base.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0467 [319] is applied.

## Issue 8

Austin Group Defect 1132 is applied, clarifying the [EINVAL] error.

Austin Group Defect 1636 is applied, clarifying the exceptions to the equivalence of `pthread_sigmask()` and `sigprocmask()`.

Austin Group Defect 1731 is applied, clarifying that although `pthread_sigmask()` has to deliver at least one of any pending unblocked

signals that exist after it has changed the currently blocked set of signals, there is no requirement that the delivered signal(s) include any that were unblocked by the change.

---

## 1.217. sigqueue

---

### SYNOPSIS

```
#include <signal.h>

int sigqueue(pid_t pid, int signo, union sigval value);
```

### DESCRIPTION

The `sigqueue()` function shall cause the signal specified by `signo` to be sent with the value specified by `value` to the process specified by `pid`. If `signo` is zero (the null signal), error checking is performed but no signal is actually sent. The null signal can be used to check the validity of `pid`.

The conditions required for a process to have permission to queue a signal to another process are the same as for the `kill()` function.

The `sigqueue()` function shall return immediately. If `SA_SIGINFO` is set for `signo` and if the resources were available to queue the signal, the signal shall be queued and sent to the receiving process. If `SA_SIGINFO` is not set for `signo`, then `signo` shall be sent at least once to the receiving process; it is unspecified whether `value` shall be sent to the receiving process as a result of this call.

If the value of `pid` causes `signo` to be generated for the sending process, and if `signo` is not blocked for the calling thread and if no other thread has `signo` unblocked or is waiting in a `sigwait()` function for `signo`, either `signo` or at least the pending, unblocked signal shall be delivered to the calling thread before the `sigqueue()` function returns. Should any multiple pending signals in the range `SIGRTMIN` to `SIGRTMAX` be selected for delivery, it shall be the lowest numbered one. The selection order between realtime and non-realtime signals, or between multiple pending non-realtime signals, is unspecified.

### RETURN VALUE

Upon successful completion, the specified signal shall have been queued, and the `sigqueue()` function shall return a value of zero. Otherwise, the function shall return a value of -1 and set `errno` to indicate the error.

# ERRORS

The `sigqueue()` function shall fail if:

- **[EAGAIN]**
  - No resources are available to queue the signal. The process has already queued {SIGQUEUE\_MAX} signals that are still pending at the receiver(s), or a system-wide resource limit has been exceeded.
  - **[EINVAL]**
  - The value of the `signo` argument is an invalid or unsupported signal number.
  - **[EPERM]**
  - The process does not have appropriate privileges to send the signal to the receiving process.
  - **[ESRCH]**
  - The process `pid` does not exist.
- 

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

The `sigqueue()` function allows an application to queue a realtime signal to itself or to another process, specifying the application-defined value. This is common practice in realtime applications on existing realtime systems. It was felt that specifying another function in the `sig` ... name space already carved out for signals was preferable to extending the interface to `kill()`.

Such a function became necessary when the put/get event function of the message queues was removed. It should be noted that the `sigqueue()` function implies reduced performance in a security-conscious implementation as the access permissions between the sender and receiver have to be checked on each send when the `pid` is resolved into a target process. Such access checks were necessary only at message queue open in the previous interface.

The standard developers required that `sigqueue()` have the same semantics with respect to the null signal as `kill()`, and that the same permission checking be used. But because of the difficulty of implementing the "broadcast" semantic of `kill()` (for example, to process groups) and the interaction with resource allocation, this semantic was not adopted. The `sigqueue()` function queues a signal to a single process specified by the `pid` argument.

The `sigqueue()` function can fail if the system has insufficient resources to queue the signal. An explicit limit on the number of queued signals that a process could send was introduced. While the limit is "per-sender", this volume of POSIX.1-2024 does not specify that the resources be part of the state of the sender. This would require either that the sender be maintained after exit until all signals that it had sent to other processes were handled or that all such signals that had not yet been acted upon be removed from the queue(s) of the receivers. This volume of POSIX.1-2024 does not preclude this behavior, but an implementation that allocated queuing resources from a system-wide pool (with per-sender limits) and that leaves queued signals pending after the sender exits is also permitted.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [2.8.1 Realtime Signals](#)
- [XBD](#)

## CHANGE HISTORY

### First released in Issue 5

Included for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

## Issue 6

The `sigqueue()` function is marked as part of the Realtime Signals Extension option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Realtime Signals Extension option.

## Issue 7

The `sigqueue()` function is moved from the Realtime Signals Extension option to the Base.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0332 [844] is applied.

## 1.218. `sigsuspend` — wait for a signal

---

### SYNOPSIS

```
#include <signal.h>

int sigsuspend(const sigset_t *sigmask);
```

### DESCRIPTION

The `sigsuspend()` function shall atomically both replace the current signal mask of the calling thread with the set of signals pointed to by `sigmask` and suspend the thread until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process. This shall not cause any other signals that may have been pending on the process to become pending on the thread.

If the action is to terminate the process then `sigsuspend()` shall never return. If the action is to execute a signal-catching function, then `sigsuspend()` shall return after the signal-catching function returns, with the signal mask restored to the set that existed prior to the `sigsuspend()` call.

It is not possible to block signals that cannot be ignored. This is enforced by the system without causing an error to be indicated.

### RETURN VALUE

Since `sigsuspend()` suspends thread execution indefinitely, there is no successful completion return value. If a return occurs, -1 shall be returned and `errno` set to indicate the error.

### ERRORS

The `sigsuspend()` function shall fail if:

- **[EINTR]** - A signal is caught by the calling process and control is returned from the signal-catching function.

## APPLICATION USAGE

Normally, at the beginning of a critical code section, a specified set of signals is blocked using the `sigprocmask()` function. When the thread has completed the critical section and needs to wait for the previously blocked signal(s), it pauses by calling `sigsuspend()` with the mask that was returned by the `sigprocmask()` call.

## RATIONALE

Code which wants to avoid the ambiguity of the signal mask for thread cancellation handlers can install an additional cancellation handler which resets the signal mask to the expected value.

```
void cleanup(void *arg)
{
    sigset_t *ss = (sigset_t *) arg;
    pthread_sigmask(SIG_SETMASK, ss, NULL);
}

int call_sigsuspend(const sigset_t *mask)
{
    sigset_t oldmask;
    int result;
    pthread_sigmask(SIG_SETMASK, NULL, &oldmask);
    pthread_cleanup_push(cleanup, &oldmask);
    result = sigsuspend(mask);
    pthread_cleanup_pop(0);
    return result;
}
```

## SEE ALSO

- [2.4 Signal Concepts](#)
- [pause\(\)](#)
- [sigaction\(\)](#)
- [sigaddset\(\)](#)
- [sigdelset\(\)](#)
- [sigemptyset\(\)](#)
- [sigfillset\(\)](#)

- 

## CHANGE HISTORY

### First released in Issue 3

Included for alignment with the POSIX.1-1988 standard.

### Issue 5

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

### Issue 6

- The text in the RETURN VALUE section has been changed from "suspends process execution" to "suspends thread execution". This reflects IEEE PASC Interpretation 1003.1c #40.
- Text in the APPLICATION USAGE section has been replaced.
- The SYNOPSIS is marked CX since the presence of this function in the header is an extension over the ISO C standard.

### Issue 7

SD5-XSH-ERN-122 is applied, adding the example code in the RATIONALE.

### Issue 8

- Austin Group Defect 1201 is applied, clarifying the atomicity requirements for `sigsuspend()`.
  - Austin Group Defect 1223 is applied, changing the example code in the RATIONALE.
-

## 1.219. `sigtimedwait`, `sigwaitinfo`

---

### SYNOPSIS

```
#include <signal.h>

int sigtimedwait(const sigset_t *restrict set,
                  siginfo_t *restrict info,
                  const struct timespec *restrict timeout);

int sigwaitinfo(const sigset_t *restrict set,
                siginfo_t *restrict info);
```

### DESCRIPTION

The `sigtimedwait()` function shall be equivalent to `sigwaitinfo()` except that if none of the signals specified by *set* are pending, `sigtimedwait()` shall wait for the time interval specified in the **timespec** structure referenced by *timeout*. If the **timespec** structure pointed to by *timeout* is zero-valued and if none of the signals specified by *set* are pending, then `sigtimedwait()` shall return immediately with an error. If *timeout* is the null pointer, the behavior is unspecified. The **CLOCK\_MONOTONIC** clock shall be used to measure the time interval specified by the *timeout* argument.

The `sigwaitinfo()` function selects the pending signal from the set specified by *set*. Should any of multiple pending signals in the range **SIGRTMIN** to **SIGRTMAX** be selected, it shall be the lowest numbered one. The selection order between realtime and non-realtime signals, or between multiple pending non-realtime signals, is unspecified. If no signal in *set* is pending at the time of the call, the calling thread shall be suspended until one or more signals in *set* become pending or until it is interrupted by an unblocked, caught signal.

The `sigwaitinfo()` function shall be equivalent to the `sigwait()` function, except that the return value and the error reporting method are different (see RETURN VALUE), and that if the *info* argument is non-NULL, the selected signal number shall be stored in the *si\_signo* member, and the cause of the signal shall be stored in the *si\_code* member. If any value is queued to the selected signal, the first such queued value shall be dequeued and, if the *info* argument is non-NULL, the value shall be stored in the *si\_value* member of *info*. The system resource used to queue the signal shall be released and returned to the system for other use. If no value is queued, the content of the *si\_value* member is undefined. If no

further signals are queued for the selected signal, the pending indication for that signal shall be reset.

## RETURN VALUE

Upon successful completion (that is, one of the signals specified by *set* is pending or is generated) `sigwaitinfo()` and `sigtimedwait()` shall return the selected signal number. Otherwise, the function shall return a value of -1 and set *errno* to indicate the error.

## ERRORS

The `sigtimedwait()` function shall fail if:

- **[EAGAIN]**
- No signal specified by *set* was generated within the specified timeout period.

The `sigtimedwait()` and `sigwaitinfo()` functions may fail if:

- **[EINTR]**
- The wait was interrupted by an unblocked, caught signal. It shall be documented in system documentation whether this error causes these functions to fail.

The `sigtimedwait()` function may also fail if:

- **[EINVAL]**
- The *timeout* argument specified a *tv\_nsec* value less than zero or greater than or equal to 1000 million.

An implementation should only check for this error if no signal is pending in *set* and it is necessary to wait.

## APPLICATION USAGE

The `sigtimedwait()` function times out and returns an [EAGAIN] error. Application developers should note that this is inconsistent with other functions such as `pthread_cond_timedwait()` that return [ETIMEDOUT].

Note that in order to ensure that generated signals are queued and signal values passed to `sigqueue()` are available in *si\_value*, applications which use `sigwaitinfo()` or `sigtimedwait()` need to set the SA\_SIGINFO flag for each signal in the set (see 2.4 *Signal Concepts*). This means setting each signal to

be handled by a three-argument signal-catching function, even if the handler will never be called. It is not possible (portably) to set a signal handler to SIG\_DFL while setting the SA\_SIGINFO flag, because assigning to the *sa\_handler* member of **struct sigaction** instead of the *sa\_sigaction* member would result in undefined behavior, and SIG\_DFL need not be assignment-compatible with *sa\_sigaction*. Even if an assignment of SIG\_DFL to *sa\_sigaction* is accepted by the compiler, the implementation need not treat this value as special—it could just be taken as the address of a signal-catching function.

## RATIONALE

Existing programming practice on realtime systems uses the ability to pause waiting for a selected set of events and handle the first event that occurs in-line instead of in a signal-handling function. This allows applications to be written in an event-directed style similar to a state machine. This style of programming is useful for largescale transaction processing in which the overall throughput of an application and the ability to clearly track states are more important than the ability to minimize the response time of individual event handling.

It is possible to construct a signal-waiting macro function out of the realtime signal function mechanism defined in this volume of POSIX.1-2024. However, such a macro has to include the definition of a generalized handler for all signals to be waited on. A significant portion of the overhead of handler processing can be avoided if the signal-waiting function is provided by the kernel. This volume of POSIX.1-2024 therefore provides two signal-waiting functions—one that waits indefinitely and one with a timeout—as part of the overall realtime signal function specification.

The specification of a function with a timeout allows an application to be written that can be broken out of a wait after a set period of time if no event has occurred. It was argued that setting a timer event before the wait and recognizing the timer event in the wait would also implement the same functionality, but at a lower performance level. Because of the performance degradation associated with the user-level specification of a timer event and the subsequent cancellation of that timer event after the wait completes for a valid event, and the complexity associated with handling potential race conditions associated with the user-level method, the separate function has been included.

Note that the semantics of the `sigwaitinfo()` function are nearly identical to that of the `sigwait()` function defined by this volume of POSIX.1-2024. The only difference is that `sigwaitinfo()` returns the queued signal value in the *value* argument. The return of the queued value is required so that applications can differentiate between multiple events queued to the same signal number.

The two distinct functions are being maintained because some implementations may choose to implement the POSIX Threads Extension functions and not implement the queued signals extensions. Note, though, that `sigwaitinfo()` does not return the queued value if the `value` argument is `NULL`, so the POSIX Threads Extension `sigwait()` function can be implemented as a macro on `sigwaitinfo()`.

The `sigtimedwait()` function was separated from the `sigwaitinfo()` function to address concerns regarding the overloading of the `timeout` pointer to indicate indefinite wait (no timeout), timed wait, and immediate return, and concerns regarding consistency with other functions where the conditional and timed waits were separate functions from the pure blocking function. The semantics of `sigtimedwait()` are specified such that `sigwaitinfo()` could be implemented as a macro with a null pointer for `timeout`.

The `sigwait` functions provide a synchronous mechanism for threads to wait for asynchronously-generated signals. One important question was how many threads that are suspended in a call to a `sigwait()` function for a signal should return from the call when the signal is sent. Four choices were considered:

1. Return an error for multiple simultaneous calls to `sigwait` functions for the same signal.
2. One or more threads return.
3. All waiting threads return.
4. Exactly one thread returns.

Prohibiting multiple calls to `sigwait()` for the same signal was felt to be overly restrictive. The "one or more" behavior made implementation of conforming packages easy at the expense of forcing POSIX threads clients to protect against multiple simultaneous calls to `sigwait()` in application code in order to achieve predictable behavior. There was concern that the "all waiting threads" behavior would result in "signal broadcast storms", consuming excessive CPU resources by replicating the signals in the general case. Furthermore, no convincing examples could be presented that delivery to all was either simpler or more powerful than delivery to one.

Thus, the consensus was that exactly one thread that was suspended in a call to a `sigwait` function for a signal should return when that signal occurs. This is not an onerous restriction as:

- A multi-way signal wait can be built from the single-way wait.
- Signals should only be handled by application-level code, as library routines cannot guess what the application wants to do with signals generated for the entire process.

- Applications can thus arrange for a single thread to wait for any given signal and call any needed routines upon its arrival.

In an application that is using signals for interprocess communication, signal processing is typically done in one place. Alternatively, if the signal is being caught so that process cleanup can be done, the signal handler thread can call separate process cleanup routines for each portion of the application. Since the application main line started each portion of the application, it is at the right abstraction level to tell each portion of the application to clean up.

Certainly, there exist programming styles where it is logical to consider waiting for a single signal in multiple threads. A simple `sigwait_multiple()` routine can be constructed to achieve this goal. A possible implementation would be to have each `sigwait_multiple()` caller registered as having expressed interest in a set of signals. The caller then waits on a thread-specific condition variable. A single server thread calls a `sigwait()` function on the union of all registered signals. When the `sigwait()` function returns, the appropriate state is set and condition variables are broadcast. New `sigwait_multiple()` callers may cause the pending `sigwait()` call to be canceled and reissued in order to update the set of signals being waited for.

## FUTURE DIRECTIONS

None.

## SEE ALSO

2.4 Signal Concepts, 2.8.1 Realtime Signals, `pause()`, `pthread_sigmask()`,  
`sigaction()`, `sigpending()`, `sigsuspend()`, `sigwait()`  
XBD `<signal.h>`, `<time.h>`

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

### Issue 6

These functions are marked as part of the Realtime Signals Extension option.

The Open Group Corrigendum U035/3 is applied. The SYNOPSIS of the `sigwaitinfo()` function has been corrected so that the second argument is of

type **siginfo\_t** \*.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Realtime Signals Extension option.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that the CLOCK\_MONOTONIC clock, if supported, is used to measure timeout intervals.

The **restrict** keyword is added to the `sigtimedwait()` and `sigwaitinfo()` prototypes for alignment with the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/130 is applied, restoring wording in the RETURN VALUE section to that in the original base document ("An implementation should only check for this error if no signal is pending in *set* and it is necessary to wait").

## Issue 7

The `sigtimedwait()` and `sigwaitinfo()` functions are moved from the Realtime Signals Extension option to the Base.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0583 [392] is applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0333 [815] is applied.

## Issue 8

Austin Group Defect 1346 is applied, requiring support for Monotonic Clock.

## 1.220. `sigwait` — wait for queued signals

---

### SYNOPSIS

```
#include <signal.h>

int sigwait(const sigset_t *restrict set, int *restrict sig);
```

### DESCRIPTION

The `sigwait()` function shall select a pending signal from `set`, atomically clear it from the system's set of pending signals, and return that signal number in the location referenced by `sig`. If prior to the call to `sigwait()` there are multiple pending instances of a single signal number, it is implementation-defined whether upon successful return there are any remaining pending signals for that signal number. If the implementation supports queued signals and there are multiple signals queued for the signal number selected, the first such queued signal shall cause a return from `sigwait()` and the remainder shall remain queued. If no signal in `set` is pending at the time of the call, the thread shall be suspended until one or more becomes pending. The signals defined by `set` shall have been blocked at the time of the call to `sigwait()`; otherwise, the behavior is undefined. The effect of `sigwait()` on the signal actions for the signals in `set` is unspecified.

If more than one thread is using `sigwait()` to wait for the same signal, no more than one of these threads shall return from `sigwait()` with the signal number. If more than a single thread is blocked in `sigwait()` for a signal when that signal is generated for the process, it is unspecified which of the waiting threads returns from `sigwait()`. If the signal is generated for a specific thread, as by `pthread_kill()`, only that thread shall return.

Should any of the multiple pending signals in the range SIGRTMIN to SIGRTMAX be selected, it shall be the lowest numbered one. The selection order between realtime and non-realtime signals, or between multiple pending non-realtime signals, is unspecified.

### RETURN VALUE

Upon successful completion, `sigwait()` shall store the signal number of the received signal at the location referenced by `sig` and return zero. Otherwise, an

error number shall be returned to indicate the error.

## ERRORS

The `sigwait()` function may fail if:

- **[EINVAL]** The `set` argument contains an invalid or unsupported signal number.

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

To provide a convenient way for a thread to wait for a signal, this volume of POSIX.1-2024 provides the `sigwait()` function. For most cases where a thread has to wait for a signal, the `sigwait()` function should be quite convenient, efficient, and adequate.

However, requests were made for a lower-level primitive than `sigwait()` and for semaphores that could be used by threads. After some consideration, threads were allowed to use semaphores and `sem_post()` was defined to be async-signal-safe.

In summary, when it is necessary for code run in response to an asynchronous signal to notify a thread, `sigwait()` should be used to handle the signal. Alternatively, if the implementation provides semaphores, they also can be used, either following `sigwait()` or from within a signal handling routine previously registered with `sigaction()`.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [Signal Concepts](#)
- [Realtime Signals](#)
- [pause\(\)](#)
- [pthread\\_sigmask\(\)](#)
- [sigaction\(\)](#)
- [sigpending\(\)](#)
- [sigsuspend\(\)](#)
- [sigtimedwait\(\)](#)
- [<signal.h>](#)
- [<time.h>](#)

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

### Issue 6

The **restrict** keyword is added to the [sigwait\(\)](#) prototype for alignment with the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/131 is applied, updating the DESCRIPTION to state that if more than a single thread is blocked in [sigwait\(\)](#), it is unspecified which of the waiting threads returns, and that if a signal is generated for a specific thread only that thread shall return.

### Issue 7

Functionality relating to the Realtime Signals Extension option is moved to the Base.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0584 [76] is applied.

---

## 1.221. `sigtimedwait`, `sigwaitinfo` — wait for queued signals

### SYNOPSIS

```
#include <signal.h>

int sigtimedwait(const sigset_t *restrict set,
                  siginfo_t *restrict info,
                  const struct timespec *restrict timeout);

int sigwaitinfo(const sigset_t *restrict set,
                siginfo_t *restrict info);
```

### DESCRIPTION

The `sigtimedwait()` function shall be equivalent to `sigwaitinfo()` except that if none of the signals specified by `set` are pending, `sigtimedwait()` shall wait for the time interval specified in the `timespec` structure referenced by `timeout`. If the `timespec` structure pointed to by `timeout` is zero-valued and if none of the signals specified by `set` are pending, then `sigtimedwait()` shall return immediately with an error. If `timeout` is the null pointer, the behavior is unspecified. The CLOCK\_MONOTONIC clock shall be used to measure the time interval specified by the `timeout` argument.

The `sigwaitinfo()` function selects the pending signal from the set specified by `set`. Should any of multiple pending signals in the range SIGRTMIN to SIGRTMAX be selected, it shall be the lowest numbered one. The selection order between realtime and non-realtime signals, or between multiple pending non-realtime signals, is unspecified. If no signal in `set` is pending at the time of the call, the calling thread shall be suspended until one or more signals in `set` become pending or until it is interrupted by an unblocked, caught signal.

The `sigwaitinfo()` function shall be equivalent to the `sigwait()` function, except that the return value and the error reporting method are different (see RETURN VALUE), and that if the `info` argument is non-NULL, the selected signal number shall be stored in the `si_signo` member, and the cause of the signal shall be stored in the `si_code` member. If any value is queued to the selected signal, the first such queued value shall be dequeued and, if the `info` argument is non-NULL, the value shall be stored in the `si_value` member of `info`. The system resource used to queue the signal shall be released and

returned to the system for other use. If no value is queued, the content of the `si_value` member is undefined. If no further signals are queued for the selected signal, the pending indication for that signal shall be reset.

## RETURN VALUE

Upon successful completion (that is, one of the signals specified by `set` is pending or is generated) `sigwaitinfo()` and `sigtimedwait()` shall return the selected signal number. Otherwise, the function shall return a value of -1 and set `errno` to indicate the error.

## ERRORS

The `sigtimedwait()` function shall fail if:

- **[EAGAIN]**
- No signal specified by `set` was generated within the specified timeout period.

The `sigtimedwait()` and `sigwaitinfo()` functions may fail if:

- **[EINTR]**
- The wait was interrupted by an unblocked, caught signal. It shall be documented in system documentation whether this error causes these functions to fail.

The `sigtimedwait()` function may also fail if:

- **[EINVAL]**
- The `timeout` argument specified a `tv_nsec` value less than zero or greater than or equal to 1000 million.

An implementation should only check for this error if no signal is pending in `set` and it is necessary to wait.

## APPLICATION USAGE

The `sigtimedwait()` function times out and returns an [EAGAIN] error. Application developers should note that this is inconsistent with other functions such as `pthread_cond_timedwait()` that return [ETIMEDOUT].

Note that in order to ensure that generated signals are queued and signal values passed to `sigqueue()` are available in `si_value`, applications which use

`sigwaitinfo()` or `sigtimedwait()` need to set the `SA_SIGINFO` flag for each signal in the set (see 2.4 Signal Concepts). This means setting each signal to be handled by a three-argument signal-catching function, even if the handler will never be called. It is not possible (portably) to set a signal handler to `SIG_DFL` while setting the `SA_SIGINFO` flag, because assigning to the `sa_handler` member of `struct sigaction` instead of the `sa_sigaction` member would result in undefined behavior, and `SIG_DFL` need not be assignment-compatible with `sa_sigaction`. Even if an assignment of `SIG_DFL` to `sa_sigaction` is accepted by the compiler, the implementation need not treat this value as special—it could just be taken as the address of a signal-catching function.

## RATIONALE

Existing programming practice on realtime systems uses the ability to pause waiting for a selected set of events and handle the first event that occurs in-line instead of in a signal-handling function. This allows applications to be written in an event-directed style similar to a state machine. This style of programming is useful for largescale transaction processing in which the overall throughput of an application and the ability to clearly track states are more important than the ability to minimize the response time of individual event handling.

It is possible to construct a signal-waiting macro function out of the realtime signal function mechanism defined in this volume of POSIX.1-2024. However, such a macro has to include the definition of a generalized handler for all signals to be waited on. A significant portion of the overhead of handler processing can be avoided if the signal-waiting function is provided by the kernel. This volume of POSIX.1-2024 therefore provides two signal-waiting functions—one that waits indefinitely and one with a timeout—as part of the overall realtime signal function specification.

The specification of a function with a timeout allows an application to be written that can be broken out of a wait after a set period of time if no event has occurred. It was argued that setting a timer event before the wait and recognizing the timer event in the wait would also implement the same functionality, but at a lower performance level. Because of the performance degradation associated with the user-level specification of a timer event and the subsequent cancellation of that timer event after the wait completes for a valid event, and the complexity associated with handling potential race conditions associated with the user-level method, the separate function has been included.

Note that the semantics of the `sigwaitinfo()` function are nearly identical to that of the `sigwait()` function defined by this volume of POSIX.1-2024. The only difference is that `sigwaitinfo()` returns the queued signal value in the `value`

argument. The return of the queued value is required so that applications can differentiate between multiple events queued to the same signal number.

The two distinct functions are being maintained because some implementations may choose to implement the POSIX Threads Extension functions and not implement the queued signals extensions. Note, though, that `sigwaitinfo()` does not return the queued value if the `value` argument is NULL, so the POSIX Threads Extension `sigwait()` function can be implemented as a macro on `sigwaitinfo()`.

The `sigtimedwait()` function was separated from the `sigwaitinfo()` function to address concerns regarding the overloading of the `timeout` pointer to indicate indefinite wait (no timeout), timed wait, and immediate return, and concerns regarding consistency with other functions where the conditional and timed waits were separate functions from the pure blocking function. The semantics of `sigtimedwait()` are specified such that `sigwaitinfo()` could be implemented as a macro with a null pointer for `timeout`.

The `sigwait` functions provide a synchronous mechanism for threads to wait for asynchronously-generated signals. One important question was how many threads that are suspended in a call to a `sigwait()` function for a signal should return from the call when the signal is sent. Four choices were considered:

1. Return an error for multiple simultaneous calls to `sigwait` functions for the same signal.
2. One or more threads return.
3. All waiting threads return.
4. Exactly one thread returns.

Prohibiting multiple calls to `sigwait()` for the same signal was felt to be overly restrictive. The "one or more" behavior made implementation of conforming packages easy at the expense of forcing POSIX threads clients to protect against multiple simultaneous calls to `sigwait()` in application code in order to achieve predictable behavior. There was concern that the "all waiting threads" behavior would result in "signal broadcast storms", consuming excessive CPU resources by replicating the signals in the general case. Furthermore, no convincing examples could be presented that delivery to all was either simpler or more powerful than delivery to one.

Thus, the consensus was that exactly one thread that was suspended in a call to a `sigwait` function for a signal should return when that signal occurs. This is not an onerous restriction as:

- A multi-way signal wait can be built from the single-way wait.

- Signals should only be handled by application-level code, as library routines cannot guess what the application wants to do with signals generated for the entire process.
- Applications can thus arrange for a single thread to wait for any given signal and call any needed routines upon its arrival.

In an application that is using signals for interprocess communication, signal processing is typically done in one place. Alternatively, if the signal is being caught so that process cleanup can be done, the signal handler thread can call separate process cleanup routines for each portion of the application. Since the application main line started each portion of the application, it is at the right abstraction level to tell each portion of the application to clean up.

Certainly, there exist programming styles where it is logical to consider waiting for a single signal in multiple threads. A simple `sigwait_multiple()` routine can be constructed to achieve this goal. A possible implementation would be to have each `sigwait_multiple()` caller registered as having expressed interest in a set of signals. The caller then waits on a thread-specific condition variable. A single server thread calls a `sigwait()` function on the union of all registered signals. When the `sigwait()` function returns, the appropriate state is set and condition variables are broadcast. New `sigwait_multiple()` callers may cause the pending `sigwait()` call to be canceled and reissued in order to update the set of signals being waited for.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- 2.4 Signal Concepts
- 2.8.1 Realtime Signals
- `pause()`
- `pthread_sigmask()`
- `sigaction()`
- `sigpending()`
- `sigsuspend()`
- `sigwait()`
- `<signal.h>`

- `<time.h>`

## CHANGE HISTORY

### First released in Issue 5

Included for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

### Issue 6

- These functions are marked as part of the Realtime Signals Extension option.
- The Open Group Corrigendum U035/3 is applied. The SYNOPSIS of the `sigwaitinfo()` function has been corrected so that the second argument is of type `siginfo_t *`.
- The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Realtime Signals Extension option.
- The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that the CLOCK\_MONOTONIC clock, if supported, is used to measure timeout intervals.
- The `restrict` keyword is added to the `sigtimedwait()` and `sigwaitinfo()` prototypes for alignment with the ISO/IEC 9899:1999 standard.
- IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/130 is applied, restoring wording in the RETURN VALUE section to that in the original base document ("An implementation should only check for this error if no signal is pending in `set` and it is necessary to wait").

### Issue 7

- The `sigtimedwait()` and `sigwaitinfo()` functions are moved from the Realtime Signals Extension option to the Base.
- POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0583 [392] is applied.
- POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0333 [815] is applied.

## Issue 8

- Austin Group Defect 1346 is applied, requiring support for Monotonic Clock.
-

## 1.222. `snprintf`, `asprintf`, `dprintf`, `fprintf`, `printf`, `sprintf` — print formatted output

### SYNOPSIS

```
#include <stdio.h>

[CX] int asprintf(char **restrict ptr, const char *restrict format
[CX] int dprintf(int fildes, const char *restrict format, ...);
int fprintf(FILE *restrict stream, const char *restrict format, ...
int printf(const char *restrict format, ...);
int snprintf(char *restrict s, size_t n,
             const char *restrict format, ...);
int sprintf(char *restrict s, const char *restrict format, ...);
```

### DESCRIPTION

[CX] Except for `asprintf()`, `dprintf()`, and the behavior of the `%lc` conversion when passed a null wide character, the functionality described on this reference page is aligned with the ISO C standard. Any other conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard for all `fprintf()`, `printf()`, `snprintf()`, and `sprintf()` functionality except in relation to the `%lc` conversion when passed a null wide character.

The `fprintf()` function shall place output on the named output `stream`. The `printf()` function shall place output on the standard output stream `stdout`. The `sprintf()` function shall place output followed by the null byte, '\0', in consecutive bytes starting at `*s`; it is the user's responsibility to ensure that enough space is available.

[CX] The `asprintf()` function shall be equivalent to `sprintf()`, except that the output string shall be written to dynamically allocated memory, allocated as if by a call to `malloc()`, of sufficient length to hold the resulting string, including a terminating null byte. If the call to `asprintf()` is successful, the address of this dynamically allocated string shall be stored in the location referenced by `ptr`.

The `dprintf()` function shall be equivalent to the `fprintf()` function, except that `dprintf()` shall write output to the file associated with the file descriptor specified by the `fildes` argument rather than place output on a stream.

The `snprintf()` function shall be equivalent to `sprintf()`, with the addition of the `n` argument which limits the number of bytes written to the buffer referred to by `s`. If `n` is zero, nothing shall be written and `s` may be a null pointer. Otherwise, output bytes beyond the `n`-1st shall be discarded instead of being written to the array, and a null byte is written at the end of the bytes actually written into the array.

If copying takes place between objects that overlap as a result of a call to `sprintf()` or `snprintf()`, the results are undefined.

Each of these functions converts, formats, and prints its arguments under control of the `format`. The application shall ensure that the format is a character string, beginning and ending in its initial shift state, if any. The format is composed of zero or more directives: **ordinary characters**, which are simply copied to the output stream, and **conversion specifications**, each of which shall result in the fetching of zero or more arguments. The results are undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments shall be evaluated but are otherwise ignored.

[CX] Conversions can be applied to the `n`-th argument after the `format` in the argument list, rather than to the next unused argument. In this case, the conversion specifier character `%` (see below) is replaced by the sequence `"%n$"`, where `n` is a decimal integer in the range `[1,{NL_ARGMAX}]`, giving the position of the argument in the argument list. This feature provides for the definition of format strings that select arguments in an order appropriate to specific languages (see the EXAMPLES section).

The `format` can contain either numbered argument conversion specifications (that is, those introduced by `"%n$"` and optionally containing the `"*m$"` forms of field width and precision), or unnumbered argument conversion specifications (that is, those introduced by the `%` character and optionally containing the `*` form of field width and precision), but not both. The only exception to this is that `%%` can be mixed with the `"%n$"` form. The results of mixing numbered and unnumbered argument specifications in a format string are undefined. When numbered argument specifications are used, specifying the `N`-th argument requires that all the leading arguments, from the first to the `(N-1)`th, are specified in the format string.

In format strings containing the `"%n$"` form of conversion specification, numbered arguments in the argument list can be referenced from the format string as many times as required.

In format strings containing the `%` form of conversion specification, each conversion specification uses the first unused argument in the argument list.

[CX] All forms of the `fprintf()` functions allow for the insertion of a language-dependent radix character in the output string. The radix character is defined in the

current locale (category `LC_NUMERIC`). In the POSIX locale, or in a locale where the radix character is not defined, the radix character shall default to a ('.')�

Each conversion specification is introduced by the '%' character [CX] or by the character sequence "%n\$", after which the following appear in sequence:

- Zero or more **flags** (in any order), which modify the meaning of the conversion specification.
- An optional minimum **field width**. If the converted value has fewer bytes than the field width, it shall be padded with characters by default on the left; it shall be padded on the right if the left-adjustment flag ('-'), described below, is given to the field width. The field width takes the form of an ('), [CX] or in conversion specifications introduced by "%n\$" the "m\$" string, described below, or a decimal integer.
- An optional **precision** that gives the minimum number of digits to appear for the d, i, o, u, x, and X conversion specifiers; the number of digits to appear after the radix character for the a, A, e, E, f, and F conversion specifiers; the maximum number of significant digits for the g and G conversion specifiers; or the maximum number of bytes to be printed from a string in the s [XSI] and S conversion specifiers. The precision takes the form of a ('.) followed either by an ('), [CX] or in conversion specifications introduced by "%n\$" the "m\$" string, described below, or an optional decimal digit string, where a null digit string is treated as zero. If a precision appears with any other conversion specifier, the behavior is undefined.
- An optional **length modifier** that specifies the size of the argument.
- A **conversion specifier** character that indicates the type of conversion to be applied.

A field width, or precision, or both, may be indicated by an ('). *In this case an argument of type `int` supplies the field width or precision. Applications shall ensure that arguments specifying field width, or precision, or both appear in that order before the argument, if any, to be converted. A negative field width is taken as a '-' flag followed by a positive field width. A negative precision is taken as if the precision were omitted.* [CX] *In format strings containing conversion specifications introduced by "%n\$", in addition to being indicated by the decimal digit string, a field width may be indicated by the sequence "m\$" and precision by the sequence ".\*m\$"*, where m is a decimal integer in the range [1,{NL\_ARGMAX}] giving the position in the argument list (after the `format` argument) of an integer argument containing the field width or precision, for example:

```
printf("%1$d:%2$.*3$d:%4$.*3$d\n", hour, min, precision, sec);
```

## Flag Characters

The flag characters and their meanings are:

**'** (apostrophe)

[CX] The integer portion of the result of a decimal conversion (%i, %d, %u, %f, %F, %g, or %G) shall be formatted with thousands' grouping characters. For other conversions the behavior is undefined. The non-monetary grouping character is used.

-

The result of the conversion shall be left-justified within the field. The conversion is right-justified if this flag is not specified.

**+**

The result of a signed conversion shall always begin with a sign ('+' or '-'). The conversion shall begin with a sign only when a negative value is converted if this flag is not specified.

If the first character of a signed conversion is not a sign or if a signed conversion results in no characters, a shall be prefixed to the result. This means that if the and '+' flags both appear, the flag shall be ignored.

**#**

Specifies that the value is to be converted to an alternative form. For o conversion, it shall increase the precision, if and only if necessary, to force the first digit of the result to be a zero (if the value and precision are both 0, a single 0 is printed). For x or X conversion specifiers, a non-zero result shall have 0x (or 0X) prefixed to it. For a, A, e, E, f, F, g, and G conversion specifiers, the result shall always contain a radix character, even if no digits follow the radix character. Without this flag, a radix character appears in the result of these conversions only if a digit follows it. For g and G conversion specifiers, trailing zeros shall **not** be removed from the result as they normally are. For other conversion specifiers, the behavior is undefined.

**0**

For d, i, o, u, x, X, a, A, e, E, f, F, g, and G conversion specifiers, leading zeros (following any indication of sign or base) are used to pad to the field width rather than performing space padding, except when converting an infinity or NaN. If the '0' and '-' flags both appear, the '0' flag is ignored. For d, i, o, u, x, and X conversion specifiers, if a precision is specified, the '0' flag shall be ignored. [CX] If the '0' and flags both appear, the grouping characters are inserted before zero padding. For other conversions, the behavior is undefined.

## Length Modifiers

The length modifiers and their meanings are:

## hh

Specifies that a following d, i, o, u, x, or X conversion specifier applies to a `signed char` or `unsigned char` argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to `signed char` or `unsigned char` before printing); or that a following n conversion specifier applies to a pointer to a `signed char` argument.

## h

Specifies that a following d, i, o, u, x, or X conversion specifier applies to a `short` or `unsigned short` argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to `short` or `unsigned short` before printing); or that a following n conversion specifier applies to a pointer to a `short` argument.

## I (ell)

Specifies that a following d, i, o, u, x, or X conversion specifier applies to a `long` or `unsigned long` argument; that a following n conversion specifier applies to a pointer to a `long` argument; that a following c conversion specifier applies to a `wint_t` argument; that a following s conversion specifier applies to a pointer to a `wchar_t` argument; or has no effect on a following a, A, e, E, f, F, g, or G conversion specifier.

## II (ell-ell)

Specifies that a following d, i, o, u, x, or X conversion specifier applies to a `long long` or `unsigned long long` argument; or that a following n conversion specifier applies to a pointer to a `long long` argument.

## j

Specifies that a following d, i, o, u, x, or X conversion specifier applies to an `intmax_t` or `uintmax_t` argument; or that a following n conversion specifier applies to a pointer to an `intmax_t` argument.

## z

Specifies that a following d, i, o, u, x, or X conversion specifier applies to a `size_t` or the corresponding signed integer type argument; or that a following n conversion specifier applies to a pointer to a signed integer type corresponding to a `size_t` argument.

## t

Specifies that a following d, i, o, u, x, or X conversion specifier applies to a `ptrdiff_t` or the corresponding `unsigned` type argument; or that a following n conversion specifier applies to a pointer to a `ptrdiff_t` argument.

## L

Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to a `long double` argument.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

## Conversion Specifiers

The conversion specifiers and their meanings are:

### d, i

The `int` argument shall be converted to a signed decimal in the style "[-]dddd". The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1. The result of converting zero with an explicit precision of zero shall be no characters.

### o

The `unsigned` argument shall be converted to unsigned octal format in the style "dddd". The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1. The result of converting zero with an explicit precision of zero shall be no characters.

### u

The `unsigned` argument shall be converted to unsigned decimal format in the style "dddd". The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1. The result of converting zero with an explicit precision of zero shall be no characters.

### x

The `unsigned` argument shall be converted to unsigned hexadecimal format in the style "dddd"; the letters "abcdef" are used. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1. The result of converting zero with an explicit precision of zero shall be no characters.

### X

Equivalent to the x conversion specifier, except that letters "ABCDEF" are used instead of "abcdef".

### f, F

The `double` argument shall be converted to decimal notation in the style "[-]ddd.ddd", where the number of digits after the radix character is equal to the precision specification. If the precision is missing, it shall be taken as 6; if the precision is explicitly zero and no '#' flag is present, no radix character shall

appear. If a radix character appears, at least one digit appears before it. The low-order digit shall be rounded in an implementation-defined manner.

A `double` argument representing an infinity shall be converted in one of the styles "[-]inf" or "[-]infinity"; which style is implementation-defined. A `double` argument representing a NaN shall be converted in one of the styles "[-]nan(n-char-sequence)" or "[-]nan"; which style, and the meaning of any n-char-sequence, is implementation-defined. The F conversion specifier produces "INF", "INFINITY", or "NAN" instead of "inf", "infinity", or "nan", respectively.

### e, E

The `double` argument shall be converted in the style "[-]d.ddd\_e±dd", where there is one digit before the radix character (which is non-zero if the argument is non-zero) and the number of digits after it is equal to the precision; if the precision is missing, it shall be taken as 6; if the precision is zero and no '#' flag is present, no radix character shall appear. The low-order digit shall be rounded in an implementation-defined manner. The E conversion specifier shall produce a number with 'E' instead of 'e' introducing the exponent. The exponent shall always contain at least two digits. If the value is zero, the exponent shall be zero.

A `double` argument representing an infinity or NaN shall be converted in the style of an f or F conversion specifier.

### g, G

The `double` argument representing a floating-point number shall be converted in the style f or e (or in the style F or E in the case of a G conversion specifier), depending on the value converted and the precision. Let P equal the precision if non-zero, 6 if the precision is omitted, or 1 if the precision is zero. Then, if a conversion with style E would have an exponent of X:

- If  $P > X \geq -4$ , the conversion shall be with style f (or F) and precision  $P - (X + 1)$ .
- Otherwise, the conversion shall be with style e (or E) and precision P-1.

Finally, unless the '#' flag is used, any trailing zeros shall be removed from the fractional portion of the result and the decimal-point character shall be removed if there is no fractional portion remaining.

A `double` argument representing an infinity or NaN shall be converted in the style of an f or F conversion specifier.

### a, A

A `double` argument representing a floating-point number shall be converted in the style "[-]0x\_h.\_hhhh\_p±d", where there is one hexadecimal digit (which shall be non-zero if the argument is a normalized floating-point number and is otherwise unspecified) before the decimal-point character and the number of hexadecimal digits after it is equal to the precision; if the precision is missing and `FLT_RADIX` is a power of 2, then the precision shall be sufficient for an exact representation of

the value; if the precision is missing and FLT\_RADIX is not a power of 2, then the precision shall be sufficient to distinguish values of type `double`, except that trailing zeros may be omitted; if the precision is zero and the '#' flag is not specified, no decimal-point character shall appear. The letters "abcdef" shall be used for a conversion and the letters "ABCDEF" for A conversion. The A conversion specifier produces a number with 'X' and 'P' instead of 'x' and 'p'. The exponent shall always contain at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent shall be zero.

A `double` argument representing an infinity or NaN shall be converted in the style of an f or F conversion specifier.

#### c

The `int` argument shall be converted to an `unsigned char`, and the resulting byte shall be written.

If an l (ell) qualifier is present, [CX] the `wint_t` argument shall be converted to a multi-byte sequence as if by a call to `wcrtomb()` with a pointer to storage of at least MB\_CUR\_MAX bytes, the `wint_t` argument converted to `wchar_t`, and an initial shift state, and the resulting byte(s) written.

#### s

The argument shall be a pointer to an array of `char`. Bytes from the array shall be written up to (but not including) any terminating null byte. If the precision is specified, no more than that many bytes shall be written. If the precision is not specified or is greater than the size of the array, the application shall ensure that the array contains a null byte.

If an l (ell) qualifier is present, the argument shall be a pointer to an array of type `wchar_t`. Wide characters from the array shall be converted to characters (each as if by a call to the `wcrtomb()` function, with the conversion state described by an `mbstate_t` object initialized to zero before the first wide character is converted) up to and including a terminating null wide character. The resulting characters shall be written up to (but not including) the terminating null character (byte). If no precision is specified, the application shall ensure that the array contains a null wide character. If a precision is specified, no more than that many characters (bytes) shall be written (including shift sequences, if any), and the array shall contain a null wide character if, to equal the character sequence length given by the precision, the function would need to access a wide character one past the end of the array. In no case shall a partial character be written.

#### p

The argument shall be a pointer to `void`. The value of the pointer is converted to a sequence of printable characters, in an implementation-defined manner.

**n**

The argument shall be a pointer to an integer into which is written the number of bytes written to the output so far by this call to one of the `fprintf()` functions. No argument is converted.

**C**

[XSI] Equivalent to `lc.`

**S**

[XSI] Equivalent to `ls.`

**%**

Write a '%' character; no argument shall be converted. The application shall ensure that the complete conversion specification is `%%`.

If a conversion specification does not match one of the above forms, the behavior is undefined. If any argument is not the correct type for the corresponding conversion specification, the behavior is undefined.

In no case shall a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field shall be expanded to contain the conversion result. Characters generated by `fprintf()` and `printf()` are printed as if `fputc()` had been called.

For the a and A conversion specifiers, if `FLT_RADIX` is a power of 2, the value shall be correctly rounded to a hexadecimal floating number with the given precision.

For a and A conversions, if `FLT_RADIX` is not a power of 2 and the result is not exactly representable in the given precision, the result should be one of the two adjacent numbers in hexadecimal floating style with the given precision, with the extra stipulation that the error should have a correct sign for the current rounding direction.

For the e, E, f, F, g, and G conversion specifiers, if the number of significant decimal digits is at most `DECIMAL_DIG`, then the result should be correctly rounded. If the number of significant decimal digits is more than `DECIMAL_DIG` but the source value is exactly representable with `DECIMAL_DIG` digits, then the result should be an exact representation with trailing zeros. Otherwise, the source value is bounded by two adjacent decimal strings  $L < U$ , both having `DECIMAL_DIG` significant digits; the value of the resultant decimal string  $D$  should satisfy  $L \leq D \leq U$ , with the extra stipulation that the error should have a correct sign for the current rounding direction.

[CX] The last data modification and last file status change timestamps of the file shall be marked for update:

1. Between the call to a successful execution of `fprintf()` or `printf()` and the next successful completion of a call to `fflush()` or `fclose()` on the

same stream or a call to `exit()` or `abort()`

2. Upon successful completion of a call to `dprintf()`

## RETURN VALUE

Upon successful completion, the [CX] `dprintf()`, `fprintf()`, and `printf()` functions shall return the number of bytes transmitted.

[CX] Upon successful completion, the `asprintf()` function shall return the number of bytes written to the allocated string stored in the location referenced by `ptr`, excluding the terminating null byte.

Upon successful completion, the `sprintf()` function shall return the number of bytes written to `s`, excluding the terminating null byte.

Upon successful completion, the `snprintf()` function shall return the number of bytes that would be written to `s` had `n` been sufficiently large excluding the terminating null byte.

If an error was encountered, these functions shall return a negative value [CX] and set `errno` to indicate the error. For `asprintf()`, if memory allocation was not possible, or if some other error occurs, the function shall return a negative value, and the contents of the location referenced by `ptr` are undefined, but shall not refer to allocated memory.

If the value of `n` is zero on a call to `snprintf()`, nothing shall be written, the number of bytes that would have been written had `n` been sufficiently large excluding the terminating null shall be returned, and `s` may be a null pointer.

## ERRORS

For the conditions under which [CX] `dprintf()`, `fprintf()`, and `printf()` fail and may fail, refer to `fputc()` or `fputwc()`.

In addition, all forms of `fprintf()` shall fail if:

### [EILSEQ]

[CX] A wide-character code that does not correspond to a valid character has been detected.

### [EOVERFLOW]

[CX] The value to be returned is greater than `{INT_MAX}`.

[CX] The `asprintf()` function shall fail if:

### [ENOMEM]

Insufficient storage space is available.

The `dprintf()` function may fail if:

**[EBADF]**

The `fildes` argument is not a valid file descriptor.

The [CX] `dprintf()`, `fprintf()`, and `printf()` functions may fail if:

**[ENOMEM]**

[CX] Insufficient storage space is available.

## EXAMPLES

### Printing Language-Independent Date and Time

The following statement can be used to print date and time using a language-independent format:

```
printf(format, weekday, month, day, hour, min);
```

For American usage, `format` could be a pointer to the following string:

```
"%s, %s %d, %d:%.2d\n"
```

This example would produce the following message:

```
Sunday, July 3, 10:02
```

For German usage, `format` could be a pointer to the following string:

```
"%1$s, %3$d. %2$s, %4$d:%5$.2d\n"
```

This definition of `format` would produce the following message:

```
Sonntag, 3. Juli, 10:02
```

### Printing File Information

The following example prints information about the type, permissions, and number of links of a specific file in a directory.

```
#include <stdio.h>
#include <sys/types.h>
#include <pwd.h>
#include <grp.h>

char *strperm (mode_t);
```

```

...
struct stat statbuf;
struct passwd *pwd;
struct group *grp;
...
printf("%10.10s", strperm (statbuf.st_mode));
printf("%4d", statbuf.st_nlink);

if ((pwd = getpwuid(statbuf.st_uid)) != NULL)
    printf(" %-8.8s", pwd->pw_name);
else
    printf(" %-8ld", (long) statbuf.st_uid);

if ((grp = getgrgid(statbuf.st_gid)) != NULL)
    printf(" %-8.8s", grp->gr_name);
else
    printf(" %-8ld", (long) statbuf.st_gid);

printf("%9jd", (intmax_t) statbuf.st_size);

```

## Printing a Localized Date String

The following example gets a localized date string:

```

#include <stdio.h>
#include <time.h>
#include <langinfo.h>
...
struct dirent *dp;
struct tm *tm;
char datestring[256];
...
strftime(datestring, sizeof(datestring), nl_langinfo (D_T_FMT), tm
printf(" %s %s\n", datestring, dp->d_name);

```

## Printing Error Information

The following example uses `fprintf()` to write error information to standard error:

```

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

```

```

#include <string.h>
#include <errno.h>

#define LOCKFILE "/etc/ptmp"
#define PASSWDFILE "/etc/passwd"

...
int pfd;
...
if ((pfd = open(LOCKFILE, O_WRONLY | O_CREAT | O_EXCL,
    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
{
    fprintf(stderr, "Cannot open /etc/ptmp. Try again later.\n");
    exit(1);
}
...
if (link(LOCKFILE,PASSWDFILE) == -1) {
    fprintf(stderr, "Link error: %s\n", strerror(errno));
    exit(1);
}

```

## Printing Usage Information

The following example checks to make sure the program has the necessary arguments, and uses `fprintf()` to print usage information if the expected number of arguments is not present:

```

#include <stdio.h>
#include <stdlib.h>
...
char *Options = "hdbtl";
...
if (argc < 2) {
    fprintf(stderr, "Usage: %s -%s <file\n", argv[0], Options); exit(1);
}

```

## Formatting a Decimal String

The following example prints a key and data pair on `stdout`. Note use of the '\*' in the format string; this ensures the correct number of decimal places for the element based on the number of elements requested:

```

#include <stdio.h>
...
long i;
char *keystr;
int elementlen, len;

```

```

...
while (len < elementlen) {
...
    printf("%s Element%0*ld\n", keystr, elementlen, i);
...
}

```

## Creating a Pathname

The following example creates a pathname using information from a previous `getpwnam()` function that returned the password database entry of the user:

```

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
...
char *pathname;
struct passwd *pw;
size_t len;
...
// digits required for pid_t is number of bits times
// log2(10) = approx 10/33
len = strlen(pw->pw_dir) + 1 + 1+(sizeof(pid_t)*80+32)/33 +
      sizeof ".out";
pathname = malloc(len);
if (pathname != NULL)
{
    snprintf(pathname, len, "%s/%jd.out", pw->pw_dir,
             (intmax_t)getpid());
...
}

```

## Reporting an Event

The following example loops until an event has timed out. The `pause()` function waits forever unless it receives a signal. The `fprintf()` statement should never occur due to the possible return values of `pause()`:

```

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
...
while (!event_complete) {

```

```
...
    if (pause() != -1 || errno != EINTR)
        fprintf(stderr, "pause: unknown error: %s\n", strerror(errno));
}
```

## Printing Monetary Information

The following example uses `strfmon()` to convert a number and store it as a formatted monetary string named `convbuf`. If the first number is printed, the program prints the format and the description; otherwise, it just prints the number:

```
#include <monetary.h>
#include <stdio.h>
...
struct tblfmt {
    char *format;
    char *description;
};

struct tblfmt table[] = {
    { "%n", "default formatting" },
    { "%11n", "right align within an 11 character field" },
    { "%#5n", "aligned columns for values up to 99999" },
    { "%=*#5n", "specify a fill character" },
    { "%=0#5n", "fill characters do not use grouping" },
    { "%^#5n", "disable the grouping separator" },
    { "%^#5.0n", "round off to whole units" },
    { "%^#5.4n", "increase the precision" },
    { "%(#5n", "use an alternative pos/neg style" },
    { "%!(#5n", "disable the currency symbol" },
};

...
float input[3];
int i, j;
char convbuf[100];
...
strfmon(convbuf, sizeof(convbuf), table[i].format, input[j]);

if (j == 0) {
    printf("%s %s      %s\n", table[i].format,
           convbuf, table[i].description);
}
else {
    printf("      %s\n", convbuf);
}
```

## Printing Wide Characters

The following example prints a series of wide characters. Suppose that "L @" expands to three bytes:

```
wchar_t wz [3] = L"@@";           // Zero-terminated
wchar_t wn [3] = L"@@";           // Untermminated

fprintf (stdout,"%ls", wz);      // Outputs 6 bytes
fprintf (stdout,"%ls", wn);      // Undefined because wn has no terminator
fprintf (stdout,"%4ls", wz);     // Outputs 3 bytes
fprintf (stdout,"%4ls", wn);     // Outputs 3 bytes; no terminator needed
fprintf (stdout,"%9ls", wz);     // Outputs 6 bytes
fprintf (stdout,"%9ls", wn);     // Outputs 9 bytes; no terminator needed
fprintf (stdout,"%10ls", wz);    // Outputs 6 bytes
fprintf (stdout,"%10ls", wn);    // Undefined because wn has no terminator
```

In the last line of the example, after processing three characters, nine bytes have been output. The fourth character must then be examined to determine whether it converts to one byte or more. If it converts to more than one byte, the output is only nine bytes. Since there is no fourth character in the array, the behavior is undefined.

## APPLICATION USAGE

If the application calling `fprintf()` has any objects of type `wint_t` or `wchar_t`, it must also include the `<wchar.h>` header to have these objects defined.

The space allocated by a successful call to `asprintf()` should be subsequently freed by a call to `free()`.

## RATIONALE

If an implementation detects that there are insufficient arguments for the format, it is recommended that the function should fail and report an [EINVAL] error.

The behavior specified for the %lc conversion differs slightly from the specification in the ISO C standard, in that printing the null wide character produces a null byte instead of 0 bytes of output as would be required by a strict reading of the ISO C standard's direction to behave as if applying the %ls specifier to a `wchar_t` array whose first element is the null wide character. Requiring a multi-byte output for every possible wide character, including the null character, matches historical practice, and provides consistency with %c in `fprintf()` and with both %c and

%lc in `fwprintf()`. It is anticipated that a future edition of the ISO C standard will change to match the behavior specified here.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [2.5 Standard I/O Streams](#)

- `fputc()`
- `fscanf()`
- `setlocale()`
- `strfmon()`
- `strlcat()`
- `wcrtomb()`
- `wcslcat()`

XBD 7. Locale, `<inttypes.h>`, `<stdio.h>`, `<wchar.h>`

## CHANGE HISTORY

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

### Issue 5

Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the l (ell) qualifier can now be used with c and s conversion specifiers.

The `snprintf()` function is new in Issue 5.

### Issue 6

Extensions beyond the ISO C standard are marked.

The normative text is updated to avoid use of the term "must" for application requirements.

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

- The prototypes for `fprintf()`, `printf()`, `snprintf()`, and `sprintf()` are updated, and the XSI shading is removed from `snprintf()`.
- The description of `snprintf()` is aligned with the ISO C standard. Note that this supersedes the `snprintf()` description in The Open Group Base Resolution bwg98-006, which changed the behavior from Issue 5.
- The DESCRIPTION is updated.

The DESCRIPTION is updated to use the terms "conversion specifier" and "conversion specification" consistently.

ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

An example of printing wide characters is added.

## Issue 7

Austin Group Interpretation 1003.1-2001 #161 is applied, updating the DESCRIPTION of the 0 flag.

Austin Group Interpretation 1003.1-2001 #170 is applied.

ISO/IEC 9899:1999 standard, Technical Corrigendum 2 #68 (SD5-XSH-ERN-70) is applied, revising the description of g and G.

SD5-XSH-ERN-174 is applied.

The `dprintf()` function is added from The Open Group Technical Standard, 2006, Extended API Set Part 1.

Functionality relating to the %n\$ form of conversion specification and the flag is moved from the XSI option to the Base.

Changes are made related to support for finegrained timestamps.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0163 [302], XSH/TC1-2008/0164 [316], XSH/TC1-2008/0165 [316], XSH/TC1-2008/0166 [451,291], and XSH/TC1-2008/0167 [14] are applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0126 [894], XSH/TC2-2008/0127 [557], and XSH/TC2-2008/0128 [936] are applied.

## Issue 8

Austin Group Defect 986 is applied, adding `strlcat()` and `wcslcat()` to the SEE ALSO section.

Austin Group Defect 1020 is applied, clarifying that the `snprintf()` argument `n` limits the number of bytes written to `s`; it is not necessarily the same as the size of `s`.

Austin Group Defect 1021 is applied, changing "output error" to "error" in the RETURN VALUE section.

Austin Group Defect 1137 is applied, clarifying the use of "%n\$" and "\*m\$" in conversion specifications.

Austin Group Defect 1205 is applied, changing the description of the % conversion specifier.

Austin Group Defect 1219 is applied, removing the `snprintf()`-specific [EOVERFLOW] error.

Austin Group Defect 1496 is applied, adding the `asprintf()` function.

Austin Group Defect 1562 is applied, clarifying that it is the application's responsibility to ensure that the format is a character string, beginning and ending in its initial shift state, if any.

Austin Group Defect 1647 is applied, changing the description of the c conversion specifier and updating the statement that this volume of POSIX.1-2024 defers to the ISO C standard so that it excludes the %lc conversion when passed a null wide character.

---

## 1.223. sprintf - print formatted output

### SYNOPSIS

```
#include <stdio.h>

int asprintf(char **restrict ptr, const char *restrict format, ...
int dprintf(int fildes, const char *restrict format, ...);

int fprintf(FILE *restrict stream, const char *restrict format, ...
int printf(const char *restrict format, ...);
int snprintf(char *restrict s, size_t n, const char *restrict format,
int sprintf(char *restrict s, const char *restrict format, ...);
```

### DESCRIPTION

The `fprintf()` function shall place output on the named output `stream`. The `printf()` function shall place output on the standard output stream `stdout`. The `sprintf()` function shall place output followed by the null byte, '\0', in consecutive bytes starting at `s`; it is the user's responsibility to ensure that enough space is available.

The `asprintf()` function shall be equivalent to `sprintf()`, except that the output string shall be written to dynamically allocated memory, allocated as if by a call to `malloc()`, of sufficient length to hold the resulting string, including a terminating null byte. If the call to `asprintf()` is successful, the address of this dynamically allocated string shall be stored in the location referenced by `ptr`.

The `dprintf()` function shall be equivalent to the `fprintf()` function, except that `dprintf()` shall write output to the file associated with the file descriptor specified by the `fildes` argument rather than place output on a stream.

The `snprintf()` function shall be equivalent to `sprintf()`, with the addition of the `n` argument which limits the number of bytes written to the buffer referred to by `s`. If `n` is zero, nothing shall be written and `s` may be a null pointer. Otherwise, output bytes beyond the `n`-1st shall be discarded instead of being written to the array, and a null byte is written at the end of the bytes actually written into the array.

If copying takes place between objects that overlap as a result of a call to `sprintf()` or `snprintf()`, the results are undefined.

Each of these functions converts, formats, and prints its arguments under control of the `format`. The application shall ensure that the format is a character string, beginning and ending in its initial shift state, if any. The format is composed of zero or more directives: *ordinary characters*, which are simply copied to the output stream, and *conversion specifications*, each of which shall result in the fetching of zero or more arguments. The results are undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments shall be evaluated but are otherwise ignored.

Conversions can be applied to the nth argument after the format in the argument list, rather than to the next unused argument. In this case, the conversion specifier character % (see below) is replaced by the sequence "%n\$", where n is a decimal integer in the range [1,{NL\_ARGMAX}], giving the position of the argument in the argument list. This feature provides for the definition of format strings that select arguments in an order appropriate to specific languages (see the EXAMPLES section).

The format can contain either numbered argument conversion specifications (that is, those introduced by "%n\$" and optionally containing the "\*m\$" forms of field width and precision), or unnumbered argument conversion specifications (that is, those introduced by the % character and optionally containing the \* form of field width and precision), but not both. The only exception to this is that %% can be mixed with the "%n\$" form. The results of mixing numbered and unnumbered argument specifications in a format string are undefined. When numbered argument specifications are used, specifying the Nth argument requires that all the leading arguments, from the first to the (N-1)th, are specified in the format string.

In format strings containing the "%n\$" form of conversion specification, numbered arguments in the argument list can be referenced from the format string as many times as required.

In format strings containing the % form of conversion specification, each conversion specification uses the first unused argument in the argument list.

All forms of the `fprintf()` functions allow for the insertion of a language-dependent radix character in the output string. The radix character is defined in the current locale (category LC\_NUMERIC). In the POSIX locale, or in a locale where the radix character is not defined, the radix character shall default to a period ('.') .

Each conversion specification is introduced by the '%' character or by the character sequence "%n\$", after which the following appear in sequence:

- Zero or more flags (in any order), which modify the meaning of the conversion specification.
- An optional minimum field width. If the converted value has fewer bytes than the field width, it shall be padded with space characters by default on the left;

it shall be padded on the right if the left-adjustment flag ('-'), described below, is given to the field width. The field width takes the form of an asterisk ('\*'), or in conversion specifications introduced by "%n\$" the "m\$" string, described below, or a decimal integer.

- An optional precision that gives the minimum number of digits to appear for the d, i, o, u, x, and X conversion specifiers; the number of digits to appear after the radix character for the a, A, e, E, f, and F conversion specifiers; the maximum number of significant digits for the g and G conversion specifiers; or the maximum number of bytes to be printed from a string in the s conversion specifiers. The precision takes the form of a period ('.') followed either by an asterisk ('\*'), or in conversion specifications introduced by "%n\$" the "m\$" string, described below, or an optional decimal digit string, where a null digit string is treated as zero. If a precision appears with any other conversion specifier, the behavior is undefined.
- An optional length modifier that specifies the size of the argument.
- A conversion specifier character that indicates the type of conversion to be applied.

A field width, or precision, or both, may be indicated by an asterisk ('\*'). In this case an argument of type `int` supplies the field width or precision. Applications shall ensure that arguments specifying field width, or precision, or both appear in that order before the argument, if any, to be converted. A negative field width is taken as a '-' flag followed by a positive field width. A negative precision is taken as if the precision were omitted. In format strings containing conversion specifications introduced by "%n\$", in addition to being indicated by the decimal digit string, a field width may be indicated by the sequence "m\$" and precision by the sequence ".\*m\$", where m is a decimal integer in the range [1,{NL\_ARGMAX}] giving the position in the argument list (after the format argument) of an integer argument containing the field width or precision, for example:

```
printf("%1$d:%2$.*3$d:%4$.*3$d\n", hour, min, precision, sec);
```

## Flag Characters

The flag characters and their meanings are:

' (apostrophe)

The integer portion of the result of a decimal conversion (%i, %d, %u, %f, %F, %g, or %G) shall be formatted with thousands' grouping characters. For other conversions the behavior is undefined. The non-monetary grouping character is used.

-  
The result of the conversion shall be left-justified within the field. The conversion is right-justified if this flag is not specified.

+

The result of a signed conversion shall always begin with a sign ('+' or '-'). The conversion shall begin with a sign only when a negative value is converted if this flag is not specified.

### space

If the first character of a signed conversion is not a sign or if a signed conversion results in no characters, a space shall be prefixed to the result. This means that if the space and '+' flags both appear, the space flag shall be ignored.

#

Specifies that the value is to be converted to an alternative form. For o conversion, it shall increase the precision, if and only if necessary, to force the first digit of the result to be a zero (if the value and precision are both 0, a single 0 is printed). For x or X conversion specifiers, a non-zero result shall have 0x (or 0X) prefixed to it. For a, A, e, E, f, F, g, and G conversion specifiers, the result shall always contain a radix character, even if no digits follow the radix character. Without this flag, a radix character appears in the result of these conversions only if a digit follows it. For g and G conversion specifiers, trailing zeros shall not be removed from the result as they normally are. For other conversion specifiers, the behavior is undefined.

0

For d, i, o, u, x, X, a, A, e, E, f, F, g, and G conversion specifiers, leading zeros (following any indication of sign or base) are used to pad to the field width rather than performing space padding, except when converting an infinity or NaN. If the '0' and '-' flags both appear, the '0' flag is ignored. For d, i, o, u, x, and X conversion specifiers, if a precision is specified, the '0' flag shall be ignored. If the '0' and apostrophe flags both appear, the grouping characters are inserted before zero padding. For other conversions, the behavior is undefined.

## Length Modifiers

The length modifiers and their meanings are:

### hh

Specifies that a following d, i, o, u, x, or X conversion specifier applies to a `signed char` or `unsigned char` argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to `signed char` or `unsigned char` before printing); or that a following n conversion specifier applies to a pointer to a `signed char` argument.

## **h**

Specifies that a following d, i, o, u, x, or X conversion specifier applies to a `short` or `unsigned short` argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to `short` or `unsigned short` before printing); or that a following n conversion specifier applies to a pointer to a `short` argument.

## **I (ell)**

Specifies that a following d, i, o, u, x, or X conversion specifier applies to a `long` or `unsigned long` argument; that a following n conversion specifier applies to a pointer to a `long` argument; that a following c conversion specifier applies to a `wint_t` argument; that a following s conversion specifier applies to a pointer to a `wchar_t` argument; or has no effect on a following a, A, e, E, f, F, g, or G conversion specifier.

## **II (ell-ell)**

Specifies that a following d, i, o, u, x, or X conversion specifier applies to a `long long` or `unsigned long long` argument; or that a following n conversion specifier applies to a pointer to a `long long` argument.

## **j**

Specifies that a following d, i, o, u, x, or X conversion specifier applies to an `intmax_t` or `uintmax_t` argument; or that a following n conversion specifier applies to a pointer to an `intmax_t` argument.

## **z**

Specifies that a following d, i, o, u, x, or X conversion specifier applies to a `size_t` or the corresponding signed integer type argument; or that a following n conversion specifier applies to a pointer to a signed integer type corresponding to a `size_t` argument.

## **t**

Specifies that a following d, i, o, u, x, or X conversion specifier applies to a `ptrdiff_t` or the corresponding unsigned type argument; or that a following n conversion specifier applies to a pointer to a `ptrdiff_t` argument.

## **L**

Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to a `long double` argument.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

# **Conversion Specifiers**

The conversion specifiers and their meanings are:

## d, i

The `int` argument shall be converted to a signed decimal in the style "[-]dddd". The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1. The result of converting zero with an explicit precision of zero shall be no characters.

## o

The `unsigned` argument shall be converted to unsigned octal format in the style "ddddd". The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1. The result of converting zero with an explicit precision of zero shall be no characters.

## u

The `unsigned` argument shall be converted to unsigned decimal format in the style "ddddd". The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1. The result of converting zero with an explicit precision of zero shall be no characters.

## x

The `unsigned` argument shall be converted to unsigned hexadecimal format in the style "ddddd"; the letters "abcdef" are used. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1. The result of converting zero with an explicit precision of zero shall be no characters.

## X

Equivalent to the x conversion specifier, except that letters "ABCDEF" are used instead of "abcdef".

## f, F

The `double` argument shall be converted to decimal notation in the style "[-]ddd.ddd", where the number of digits after the radix character is equal to the precision specification. If the precision is missing, it shall be taken as 6; if the precision is explicitly zero and no '#' flag is present, no radix character shall appear. If a radix character appears, at least one digit appears before it. The low-order digit shall be rounded in an implementation-defined manner.

A `double` argument representing an infinity shall be converted in one of the styles "[-]inf" or "[-]infinity"; which style is implementation-defined. A `double` argument representing a NaN shall be converted in one of the styles "[-]nan(n-char-sequence)" or "[-]nan"; which style, and the meaning of any n-char-sequence,

is implementation-defined. The F conversion specifier produces "INF", "INFINITY", or "NAN" instead of "inf", "infinity", or "nan", respectively.

### e, E

The `double` argument shall be converted in the style "[-]d.ddd e $\pm$ dd", where there is one digit before the radix character (which is non-zero if the argument is non-zero) and the number of digits after it is equal to the precision; if the precision is missing, it shall be taken as 6; if the precision is zero and no '#' flag is present, no radix character shall appear. The low-order digit shall be rounded in an implementation-defined manner. The E conversion specifier shall produce a number with 'E' instead of 'e' introducing the exponent. The exponent shall always contain at least two digits. If the value is zero, the exponent shall be zero.

A `double` argument representing an infinity or NaN shall be converted in the style of an f or F conversion specifier.

### g, G

The `double` argument representing a floating-point number shall be converted in the style f or e (or in the style F or E in the case of a G conversion specifier), depending on the value converted and the precision. Let P equal the precision if non-zero, 6 if the precision is omitted, or 1 if the precision is zero. Then, if a conversion with style E would have an exponent of X:

- If P > X  $\geq$  -4, the conversion shall be with style f (or F) and precision P-(X+1).
- Otherwise, the conversion shall be with style e (or E) and precision P-1.

Finally, unless the '#' flag is used, any trailing zeros shall be removed from the fractional portion of the result and the decimal-point character shall be removed if there is no fractional portion remaining.

A `double` argument representing an infinity or NaN shall be converted in the style of an f or F conversion specifier.

### a, A

A `double` argument representing a floating-point number shall be converted in the style "[-]0x h.ffffp $\pm$ d", where there is one hexadecimal digit (which is non-zero if the argument is a normalized floating-point number and is otherwise unspecified) before the radix character and the number of hexadecimal digits after it is equal to the precision; if the precision is missing, it shall be taken as sufficient to represent the floating-point number exactly, and if the precision is zero and no '#' flag is present, no radix character shall appear. The letters "abcdef" are used for a conversion, the letters "ABCDEF" for A conversion. The A conversion specifier produces a number with 'X' and 'P' instead of 'x' and 'p'. The exponent shall always contain at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2.

A `double` argument representing an infinity or NaN shall be converted in the style of an f or F conversion specifier.

**c**

If no l length modifier is present, the `int` argument shall be converted to an `unsigned char`, and the resulting byte shall be written.

If an l length modifier is present, the `wint_t` argument shall be converted as if by a call to `wcrtomb()` with no state, and the resulting multibyte character shall be written.

**s**

If no l length modifier is present, the application shall ensure that the argument is a pointer to a character array containing a character sequence beginning in the initial shift state. Characters from the array shall be written up to (but not including) the terminating null byte. If the precision is specified, no more than that many bytes shall be written and any partial multibyte character shall not be written. If the precision is not specified or is greater than the size of the array, the application shall ensure that the array contains a null byte.

If an l length modifier is present, the application shall ensure that the argument is a pointer to an array of type `wchar_t`. Wide characters from the array shall be converted to multibyte characters (each as if by a call to `wcrtomb()`, with the conversion state described by an `mbstate_t` object initialized to zero before the first wide character is converted) and the resulting multibyte characters shall be written up to (but not including) the terminating null wide character. If the precision is specified, no more than that many bytes shall be written and any partial multibyte character shall not be written. If the precision is not specified or is greater than the size of the converted wide character sequence, the application shall ensure that the array contains a null wide character.

**p**

The application shall ensure that the argument is a pointer to `void`. The value of the pointer is converted to a sequence of printable characters, in an implementation-defined manner.

**n**

The application shall ensure that the argument is a pointer to an integer into which is written the number of bytes written to the output so far by this call to one of the `fprintf()` functions. No argument is converted. The format string shall be written to the output as-is; if a conversion specification contains a field width or precision, the value of the bytes written to the output shall be determined as if those values had been specified.

For `snprintf()`, this means the number of bytes that would have been written to the buffer s, not counting the terminating null byte, if n were large enough.

%

A '%' character shall be written. No argument is converted. The complete conversion specification shall be "%%".

If a conversion specification does not match one of the above forms, the behavior is undefined.

## RETURN VALUE

Upon successful completion, the `fprintf()`, `printf()`, `dprintf()`, `snprintf()`, `asprintf()`, and `sprintf()` functions shall return the number of bytes transmitted.

If an output error was encountered, these functions shall return a negative value.

If the value of *n* is zero on a call to `snprintf()`, nothing shall be written, the number of bytes that would have been written had *n* been sufficiently large shall be returned, and *s* may be a null pointer.

## ERRORS

The functions shall fail if:

### EILSEQ

An illegal byte sequence was detected in a wide character conversion.

### EOVERFLOW

The value to be stored is larger than an integer of the corresponding type.

The `fprintf()`, `printf()`, `dprintf()`, `snprintf()`, and `sprintf()` functions may fail if:

### ENOMEM

Insufficient storage space is available.

The `snprintf()` function shall fail if:

### EOVERFLOW

The value of *n* is greater than `{INT_MAX}` or insufficient storage space was provided.

The `asprintf()` function shall fail if:

### ENOMEM

Insufficient memory space is available to store the resulting string.

The `dprintf()` function shall fail if:

## EBADF

The file descriptor underlying the stream is not a valid file descriptor or is not open for writing.

The `fprintf()` and `printf()` functions may fail if:

## ENOMEM

Insufficient storage space is available.

## EILSEQ

A wide-character code that does not correspond to a valid character has been detected.

## EXAMPLES

### Example 1: Print language-independent date and time

The following statement can be used to print date and time in a language-independent way:

```
printf (format, weekday, month, day, hour, min);
```

For American usage, `format` could be a pointer to the following string:

```
"%s, %s %d, %.2d:%.2d\n"
```

This example could produce the following output:

```
Sunday, July 3, 10:02
```

For German usage, `format` could be a pointer to the following string:

```
"%1$s, %3$d. %2$s, %4$02.2d:%5$02.2d\n"
```

This definition of `format` could produce the following output:

```
Sonntag, 3. Juli, 10:02
```

### Example 2: Print a file list

To print a date and time in the form Sunday, July 3, 10:02, where `weekday` and `month` are pointers to null-terminated strings:

```
printf("%s, %s %d, %.2d:%.2d\n", weekday, month, day, hour, min);
```

To print pi to 5 decimal places:

```
printf("pi = %.5f\n", 3.1415926535);
```

The above example could produce the following output:

```
pi = 3.14159
```

### Example 3: Print to different destinations

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

int main(void) {
    char buffer[100];
    char *dynamic_buffer = NULL;
    int ret;

    /* Write to standard output */
    ret = printf("Hello, World!\n");

    /* Write to buffer */
    ret = sprintf(buffer, "Count: %d", 42);

    /* Write to buffer with size limit */
    ret = snprintf(buffer, sizeof(buffer), "Value: %f", 3.14159);

    /* Write to dynamically allocated buffer */
    ret = asprintf(&dynamic_buffer, "String: %s", "Dynamic");
    if (ret != -1) {
        printf("Allocated: %s\n", dynamic_buffer);
        free(dynamic_buffer);
    }

    return 0;
}
```

## APPLICATION USAGE

Before calling `asprintf()`, the value of `*ptr` is indeterminate and should not be assumed to be `NULL`. If the call is successful, the application shall free the memory allocated by `asprintf()` using `free()`.

## RATIONALE

The `sprintf()` function is prone to buffer overruns. The `snprintf()` function was created to address this security concern by limiting the number of bytes written to the buffer.

The `asprintf()` function was added to handle the common case where the programmer does not know in advance how large a buffer needs to be for the formatted output.

The numbered argument specifications ("%"n\$") provide support for languages where word order differs from English, allowing format strings to be reordered without changing the argument order in function calls.

## FUTURE DIRECTIONS

None.

## SEE ALSO

`fclose()` , `fopen()` , `fputc()` , `puts()` , `scanf()` , `wcrtoutb()`

---

## 1.224. `rand`, `srand` — pseudo-random number generator

---

### SYNOPSIS

```
#include <stdlib.h>

int rand(void);
void srand(unsigned seed);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `rand()` function shall compute a sequence of pseudo-random integers in the range  $[0, \{RAND\_MAX\}]$  [XSI] with a period of at least  $2^{32}$ .

The `rand()` function need not be thread-safe; however, `rand()` shall avoid data races with all functions other than non-thread-safe pseudo-random sequence generation functions.

The `srand()` function uses the argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to `rand()`. If `srand()` is then called with the same seed value, the sequence of pseudo-random numbers shall be repeated. If `rand()` is called before any calls to `srand()` are made, the same sequence shall be generated as when `srand()` is first called with a seed value of 1.

The `srand()` function need not be thread-safe; however, `srand()` shall avoid data races with all functions other than non-thread-safe pseudo-random sequence generation functions.

The implementation shall behave as if no function defined in this volume of POSIX.1-2024 calls `rand()` or `srand()`.

### RETURN VALUE

The `rand()` function shall return the next pseudo-random number in the sequence.

The `srand()` function shall not return a value.

## ERRORS

No errors are defined.

## EXAMPLES

### Generating a Pseudo-Random Number Sequence

The following example demonstrates how to generate a sequence of pseudo-random numbers.

```
#include <stdio.h>
#include <stdlib.h>
...
long count, i;
char *keystr;
int elementlen, len;
char c;
...
/* Initial random number generator. */
srand(1);

/* Create keys using only lowercase characters */
len = 0;
for (i=0; i<count; i++) {
    while (len < elementlen) {
        c = (char) (rand() % 128);
        if (islower(c))
            keystr[len++] = c;
    }

    keystr[len] = '\0';
    printf("%s Element%0*ld\n", keystr, elementlen, i);
    len = 0;
}
```

### Generating the Same Sequence on Different Machines

The following code defines a pair of functions that could be incorporated into applications wishing to ensure that the same sequence of numbers is generated across different machines.

```

static unsigned long next = 1;

int myrand(void) /* RAND_MAX assumed to be 32767. */
{
    next = next * 1103515245 + 12345;
    return((unsigned)(next/65536) % 32768);
}

void mysrand(unsigned seed)
{
    next = seed;
}

```

## APPLICATION USAGE

These functions should be avoided whenever non-trivial requirements (including safety) have to be fulfilled, unless seeded using `getentropy()`.

The `drand48()` and `random()` functions provide much more elaborate pseudo-random number generators.

## RATIONALE

The ISO C standard `rand()` and `srand()` functions allow per-process pseudo-random streams shared by all threads. Those two functions need not change, but there has to be mutual-exclusion that prevents interference between two threads concurrently accessing the random number generator.

With regard to `rand()`, there are two different behaviors that may be wanted in a multi-threaded program:

1. A single per-process sequence of pseudo-random numbers that is shared by all threads that call `rand()`
2. A different sequence of pseudo-random numbers for each thread that calls `rand()`

This is provided by the modified thread-safe function based on whether the seed value is global to the entire process or local to each thread.

This does not address the known deficiencies of the `rand()` function implementations, which have been approached by maintaining more state. In effect, this specifies new thread-safe forms of a deficient function.

# FUTURE DIRECTIONS

None.

## SEE ALSO

- `drand48()`
- `getentropy()`
- `initstate()`
- `<stdlib.h>`

## CHANGE HISTORY

### First released in Issue 1

Derived from Issue 1 of the SVID.

### Issue 5

- The `rand_r()` function is included for alignment with the POSIX Threads Extension.
- A note indicating that the `rand()` function need not be reentrant is added to the DESCRIPTION.

### Issue 6

- Extensions beyond the ISO C standard are marked.
- The `rand_r()` function is marked as part of the Thread-Safe Functions option.

### Issue 7

- Austin Group Interpretation 1003.1-2001 #156 is applied.
- The `rand_r()` function is marked obsolescent.
- POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0301 [743] is applied.

## Issue 8

- Austin Group Defect 1134 is applied, adding `getentropy()`.
  - Austin Group Defect 1302 is applied, aligning these functions with the ISO/IEC 9899:2018 standard.
  - Austin Group Defect 1330 is applied, removing obsolescent interfaces.
-

## 1.225. fscanf, scanf, sscanf

### Synopsis

```
#include <stdio.h>

int fscanf(FILE *restrict stream, const char *restrict format, ...
int scanf(const char *restrict format, ...);
int sscanf(const char *restrict s, const char *restrict format, ...
```

### Description

**[CX]** The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `fscanf()` function shall read from the named input `stream`. The `scanf()` function shall read from the standard input stream `stdin`. The `sscanf()` function shall read from the string `s`. Each function reads bytes, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string `format` described below, and a set of `pointer` arguments indicating where the converted input should be stored. The result is undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments shall be evaluated but otherwise ignored.

**[CX]** Conversions can be applied to the *n*th argument after the `format` in the argument list, rather than to the next unused argument. In this case, the conversion specifier character `%` (see below) is replaced by the sequence `"%n$"`, where `n` is a decimal integer in the range `[1,{NL_ARGMAX}]`. This feature provides for the definition of format strings that select arguments in an order appropriate to specific languages. In format strings containing the `"%n$"` form of conversion specifications, it is unspecified whether numbered arguments in the argument list can be referenced from the format string more than once.

The format can contain either form of a conversion specification—that is, `%` or `"%n$"`—but the two forms cannot be mixed within a single format string. The only exception to this is that `%%` or `%"` can be mixed with the `"%n$"` form. 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 pointers.

The `fscanf()` function in all its forms shall allow detection of a language-dependent radix character in the input string. The radix character is defined in the current locale (category `LC_NUMERIC`). In the POSIX locale, or in a locale where the radix character is not defined, the radix character shall default to a period ('.').

The application shall ensure that the format is a character string, beginning and ending in its initial shift state, if any, composed of zero or more directives. Each directive is composed of one of the following: one or more white-space bytes; an ordinary character (neither '%' nor a white-space byte); or a conversion specification. Each conversion specification is introduced by the character '%' **[CX]** or the character sequence "%n\$", after which the following appear in sequence:

- An optional assignment-suppressing character '\*'.
- An optional non-zero decimal integer that specifies the maximum field width.
- **[CX]** An optional assignment-allocation character 'm'.
- An option length modifier that specifies the size of the receiving object.
- A **conversion specifier** character that specifies the type of conversion to be applied. The valid conversion specifiers are described below.

The `fscanf()` functions shall execute each directive of the format in turn. When all directives have been executed, or if a directive fails (as detailed below), the function shall return. Failures are described as input failures (due to the unavailability of input bytes) or matching failures (due to inappropriate input).

A directive composed of one or more white-space bytes shall be executed by reading input up to the first non-white-space byte, which shall remain unread, or until no more bytes can be read. The directive shall never fail.

A directive that is an ordinary character shall be executed as follows: the next byte shall be read from the input and compared with the byte that comprises the directive; if the comparison shows that they are not equivalent, the directive shall fail, and the differing and subsequent bytes shall remain unread. Similarly, if end-of-file, an encoding error, or a read error prevents a character from being read, the directive shall fail.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each conversion character. A conversion specification shall be executed in the following steps.

Input white-space bytes shall be skipped, unless the conversion specification includes a **[**, **c**, **C**, or **n** conversion specifier.

An item shall be read from the input, unless the conversion specification includes an **n** conversion specifier. An input item shall be 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 specifier) which is an initial subsequence of a matching sequence. The first byte, if any, after the input item shall remain unread. If the length of the input item is 0, the execution of the conversion specification shall fail; 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 specifier, the input item (or, in the case of a `%n` conversion specification, the count of input bytes) shall be 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 shall be 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 `%`, `[CX]` or in the `n`th argument if introduced by the character sequence `"%n$"`. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

**[CX]** The `c`, `s`, and `[` conversion specifiers shall accept an optional assignment-allocation character `'m'`, which shall cause a memory buffer to be allocated to hold the conversion results. If the conversion specifier is `s` or `[`, the allocated buffer shall include space for a terminating null character (or wide character). In such a case, the argument corresponding to the conversion specifier should be a reference to a pointer variable that will receive a pointer to the allocated buffer. The system shall allocate a buffer as if `malloc()` had been called. The application shall be responsible for freeing the memory after usage. If there is insufficient memory to allocate a buffer, the function shall set `errno` to `[ENOMEM]` and a conversion error shall result. If the function returns `EOF`, any memory successfully allocated for parameters using assignment-allocation character `'m'` by this call shall be freed before the function returns.

## Length Modifiers

The length modifiers and their meanings are:

- **hh:** Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **signed char** or **unsigned char**.
- **h:** Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **short** or **unsigned short**.

- I (ell): Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to `long` or `unsigned long`; that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to an argument with type pointer to `double`; or that a following `c`, `s`, or `[` conversion specifier applies to an argument with type pointer to `wchar_t`. **[CX]** If the `'m'` assignment-allocation character is specified, the conversion applies to an argument with the type pointer to a pointer to `wchar_t`.
- II (ell-ell): Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to `long long` or `unsigned long long`.
- j: Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to `intmax_t` or `uintmax_t`.
- z: Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to `size_t` or the corresponding signed integer type.
- t: Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to `ptrdiff_t` or the corresponding `unsigned` type.
- L: Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to an argument with type pointer to `long double`.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

## Conversion Specifiers

The following conversion specifiers are valid:

- `d`: Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of `strtol()` with the value 10 for the `base` argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to `int`.
- `i`: Matches an optionally signed integer, whose format is the same as expected for the subject sequence of `strtol()` with 0 for the `base` argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to `int`.
- `o`: Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of `strtoul()` with the value 8 for the

`base` argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to **unsigned**.

- **u**: Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of `strtoul()` with the value 10 for the `base` argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to **unsigned**.
- **x**: Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of `strtoul()` with the value 16 for the `base` argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to **unsigned**.
- **a, e, f, g**: Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected for the subject sequence of `strtod()`. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to **float**.

If the `fprintf()` family of functions generates character string representations for infinity and NaN (a symbolic entity encoded in floating-point format) to support IEEE Std 754-1985, the `fscanf()` family of functions shall recognize them as input.

- **s**: Matches a sequence of bytes that are not white-space bytes. If the '`m`' assignment-allocation character is not specified, the application shall ensure that the corresponding argument is a pointer to the initial byte of an array of **char**, **signed char**, or **unsigned char** large enough to accept the sequence and a terminating null character code, which shall be added automatically. **[CX]** Otherwise, the application shall ensure that the corresponding argument is a pointer to a pointer to a **char**.

If an `l` (ell) qualifier is present, the input is a sequence of characters that begins in the initial shift state. Each character shall be converted to a wide character as if by a call to the `mbrtowc()` function, with the conversion state described by an **mbstate\_t** object initialized to zero before the first character is converted. If the '`m`' assignment-allocation character is not specified, the application shall ensure that the corresponding argument is a pointer to an array of **wchar\_t** large enough to accept the sequence and the terminating null wide character, which shall be added automatically. **[CX]** Otherwise, the application shall ensure that the corresponding argument is a pointer to a pointer to a **wchar\_t**.

- **[:]**: Matches a non-empty sequence of bytes from a set of expected bytes (the *scanset*). The normal skip over white-space bytes shall be suppressed in this case. If the '`m`' assignment-allocation character is not specified, the application shall ensure that the corresponding argument is a pointer to the

initial byte of an array of **char**, **signed char**, or **unsigned char** large enough to accept the sequence and a terminating null byte, which shall be added automatically. **[CX]** Otherwise, the application shall ensure that the corresponding argument is a pointer to a pointer to a **char**.

If an **l** (ell) qualifier is present, the input is a sequence of characters that begins in the initial shift state. Each character in the sequence shall be converted to a wide character as if by a call to the **mbrtowc()** function, with the conversion state described by an **mbstate\_t** object initialized to zero before the first character is converted. If the **'m'** assignment-allocation character is not specified, the application shall ensure that the corresponding argument is a pointer to an array of **wchar\_t** large enough to accept the sequence and the terminating null wide character, which shall be added automatically. **[CX]** Otherwise, the application shall ensure that the corresponding argument is a pointer to a pointer to a **wchar\_t**.

The conversion specification includes all subsequent bytes in the format string up to and including the matching right-square-bracket (**' ] '**). The bytes between the square brackets (the *scanlist*) comprise the scanset, unless the byte after the left-square-bracket is a circumflex (**' ^ '**), in which case the scanset contains all bytes that do not appear in the scanlist between the circumflex and the right-square-bracket. If the conversion specification begins with **" [ ] "** or **" [ ^ ] "**, the right-square-bracket is included in the scanlist and the next right-square-bracket is the matching right-square-bracket that ends the conversion specification; otherwise, the first right-square-bracket is the one that ends the conversion specification. If a **' - '** is in the scanlist and is not the first character, nor the second where the first character is a **' ^ '**, nor the last character, the behavior is implementation-defined.

- **c**: Matches a sequence of bytes of the number specified by the field width (1 if no field width is present in the conversion specification). No null byte is added. The normal skip over white-space bytes shall be suppressed in this case. If the **'m'** assignment-allocation character is not specified, the application shall ensure that the corresponding argument is a pointer to the initial byte of an array of **char**, **signed char**, or **unsigned char** large enough to accept the sequence. **[CX]** Otherwise, the application shall ensure that the corresponding argument is a pointer to a pointer to a **char**.

If an **l** (ell) qualifier is present, the input shall be a sequence of characters that begins in the initial shift state. Each character in the sequence is converted to a wide character as if by a call to the **mbrtowc()** function, with the conversion state described by an **mbstate\_t** object initialized to zero before the first character is converted. No null wide character is added. If the **'m'** assignment-allocation character is not specified, the application shall ensure that the corresponding argument is a pointer to an array of **wchar\_t** large enough to accept the resulting

sequence of wide characters. **[CX]** Otherwise, the application shall ensure that the corresponding argument is a pointer to a pointer to a **wchar\_t**.

- **p**: Matches an implementation-defined set of sequences, which shall be the same as the set of sequences that is produced by the `%p` conversion specification of the corresponding `fprintf()` functions. The application shall ensure that the corresponding argument is a pointer to a pointer to **void**. The interpretation of the input item is implementation-defined. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise, the behavior of the `%p` conversion specification is undefined.
- **n**: No input is consumed. The application shall ensure that the corresponding argument is a pointer to the integer into which shall be written the number of bytes read from the input so far by this call to the `fscanf()` functions. Execution of a `%n` conversion specification shall not increment the assignment count returned at the completion of execution of the function. No argument shall be converted, but one shall be consumed. If the conversion specification includes an assignment-suppressing character or a field width, the behavior is undefined.
- **C**: **[XSI]** Equivalent to `lc`.
- **S**: **[XSI]** Equivalent to `ls`.
- **%**: Matches a single `'%'` character; no conversion or assignment occurs. The complete conversion specification shall be `%%`.

If a conversion specification is invalid, the behavior is undefined.

The conversion specifiers `A`, `E`, `F`, `G`, and `X` are also valid and shall be equivalent to `a`, `e`, `f`, `g`, and `x`, respectively.

## Execution Rules

If end-of-file is encountered during input, conversion shall be terminated. If end-of-file occurs before any bytes matching the current conversion specification (except for `%n`) have been read (other than leading white-space bytes, where permitted), execution of the current conversion specification shall terminate with an input failure. Otherwise, unless execution of the current conversion specification is terminated with a matching failure, execution of the following conversion specification (if any) shall be terminated with an input failure.

Reaching the end of the string in `sscanf()` shall be equivalent to encountering end-of-file for `fscanf()`.

If conversion terminates on a conflicting input, the offending input is left unread in the input. Any trailing white-space bytes (including newline characters) shall be left unread unless matched by a conversion specification. The success of literal matches and suppressed assignments is only directly determinable via the `%n` conversion specification.

**[CX]** The `fscanf()` and `scanf()` functions may mark the last data access timestamp of the file associated with `stream` for update. The last data access timestamp shall be marked for update by the first successful execution of `fgetc()`, `fgets()`, `fread()`, `getc()`, `getchar()`, `getdelim()`, `getline()`, `fscanf()`, or `scanf()` using `stream` that returns data not supplied by a prior call to `ungetc()`.

## Return Value

Upon successful completion, these functions shall return the number of successfully matched and assigned input items; this number can be zero in the event of an early matching failure. If the input ends before the first conversion (if any) has completed, and without a matching failure having occurred, EOF shall be returned. If an error occurs before the first conversion (if any) has completed, and without a matching failure having occurred, EOF shall be returned **[CX]** and `errno` shall be set to indicate the error. If an error occurs, the error indicator for the stream shall be set.

## Errors

For the conditions under which the `fscanf()` functions fail and may fail, refer to `fgetc()` or `fgetwc()`.

In addition, the `fscanf()` function shall fail if:

- **[EILSEQ] [CX]** Input byte sequence does not form a valid character.
- **[ENOMEM]** Insufficient storage space is available.

In addition, the `fscanf()` function may fail if:

- **[EINVAL] [CX]** There are insufficient arguments.

## Examples

### Basic Example

The call:

```
int i, n; float x; char name[50];
n = scanf("%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E-1 Hamster
```

assigns to `n` the value 3, to `i` the value 25, to `x` the value 5.432, and `name` contains the string `"Hamster"`.

## Advanced Example

The call:

```
int i; float x; char name[50];
(void) scanf("%2d%f%*d %[0123456789]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

assigns 56 to `i`, 789.0 to `x`, skips 0123, and places the string `"56\0"` in `name`. The next call to `getchar()` shall return the character `'a'`.

## Reading Data into an Array

The following call uses `fscanf()` to read three floating-point numbers from standard input into the `input` array.

```
float input[3];
fscanf(stdin, "%f %f %f", input, input+1, input+2);
```

## Application Usage

If the application calling `fscanf()` has any objects of type `wint_t` or `wchar_t`, it must also include the `<wchar.h>` header to have these objects defined.

For functions that allocate memory as if by `malloc()`, the application should release such memory when it is no longer required by a call to `free()`. For `fscanf()`, this is memory allocated via use of the `'m'` assignment-allocation character.

## Rationale

The set of characters allowed in a scanset is limited to single-byte characters. In other similar places, multi-byte characters have been permitted, but for alignment with the ISO C standard, it has not been done here. Applications needing this could use the corresponding wide-character functions to achieve the desired results.

## Future Directions

None.

## See Also

- [2.5 Standard I/O Streams](#)
- [fprintf\(\)](#)
- [getc\(\)](#)
- [setlocale\(\)](#)
- [strtod\(\)](#)
- [strtol\(\)](#)
- [strtoul\(\)](#)
- [wcrtomb\(\)](#)

## XBD References

- [7. Locale](#)
- [<inttypes.h>](#)
- [<langinfo.h>](#)
- [<stdio.h>](#)
- [<wchar.h>](#)

# Change History

## First released in Issue 1.

Derived from Issue 1 of the SVID.

## Issue 5

Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the `l` (ell) qualifier is now defined for the `c`, `s`, and `[` conversion specifiers.

The DESCRIPTION is updated to indicate that if infinity and NaN can be generated by the `fprintf()` family of functions, then they are recognized by the `fscanf()` family.

## Issue 6

The Open Group Corrigenda U021/7 and U028/10 are applied. These correct several occurrences of "characters" in the text which have been replaced with the term "bytes".

The normative text is updated to avoid use of the term "must" for application requirements.

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

- The prototypes for `fscanf()`, `scanf()`, and `sscanf()` are updated.
- The DESCRIPTION is updated.
- The `hh`, `ll`, `j`, `t`, and `z` length modifiers are added.
- The `a`, `A`, and `F` conversion characters are added.

The DESCRIPTION is updated to use the terms "conversion specifier" and "conversion specification" consistently.

## Issue 7

Austin Group Interpretation 1003.1-2001 #170 is applied.

SD5-XSH-ERN-9 is applied, correcting `fscanf()` to `scanf()` in the DESCRIPTION.

SD5-XSH-ERN-132 is applied, adding the assignment-allocation character `'m'`.

Functionality relating to the `%n$` form of conversion specification is moved from the XSI option to the Base.

Changes are made related to support for fine-grained timestamps.

The APPLICATION USAGE section is updated to clarify that memory is allocated as if by `malloc()`.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0185 [302], XSH/TC1-2008/0186 [90], and XSH/TC1-2008/0187 [14] are applied. XSH/TC1-2008/0186 [90] changes the second sentence in the RETURN VALUE section to align with expected wording changes in the next revision of the ISO C standard.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0135 [936] is applied.

## Issue 8

Austin Group Defect 1163 is applied, clarifying the handling of white space in the format string.

Austin Group Defect 1173 is applied, clarifying the description of the assignment-allocation character `'m'`.

Austin Group Defect 1302 is applied, aligning these functions with the ISO/IEC 9899:2018 standard.

Austin Group Defect 1330 is applied, removing obsolescent interfaces.

Austin Group Defect 1375 is applied, changing "terminating null character" to "terminating null character (or wide character)".

Austin Group Defect 1562 is applied, clarifying that it is the application's responsibility to ensure that the format is a character string, beginning and ending in its initial shift state, if any.

Austin Group Defect 1624 is applied, changing the RETURN VALUE section.

## 1.226. strcat

---

### SYNOPSIS

```
#include <string.h>

char *strcat(char *restrict s1, const char *restrict s2);
```

### DESCRIPTION

The `strcat()` function shall append a copy of the string pointed to by `s2` (including the terminating NUL character) to the end of the string pointed to by `s1`. The initial byte of `s2` overwrites the NUL character at the end of `s1`. If copying takes place between objects that overlap, the behavior is undefined.

The `strcat()` function shall not change the setting of `errno` on valid input.

### RETURN VALUE

The `strcat()` function shall return `s1`; no return value is reserved to indicate an error.

### ERRORS

No errors are defined.

---

*The following sections are informative.*

### EXAMPLES

None.

### APPLICATION USAGE

This version is aligned with the ISO C standard; this does not affect compatibility with XPG3 applications. Reliable error detection by this function was never

guaranteed.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `strncat()`
- XBD `<string.h>`

## CHANGE HISTORY

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

### Issue 6

The `strcat()` prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

### Issue 8

Austin Group Defect 448 is applied, adding a requirement that `strcat()` does not change the setting of `errno` on valid input.

---

## 1.227. strchr

---

### SYNOPSIS

```
#include <string.h>

char *strchr(const char *s, int c);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `strchr()` function shall locate the first occurrence of `c` (converted to a `char`) in the string pointed to by `s`. The terminating NUL character is considered to be part of the string.

[CX] The `strchr()` function shall not change the setting of `errno` on valid input.

### RETURN VALUE

Upon completion, `strchr()` shall return a pointer to the byte, or a null pointer if the byte was not found.

### ERRORS

No errors are defined.

---

*The following sections are informative.*

### EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `strrchr()`
- XBD `<string.h>`

## CHANGE HISTORY

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

### Issue 6

Extensions beyond the ISO C standard are marked.

### Issue 8

Austin Group Defect 448 is applied, adding a requirement that `strchr()` does not change the setting of `errno` on valid input.

## 1.228. strcmp — compare two strings

---

### SYNOPSIS

```
#include <string.h>

int strcmp(const char *s1, const char *s2);
```

### DESCRIPTION

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `strcmp()` function shall compare the string pointed to by `s1` to the string pointed to by `s2`.

The sign of a non-zero return value shall be determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type `unsigned char`) that differ in the strings being compared.

The `strcmp()` function shall not change the setting of `errno` on valid input.

### RETURN VALUE

Upon completion, `strcmp()` shall return an integer greater than, equal to, or less than 0, if the string pointed to by `s1` is greater than, equal to, or less than the string pointed to by `s2`, respectively.

### ERRORS

No errors are defined.

---

# EXAMPLES

## Checking a Password Entry

The following example compares the information read from standard input to the value of the name of the user entry. If the `strcmp()` function returns 0 (indicating a match), a further check will be made to see if the user entered the proper old password. The `crypt()` function shall encrypt the old password entered by the user, using the value of the encrypted password in the `passwd` structure as the salt. If this value matches the value of the encrypted `passwd` in the structure, the entered password `oldpasswd` is the correct user's password. Finally, the program encrypts the new password so that it can store the information in the `passwd` structure.

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

...
int valid_change;
struct passwd *p;
char user[100];
char oldpasswd[100];
char newpasswd[100];
char savepasswd[100];

...
if (strcmp(p->pw_name, user) == 0) {
    if (strcmp(p->pw_passwd, crypt(oldpasswd, p->pw_passwd)) == 0)
        strcpy(savepasswd, crypt(newpasswd, user));
    p->pw_passwd = savepasswd;
    valid_change = 1;
}
else {
    fprintf(stderr, "Old password is not valid\n");
}
...
...
```

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `strncmp()`
- `<string.h>`

## CHANGE HISTORY

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

### Issue 6

Extensions beyond the ISO C standard are marked.

### Issue 8

Austin Group Defect 448 is applied, adding a requirement that `strcmp()` does not change the setting of `errno` on valid input.

---

## 1.229. strcoll

---

### SYNOPSIS

```
#include <string.h>

int strcoll(const char *s1, const char *s2);

int strcoll_l(const char *s1, const char *s2, locale_t locale);
```

### DESCRIPTION

For `strcoll()`: The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `strcoll()` and `strcoll_l()` functions shall compare the string pointed to by `s1` to the string pointed to by `s2`, both interpreted as appropriate to the `LC_COLLATE` category of the current locale, or of the locale represented by `locale`, respectively.

The `strcoll()` and `strcoll_l()` functions shall not change the setting of `errno` if successful.

Since 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()`, or `strcoll_l()`, then check `errno`.

The behavior is undefined if the `locale` argument to `strcoll_l()` is the special locale object `LC_GLOBAL_LOCALE` or is not a valid locale object handle.

### RETURN VALUE

Upon successful completion, `strcoll()` shall return an integer greater than, equal to, or less than 0, according to whether the string pointed to by `s1` is greater than, equal to, or less than the string pointed to by `s2` when both are interpreted as appropriate to the current locale. On error, `strcoll()` may set `errno`, but no return value is reserved to indicate an error.

Upon successful completion, `strcoll_l()` shall return an integer greater than, equal to, or less than 0, according to whether the string pointed to by `s1` is

greater than, equal to, or less than the string pointed to by `s2` when both are interpreted as appropriate to the locale represented by `locale`. On error, `strcoll_l()` may set `errno`, but no return value is reserved to indicate an error.

## ERRORS

These functions may fail if:

- **[EINVAL]**
- The `s1` or `s2` arguments contain characters outside the domain of the collating sequence.

## EXAMPLES

### Comparing Nodes

The following example uses an application-defined function, `node_compare()`, to compare two nodes based on an alphabetical ordering of the `string` field.

```
#include <string.h>
...
struct node { /* These are stored in the table. */
    char *string;
    int length;
};
...
int node_compare(const void *node1, const void *node2)
{
    return strcoll(((const struct node *)node1)->string,
                  ((const struct node *)node2)->string);
}
...
```

## APPLICATION USAGE

The `strxfrm()` and `strcmp()` functions should be used for sorting large lists.

## RATIONALE

None.

# FUTURE DIRECTIONS

None.

## SEE ALSO

- `alphasort()`
- `strcmp()`
- `strxfrm()`

XBD `<string.h>`

## CHANGE HISTORY

First released in Issue 3.

### Issue 5

The DESCRIPTION is updated to indicate that `errno` does not change if the function is successful.

### Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EINVAL] optional error condition is added.

An example is added.

### Issue 7

The `strcoll_l()` function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0593 [283] and XSH/TC1-2008/0594 [283] are applied.

---

## 1.230. strcpy

---

### SYNOPSIS

```
#include <string.h>

[CX] char *stpcpy(char *restrict s1, const char *restrict s2);
char *strcpy(char *restrict s1, const char *restrict s2);
```

### DESCRIPTION

For `strcpy()` : [CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The [CX] `stpcpy()` and `strcpy()` functions shall copy the string pointed to by `s2` (including the terminating NUL character) into the array pointed to by `s1`.

If copying takes place between objects that overlap, the behavior is undefined.

[CX] The `strcpy()` and `stpcpy()` functions shall not change the setting of `errno` on valid input.

### RETURN VALUE

[CX] The `stpcpy()` function shall return a pointer to the terminating NUL character copied into the `s1` buffer.

The `strcpy()` function shall return `s1`.

No return values are reserved to indicate an error.

### ERRORS

No errors are defined.

---

*The following sections are informative.*

# EXAMPLES

## Construction of a Multi-Part Message in a Single Buffer

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

int
main (void)
{
    char buffer[10];
    char *name = buffer;

    name = stpcpy (stpcpy (stpcpy (name, "ice"), "-"), "cream");
    puts (buffer);
    return 0;
}
```

## Initializing a String

The following example copies the string "-----" into the `permstring` variable.

```
#include <string.h>
...
static char permstring[11];
...
strcpy(permstring, "-----");
...
```

## Storing a Key and Data

The following example allocates space for a key using `malloc()` then uses `strcpy()` to place the key there. Then it allocates space for data using `malloc()`, and uses `strcpy()` to place data there. (The user-defined function `dbfree()` frees memory previously allocated to an array of type `struct element *`.)

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
...
/* Structure used to read data and store it. */
struct element {
    char *key;
    char *data;
```

```
};

struct element *tbl, *curtbl;
char *key, *data;
int count;

...
void dbfree(struct element *, int);

...
if ((curtbl->key = malloc(strlen(key) + 1)) == NULL) {
    perror("malloc");
    dbfree(tbl, count);
    return NULL;
}
strcpy(curtbl->key, key);

if ((curtbl->data = malloc(strlen(data) + 1)) == NULL) {
    perror("malloc");
    free(curtbl->key);
    dbfree(tbl, count);
    return NULL;
}
strcpy(curtbl->data, data);
...
```

## APPLICATION USAGE

Character movement is performed differently in different implementations. Thus, overlapping moves may yield surprises.

This version is aligned with the ISO C standard; this does not affect compatibility with XPG3 applications. Reliable error detection by this function was never guaranteed.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [strncpy\(\)](#)
- [wcscpy\(\)](#)
- XBD [`<string.h>`](#)

# CHANGE HISTORY

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

## Issue 6

The `strcpy()` prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

## Issue 7

The `stpcpy()` function is added from The Open Group Technical Standard, 2006, Extended API Set Part 1.

## Issue 8

Austin Group Defect 448 is applied, adding a requirement that `strcpy()` and `stpcpy()` do not change the setting of `errno` on valid input.

Austin Group Defect 1787 is applied, changing the NAME section.

---

## 1.231. strcspn

---

### SYNOPSIS

```
#include <string.h>

size_t strcspn(const char *s1, const char *s2);
```

### DESCRIPTION

The `strcspn()` function shall compute the length (in bytes) of the maximum initial segment of the string pointed to by `s1` which consists entirely of bytes **not** from the string pointed to by `s2`.

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `strcspn()` function shall not change the setting of `errno` on valid input.

### RETURN VALUE

The `strcspn()` function shall return the length of the computed segment of the string pointed to by `s1`; no return value is reserved to indicate an error.

### ERRORS

No errors are defined.

---

*The following sections are informative.*

### EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `strspn()`
- XBD `<string.h>`

## CHANGE HISTORY

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

### Issue 5

The RETURN VALUE section is updated to indicate that `strcspn()` returns the length of `s1`, and not `s1` itself as was previously stated.

### Issue 6

The Open Group Corrigendum U030/1 is applied. The text of the RETURN VALUE section is updated to indicate that the computed segment length is returned, not the `s1` length.

### Issue 8

Austin Group Defect 448 is applied, adding a requirement that `strcspn()` does not change the setting of `errno` on valid input.

---

## 1.232. strerror, strerror\_l, strerror\_r — get error message string

---

### SYNOPSIS

```
#include <string.h>

char *strerror(int errnum);

/* XSI extension */
char *strerror_l(int errnum, locale_t locale);
int strerror_r(int errnum, char *strerrbuf, size_t buflen);
```

### DESCRIPTION

For strerror():

- The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The strerror() function shall map the error number in errnum to a locale-dependent error message string and shall return a pointer to it. Typically, the values for errnum come from errno, but strerror() shall map any value of type int to a message.

The application shall not modify the string returned. The returned string pointer might be invalidated or the string content might be overwritten by a subsequent call to strerror(), or by a subsequent call to strerror\_l() in the same thread. The returned pointer and the string content might also be invalidated if the calling thread is terminated.

The string may be overwritten by a subsequent call to strerror\_l() in the same thread.

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.

The implementation shall behave as if no function defined in this volume of POSIX.1-2024 calls strerror().

The strerror() and strerror\_l() functions shall not change the setting of errno if successful.

Since no return value is reserved to indicate an error of `strerror()`, an application wishing to check for error situations should set `errno` to 0, then call `strerror()`, then check `errno`. Similarly, since `strerror_l()` is required to return a string for some errors, an application wishing to check for all error situations should set `errno` to 0, then call `strerror_l()`, then check `errno`.

The `strerror()` function need not be thread-safe; however, `strerror()` shall avoid data races with all other functions.

The `strerror_l()` function shall map the error number in `errnum` to a locale-dependent error message string in the locale represented by `locale` and shall return a pointer to it.

The `strerror_r()` function shall map the error number in `errnum` to a locale-dependent error message string and shall return the string in the buffer pointed to by `strerrbuf`, with length `buflen`.

If the value of `errnum` is a valid error number, the message string shall indicate what error occurred; if the value of `errnum` is zero, the message string shall either be an empty string or indicate that no error occurred; otherwise, if these functions complete successfully, the message string shall indicate that an unknown error occurred.

The behavior is undefined if the `locale` argument to `strerror_l()` is the special locale object `LC_GLOBAL_LOCALE` or is not a valid locale object handle.

## RETURN VALUE

Upon completion, whether successful or not, `strerror()` shall return a pointer to the generated message string. On error `errno` may be set, but no return value is reserved to indicate an error.

Upon successful completion, `strerror_l()` shall return a pointer to the generated message string. If `errnum` is not a valid error number, `errno` may be set to `[EINVAL]`, but a pointer to a message string shall still be returned. If any other error occurs, `errno` shall be set to indicate the error and a null pointer shall be returned.

Upon successful completion, `strerror_r()` shall return 0. Otherwise, an error number shall be returned to indicate the error.

## ERRORS

These functions may fail if:

- **EINVAL** - The value of `errnum` is neither a valid error number nor zero.

The `strerror_r()` function shall fail if:

- **ERANGE** - Insufficient storage was supplied via strerrbuf and buflen to contain the generated message string.

## EXAMPLES

None.

## APPLICATION USAGE

Historically in some implementations, calls to perror() would overwrite the string that the pointer returned by strerror() points to. Such implementations did not conform to the ISO C standard; however, application developers should be aware of this behavior if they wish their applications to be portable to such implementations.

Applications should use strerror\_l() rather than strerror() or strerror\_r() to avoid thread safety and possible alternative (non-conforming) versions of these functions in some implementations.

## RATIONALE

The strerror\_l() function is required to be thread-safe, thereby eliminating the need for an equivalent to the strerror\_r() function.

Earlier versions of this standard did not explicitly require that the error message strings returned by strerror() and strerror\_r() provide any information about the error. This version of the standard requires a meaningful message for any successful completion.

Since no return value is reserved to indicate a strerror() error, but all calls (whether successful or not) must return a pointer to a message string, on error strerror() can return a pointer to an empty string or a pointer to a meaningful string that can be printed.

Note that the [EINVAL] error condition is a may fail error. If an invalid error number is supplied as the value of errnum, applications should be prepared to handle any of the following:

1. **Error (with no meaningful message):** errno is set to [EINVAL], the return value is a pointer to an empty string.
2. **Successful completion:** errno is unchanged and the return value points to a string like "unknown error" or "error number xxx" (where xxx is the value of errnum).

3. **Combination of #1 and #2:** `errno` is set to `[EINVAL]` and the return value points to a string like "unknown error" or "error number xxx" (where `xxx` is the value of `errnum`). Since applications frequently use the return value of `strerror()` as an argument to functions like `fprintf()` (without checking the return value) and since applications have no way to parse an error message string to determine whether `errnum` represents a valid error number, implementations are encouraged to implement #3. Similarly, implementations are encouraged to have `strerror_r()` return `[EINVAL]` and put a string like "unknown error" or "error number xxx" in the buffer pointed to by `strerrbuf` when the value of `errnum` is not a valid error number.

Additionally, implementations are encouraged to null terminate `strerrbuf` when failing with `[ERANGE]` for any size other than `buflen` of zero.

Some applications rely on being able to set `errno` to 0 before calling a function with no reserved value to indicate an error, then call `strerror(errno)` afterwards to detect whether an error occurred (because `errno` changed) or to indicate success (because `errno` remained zero). This usage pattern requires that `strerror(0)` succeed with useful results. Previous versions of the standard did not specify the behavior when `errnum` is zero.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [perror\(\)](#)
- 

## CHANGE HISTORY

First released in Issue 3.

### Issue 5

The DESCRIPTION is updated to indicate that `errno` is not changed if the function is successful.

A note indicating that the `strerror()` function need not be reentrant is added to the DESCRIPTION.

## Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE section, the fact that `errno` may be set is added.
- The [EINVAL] optional error condition is added.

The normative text is updated to avoid use of the term "must" for application requirements.

The `strerror_r()` function is added in response to IEEE PASC Interpretation 1003.1c #39.

The `strerror_r()` function is marked as part of the Thread-Safe Functions option.

## Issue 7

Austin Group Interpretation 1003.1-2001 #072 is applied, updating the ERRORS section.

Austin Group Interpretation 1003.1-2001 #156 is applied.

Austin Group Interpretation 1003.1-2001 #187 is applied, clarifying the behavior when the generated error message is an empty string.

SD5-XSH-ERN-191 is applied, updating the APPLICATION USAGE section.

The `strerror_l()` function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

The `strerror_r()` function is moved from the Thread-Safe Functions option to the Base.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0595 [75], XSH/TC1-2008/0596 [447], XSH/TC1-2008/0597 [382,428], XSH/TC1-2008/0598 [283], XSH/TC1-2008/0599 [382,428], XSH/TC1-2008/0600 [283], and XSH/TC1-2008/0601 [382,428] are applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0339 [656] is applied.

## Issue 8

Austin Group Defect 398 is applied, changing the [ERANGE] error from "may fail" to "shall fail".

Austin Group Defect 655 is applied, changing the APPLICATION USAGE section.

Austin Group Defect 1302 is applied, aligning the strerror() function with the ISO/IEC 9899:2018 standard.

---

## 1.233. strerror, strerror\_l, strerror\_r — get error message string

---

### SYNOPSIS

```
#include <string.h>

char *strerror(int errnum);

[CX] char *strerror_l(int errnum, locale_t locale);
int strerror_r(int errnum, char *strerrbuf, size_t buflen);
```

### DESCRIPTION

For `strerror()`: [CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `strerror()` function shall map the error number in `errnum` to a locale-dependent error message string and shall return a pointer to it. Typically, the values for `errnum` come from `errno`, but `strerror()` shall map any value of type `int` to a message.

The application shall not modify the string returned. [CX] The returned string pointer might be invalidated or the string content might be overwritten by a subsequent call to `strerror()`, [CX] or by a subsequent call to `strerror_l()` in the same thread. The returned pointer and the string content might also be invalidated if the calling thread is terminated.

[CX] The string may be overwritten by a subsequent call to `strerror_l()` in the same thread.

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.

The implementation shall behave as if no function defined in this volume of POSIX.1-2024 calls `strerror()`.

[CX] The `strerror()` and `strerror_l()` functions shall not change the setting of `errno` if successful.

Since no return value is reserved to indicate an error of `strerror()`, an application wishing to check for error situations should set `errno` to 0, then call

`strerror()`, then check `errno`. Similarly, since `strerror_l()` is required to return a string for some errors, an application wishing to check for all error situations should set `errno` to 0, then call `strerror_l()`, then check `errno`.

The `strerror()` function need not be thread-safe; however, `strerror()` shall avoid data races with all other functions.

[CX] The `strerror_l()` function shall map the error number in `errnum` to a locale-dependent error message string in the locale represented by `locale` and shall return a pointer to it.

The `strerror_r()` function shall map the error number in `errnum` to a locale-dependent error message string and shall return the string in the buffer pointed to by `strerrbuf`, with length `buflen`.

[CX] If the value of `errnum` is a valid error number, the message string shall indicate what error occurred; if the value of `errnum` is zero, the message string shall either be an empty string or indicate that no error occurred; otherwise, if these functions complete successfully, the message string shall indicate that an unknown error occurred.

[CX] The behavior is undefined if the `locale` argument to `strerror_l()` is the special locale object `LC_GLOBAL_LOCALE` or is not a valid locale object handle.

## RETURN VALUE

Upon completion, whether successful or not, `strerror()` shall return a pointer to the generated message string. [CX] On error `errno` may be set, but no return value is reserved to indicate an error.

Upon successful completion, `strerror_l()` shall return a pointer to the generated message string. If `errnum` is not a valid error number, `errno` may be set to `[EINVAL]`, but a pointer to a message string shall still be returned. If any other error occurs, `errno` shall be set to indicate the error and a null pointer shall be returned.

Upon successful completion, `strerror_r()` shall return 0. Otherwise, an error number shall be returned to indicate the error.

## ERRORS

These functions may fail if:

- **[EINVAL]** [CX] The value of `errnum` is neither a valid error number nor zero.

The `strerror_r()` function shall fail if:

- **[ERANGE]** [CX] Insufficient storage was supplied via `strerrbuf` and `buflen` to contain the generated message string.

## EXAMPLES

None.

## APPLICATION USAGE

Historically in some implementations, calls to `perror()` would overwrite the string that the pointer returned by `strerror()` points to. Such implementations did not conform to the ISO C standard; however, application developers should be aware of this behavior if they wish their applications to be portable to such implementations.

Applications should use `strerror_l()` rather than `strerror()` or `strerror_r()` to avoid thread safety and possible alternative (non-conforming) versions of these functions in some implementations.

## RATIONALE

The `strerror_l()` function is required to be thread-safe, thereby eliminating the need for an equivalent to the `strerror_r()` function.

Earlier versions of this standard did not explicitly require that the error message strings returned by `strerror()` and `strerror_r()` provide any information about the error. This version of the standard requires a meaningful message for any successful completion.

Since no return value is reserved to indicate a `strerror()` error, but all calls (whether successful or not) must return a pointer to a message string, on error `strerror()` can return a pointer to an empty string or a pointer to a meaningful string that can be printed.

Note that the **[EINVAL]** error condition is a may fail error. If an invalid error number is supplied as the value of `errnum`, applications should be prepared to handle any of the following:

1. Error (with no meaningful message): `errno` is set to **[EINVAL]**, the return value is a pointer to an empty string.
2. Successful completion: `errno` is unchanged and the return value points to a string like "unknown error" or "error number xxx" (where `xxx` is the value of `errnum`).

3. Combination of #1 and #2: `errno` is set to [EINVAL] and the return value points to a string like "unknown error" or "error number xxx" (where `xxx` is the value of `errnum`). Since applications frequently use the return value of `strerror()` as an argument to functions like `fprintf()` (without checking the return value) and since applications have no way to parse an error message string to determine whether `errnum` represents a valid error number, implementations are encouraged to implement #3. Similarly, implementations are encouraged to have `strerror_r()` return [EINVAL] and put a string like "unknown error" or "error number xxx" in the buffer pointed to by `strerrbuf` when the value of `errnum` is not a valid error number.

Additionally, implementations are encouraged to null terminate `strerrbuf` when failing with [ERANGE] for any size other than `buflen` of zero.

Some applications rely on being able to set `errno` to 0 before calling a function with no reserved value to indicate an error, then call `strerror(errno)` afterwards to detect whether an error occurred (because `errno` changed) or to indicate success (because `errno` remained zero). This usage pattern requires that `strerror(0)` succeed with useful results. Previous versions of the standard did not specify the behavior when `errnum` is zero.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `perror()`
- XBD `<string.h>`

## CHANGE HISTORY

First released in Issue 3.

### Issue 5

The DESCRIPTION is updated to indicate that `errno` is not changed if the function is successful.

A note indicating that the `strerror()` function need not be reentrant is added to the DESCRIPTION.

## Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE section, the fact that `errno` may be set is added.
- The [EINVAL] optional error condition is added.

The normative text is updated to avoid use of the term "must" for application requirements.

The `strerror_r()` function is added in response to IEEE PASC Interpretation 1003.1c #39.

The `strerror_r()` function is marked as part of the Thread-Safe Functions option.

## Issue 7

Austin Group Interpretation 1003.1-2001 #072 is applied, updating the ERRORS section.

Austin Group Interpretation 1003.1-2001 #156 is applied.

Austin Group Interpretation 1003.1-2001 #187 is applied, clarifying the behavior when the generated error message is an empty string.

SD5-XSH-ERN-191 is applied, updating the APPLICATION USAGE section.

The `strerror_l()` function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

The `strerror_r()` function is moved from the Thread-Safe Functions option to the Base.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0595 [75], XSH/TC1-2008/0596 [447], XSH/TC1-2008/0597 [382,428], XSH/TC1-2008/0598 [283], XSH/TC1-2008/0599 [382,428], XSH/TC1-2008/0600 [283], and XSH/TC1-2008/0601 [382,428] are applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0339 [656] is applied.

## Issue 8

Austin Group Defect 398 is applied, changing the [ERANGE] error from "may fail" to "shall fail".

Austin Group Defect 655 is applied, changing the APPLICATION USAGE section.

Austin Group Defect 1302 is applied, aligning the `strerror()` function with the ISO/IEC 9899:2018 standard.

---

## 1.234. strftime, strftime\_l — convert date and time to a string

### SYNOPSIS

```
#include <time.h>

size_t strftime(char *restrict s, size_t maxsize,
                const char *restrict format, const struct tm *restrict
                tm);

[CX] size_t strftime_l(char *restrict s, size_t maxsize,
                       const char *restrict format, const struct tm
                       locale_t locale);
```

### DESCRIPTION

For `strftime()`: [CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `strftime()` function shall place bytes into the array pointed to by `s` as controlled by the string pointed to by `format`. The application shall ensure that the format is a character string, beginning and ending in its initial shift state, if any. The format string consists of zero or more conversion specifications and ordinary characters.

Each conversion specification is introduced by the '%' character after which the following appear in sequence:

- [CX] An optional flag:
  - `0` - The zero character ('0'), which specifies that the character used as the padding character is '0'
  - `+` - The character ('+'), which specifies that the character used as the padding character is '0', and that if and only if the field being produced consumes more than four bytes to represent a year (for %F, %G, or %Y) or more than two bytes to represent the year divided by 100 (for %C) then a leading character shall be included if the year being processed is greater than or equal to zero or a leading character ('-') shall be included if the year is less than zero.

- The default padding character is unspecified.
- An optional minimum field width. If the converted value, including any leading '+' or '-' sign, has fewer bytes than the minimum field width and the padding character is not the NUL character, the output shall be padded on the left (after any leading '+' or '-' sign) with the padding character.
- An optional E or O modifier.
- A terminating conversion specifier character that indicates the type of conversion to be applied.

[CX] The results are unspecified if more than one flag character is specified, a flag character is specified without a minimum field width; a minimum field width is specified without a flag character; a modifier is specified with a flag or with a minimum field width; or if a minimum field width is specified for any conversion specifier other than C, F, G, or Y.

All ordinary characters (including the terminating NUL character) are copied unchanged into the array. If copying takes place between objects that overlap, the behavior is undefined. No more than `maxsize` bytes are placed into the array. Each conversion specifier is replaced by appropriate characters as described in the following list. The appropriate characters are determined using the `LC_TIME` category of the current locale and by the values of zero or more members of the broken-down time structure pointed to by `timeptr`, as specified in brackets in the description. If any of the specified values are outside the normal range, the characters stored are unspecified.

[CX] The `strftime_l()` function shall be equivalent to the `strftime()` function, except that the locale data used is from the locale represented by `locale`.

Local timezone information shall be set as though `strftime()` called `tzset()`.

## Conversion Specifiers

The following conversion specifiers shall be supported:

`%a` - Replaced by the locale's abbreviated weekday name. [ `tm_wday` ]

`%A` - Replaced by the locale's full weekday name. [ `tm_wday` ]

`%b` - Replaced by the locale's abbreviated month name. [ `tm_mon` ]

`%B` - Replaced by the locale's full month name. [ `tm_mon` ]

`%c` - Replaced by the locale's appropriate date and time representation.

`%C` - Replaced by the year divided by 100 and truncated to an integer, as a decimal number. [ `tm_year` ]

- If a minimum field width is not specified:
  - If the year is between 0 and 9999 inclusive, two characters shall be placed into the array pointed to by `s`, including a leading '0' if there would otherwise be only a single digit.
  - [CX] If the year is less than 0 or greater than 9999, the number of characters placed into the array pointed to by `s` shall be the number of digits and leading sign characters (if any) in the result of dividing the year by 100 and truncating, or two, whichever is greater.

**%d** - Replaced by the day of the month as a decimal number [01,31]. [ `tm_mday` ]

**%D** - Equivalent to **%m/%d/%y**. [ `tm_mon` , `tm_mday` , `tm_year` ]

**%e** - Replaced by the day of the month as a decimal number [1,31]; a single digit is preceded by a space. [ `tm_mday` ]

**%F** - Equivalent to **%Y-%m-%d** if no flag and no minimum field width are specified. (For years between 1000 and 9999 inclusive this provides the ISO 8601:2019 standard complete representation, extended format date representation of a specific day.) [ `tm_year` , `tm_mon` , `tm_mday` ]

**%g** - Replaced by the last 2 digits of the week-based year as a decimal number [00,99]. [ `tm_year` , `tm_wday` , `tm_yday` ]

**%G** - Replaced by the week-based year as a decimal number (for example, 1977). [ `tm_year` , `tm_wday` , `tm_yday` ]

**%h** - Equivalent to **%b**. [ `tm_mon` ]

**%H** - Replaced by the hour (24-hour clock) as a decimal number [00,23]. [ `tm_hour` ]

**%I** - Replaced by the hour (12-hour clock) as a decimal number [01,12]. [ `tm_hour` ]

**%j** - Replaced by the day of the year as a decimal number [001,366]. [ `tm_yday` ]

**%m** - Replaced by the month as a decimal number [01,12]. [ `tm_mon` ]

**%M** - Replaced by the minute as a decimal number [00,59]. [ `tm_min` ]

**%n** - Replaced by a .

**%p** - Replaced by the locale's equivalent of either a.m. or p.m. [ `tm_hour` ]

**%r** - Replaced by the time in 12-hour clock notation; [CX] if the 12-hour format is not supported in the locale, this shall be either an empty string or the time in a 24-hour clock notation. In the POSIX locale this shall be equivalent to **%I:%M:%S %p**. [ `tm_hour` , `tm_min` , `tm_sec` ]

**%R** - Replaced by the time in 24-hour notation (%H:%M). [ `tm_hour` , `tm_min` ]

**%s** - [CX] Replaced by the number of seconds since the Epoch as a decimal number, calculated as described for `mktme()`. [ `tm_year` , `tm_mon` , `tm_mday` , `tm_hour` , `tm_min` , `tm_sec` , `tm_isdst` ]

**%S** - Replaced by the second as a decimal number [00,60]. [ `tm_sec` ]

**%t** - Replaced by a .

**%T** - Replaced by the time (%H:%M:%S). [ `tm_hour` , `tm_min` , `tm_sec` ]

**%u** - Replaced by the weekday as a decimal number [1,7], with 1 representing Monday. [ `tm_wday` ]

**%U** - Replaced by the week number of the year as a decimal number [00,53]. The first Sunday of January is the first day of week 1; days in the new year before this are in week 0. [ `tm_year` , `tm_wday` , `tm_yday` ]

**%V** - Replaced by the week number of the year (Monday as the first day of the week) as a decimal number [01,53]. If the week containing 1 January has four or more days in the new year, then it is considered week 1. Otherwise, it is the last week of the previous year, and the next week is week 1. Both January 4th and the first Thursday of January are always in week 1. [ `tm_year` , `tm_wday` , `tm_yday` ]

**%w** - Replaced by the weekday as a decimal number [0,6], with 0 representing Sunday. [ `tm_wday` ]

**%W** - Replaced by the week number of the year as a decimal number [00,53]. The first Monday of January is the first day of week 1; days in the new year before this are in week 0. [ `tm_year` , `tm_wday` , `tm_yday` ]

**%x** - Replaced by the locale's appropriate date representation.

**%X** - Replaced by the locale's appropriate time representation.

**%y** - Replaced by the last two digits of the year as a decimal number [00,99]. [ `tm_year` ]

**%Y** - Replaced by the year as a decimal number (for example, 1997). [ `tm_year` ]

**%z** - Replaced by the offset from UTC in the ISO 8601:2019 standard format (+hhmm or -hhmm), or by no characters if no timezone is determinable. For example, "-0430" means 4 hours 30 minutes behind UTC (west of Greenwich). [CX] If `tm_isdst` is zero, the standard time offset is used. If `tm_isdst` is greater than zero, the daylight saving time offset is used. If `tm_isdst` is negative, no characters are returned. [ `tm_isdst` , [CX] `tm_gmtoff` ]

**%Z** - Replaced by the timezone name or abbreviation, or by no bytes if no timezone information exists. [ `tm_isdst` , [CX] `tm_zone` ]

**%%** - Replaced by %.

If a conversion specification does not correspond to any of the above, the behavior is undefined.

[CX] If a `struct tm` broken-down time structure is created by `localtime()` or `localtime_r()`, or modified by `mktime()`, and the value of `TZ` is subsequently modified, the results of the `%Z` and `%z` `strftime()` conversion specifiers are undefined, when `strftime()` is called with such a broken-down time structure.

If a `struct tm` broken-down time structure is created or modified by `gmtime()` or `gmtime_r()`, it is unspecified whether the result of the `%Z` and `%z` conversion specifiers shall refer to UTC or the current local timezone, when `strftime()` is called with such a broken-down time structure.

## Modified Conversion Specifiers

Some conversion specifiers can be modified by the E or O modifier characters to indicate that an alternative format or specification should be used rather than the one normally used by the unmodified conversion specifier. If the alternative format or specification does not exist for the current locale, the behavior shall be as if the unmodified conversion specification were used.

**%Ec** - Replaced by the locale's alternative appropriate date and time representation.

**%EC** - Replaced by the name of the base year (period) in the locale's alternative representation.

**%Ex** - Replaced by the locale's alternative date representation.

**%EX** - Replaced by the locale's alternative time representation.

**%Ey** - Replaced by the offset from `%EC` (year only) in the locale's alternative representation.

**%EY** - Replaced by the full alternative year representation.

**%Ob** - [CX] Replaced by the locale's abbreviated alternative month name.

**%OB** - [CX] Replaced by the locale's alternative appropriate full month name.

**%Od** - Replaced by the day of the month, using the locale's alternative numeric symbols, filled as needed with leading zeros if there is any alternative symbol for zero; otherwise, with leading characters.

**%Oe** - Replaced by the day of the month, using the locale's alternative numeric symbols, filled as needed with leading characters.

**%OH** - Replaced by the hour (24-hour clock) using the locale's alternative numeric symbols.

**%OI** - Replaced by the hour (12-hour clock) using the locale's alternative numeric symbols.

**%Om** - Replaced by the month using the locale's alternative numeric symbols.

**%OM** - Replaced by the minutes using the locale's alternative numeric symbols.

**%OS** - Replaced by the seconds using the locale's alternative numeric symbols.

**%Ou** - Replaced by the weekday as a number in the locale's alternative representation (Monday=1).

**%OU** - Replaced by the week number of the year (Sunday as the first day of the week, rules corresponding to %U) using the locale's alternative numeric symbols.

**%OV** - Replaced by the week number of the year (Monday as the first day of the week, rules corresponding to %V) using the locale's alternative numeric symbols.

**%Ow** - Replaced by the number of the weekday (Sunday=0) using the locale's alternative numeric symbols.

**%OW** - Replaced by the week number of the year (Monday as the first day of the week) using the locale's alternative numeric symbols.

**%Oy** - Replaced by the year (offset from %C) using the locale's alternative numeric symbols.

%g, %G, and %V give values according to the ISO 8601:2019 standard week-based year. In this system, weeks begin on a Monday and week 1 of the year is the week that includes January 4th, which is also the week that includes the first Thursday of the year, and is also the first week that contains at least four days in the year. If the first Monday of January is the 2nd, 3rd, or 4th, the preceding days are part of the last week of the preceding year; thus, for Saturday 2nd January 1999, %G is replaced by 1998 and %V is replaced by 53. If December 29th, 30th, or 31st is a Monday, it and any following days are part of week 1 of the following year. Thus, for Tuesday 30th December 1997, %G is replaced by 1998 and %V is replaced by 01.

[CX] The behavior is undefined if the `locale` argument to `strftime_l()` is the special locale object `LC_GLOBAL_LOCALE` or is not a valid locale object handle.

## RETURN VALUE

If successful, these functions shall return the number of bytes placed into the array pointed to by `s`, not including the terminating NUL character. [CX] If successful, `errno` shall not be changed. Otherwise, 0 shall be returned, [CX] `errno` shall be set to indicate the error, and the contents of the array are unspecified.

# ERRORS

[CX] These functions shall fail if:

- **[ERANGE]** - The total number of resulting bytes including the terminating NUL character is more than `maxsize`.

These functions may fail if:

- **[EINVAL]** - The `format` string includes a %s conversion and the number of seconds since the Epoch would be negative.
- **[EOVERFLOW]** - The `format` string includes a %s conversion and the number of seconds since the Epoch cannot be represented in a `time_t`.

# EXAMPLES

## Getting a Localized Date String

The following example first sets the locale to the user's default. The locale information will be used in the `nl_langinfo()` and `strftime()` functions. The `nl_langinfo()` function returns the localized date string which specifies how the date is laid out. The `strftime()` function takes this information and, using the `tm` structure for values, places the date and time information into `datestring`.

```
#include <time.h>
#include <locale.h>
#include <langinfo.h>
...
struct tm *tm;
char datestring[256];
...
setlocale (LC_ALL, "");
...
strftime (datestring, sizeof(datestring), nl_langinfo (D_T_FMT), tm)
```

# APPLICATION USAGE

A return value of 0 may indicate either success or failure if a format is the empty string or consists of conversion specifications such as %p or %Z that are replaced by no characters in the current locale or because of the current setting of `tzname[]`, respectively. To distinguish between success and failure when

`strftime()` returns 0, an application can set `errno` to 0 before calling `strftime()` and test whether `errno` is 0 afterwards.

The range of values for %S is [00,60] rather than [00,59] to allow for the occasional leap second.

Some of the conversion specifications are duplicates of others. They are included for compatibility with `nl_cxtime()` and `nl_ascxtime()`, which were published in Issue 2.

The %C, %F, %G, and %Y format specifiers in `strftime()` always print full values, but the `strptime()` %C, %F, and %Y format specifiers only scan two digits (assumed to be the first two digits of a four-digit year) for %C and four digits (assumed to be the entire (four-digit) year) for %F and %Y. This mimics the behavior of `printf()` and `scanf()`.

In the C or POSIX locale, the E and O modifiers are ignored and the replacement strings for the following specifiers are:

- %a - The first three characters of %A.
- %A - One of Sunday, Monday, ..., Saturday.
- %b - The first three characters of %B.
- %B - One of January, February, ..., December.
- %c - Equivalent to %a %b %e %T %Y.
- %p - One of AM or PM.
- %r - Equivalent to %l:%M:%S %p.
- %x - Equivalent to %m/%d/%y.
- %X - Equivalent to %T.
- %Z - Implementation-defined.

## SEE ALSO

`asctime()`, `clock()`, `ctime()`, `difftime()`, `futimens()`, `getdate()`,  
`gmtime()`, `localtime()`, `mktimed()`, `strptime()`, `time()`, `tzset()`,  
`use locale()`

XBD 7.3.5 LC\_TIME, `<time.h>`

## 1.235. `strlen`, `strnlen` — get length of fixed size string

---

### SYNOPSIS

```
#include <string.h>

size_t strlen(const char *s);

[CX] size_t strnlen(const char *s, size_t maxlen);
```

### DESCRIPTION

For `strlen()` : [CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `strlen()` function shall compute the number of bytes in the string to which `s` points, not including the terminating NUL character.

[CX] The `strnlen()` function shall compute the smaller of the number of bytes in the array to which `s` points, not including any terminating NUL character, or the value of the `maxlen` argument. The `strnlen()` function shall never examine more than `maxlen` bytes of the array pointed to by `s`.

[CX] The `strlen()` and `strnlen()` functions shall not change the setting of `errno` on valid input.

### RETURN VALUE

The `strlen()` function shall return the length of `s`; no return value shall be reserved to indicate an error.

[CX] The `strnlen()` function shall return the number of bytes preceding the first null byte in the array to which `s` points, if `s` contains a null byte within the first `maxlen` bytes; otherwise, it shall return `maxlen`.

### ERRORS

No errors are defined.

# EXAMPLES

## Getting String Lengths

The following example sets the maximum length of `key` and `data` by using `strlen()` to get the lengths of those strings.

```
#include <string.h>
...
struct element {
    char *key;
    char *data;
};
...
char *key, *data;
int len;

*keylength = *datalength = 0;
...
if ((len = strlen(key)) > *keylength)
    *keylength = len;
if ((len = strlen(data)) > *datalength)
    *datalength = len;
...
```

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `strlcat()`
- `wcslen()`

## CHANGE HISTORY

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

### Issue 5

The RETURN VALUE section is updated to indicate that `strlen()` returns the length of `s`, and not `s` itself as was previously stated.

### Issue 7

The `strnlen()` function is added from The Open Group Technical Standard, 2006, Extended API Set Part 1.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0344 [560] is applied.

### Issue 8

Austin Group Defect 448 is applied, adding a requirement that `strlen()` and `strnlen()` do not change the setting of `errno` on valid input.

Austin Group Defect 986 is applied, adding `strlcat()` to the SEE ALSO section.

## 1.236. `strncat` — concatenate a string with part of another

---

### SYNOPSIS

```
#include <string.h>

char *strncat(char *restrict s1, const char *restrict s2, size_t n
```



### DESCRIPTION

The `strncat()` function shall append not more than `n` bytes (a NUL character and bytes that follow it are not appended) from the array pointed to by `s2` to the end of the string pointed to by `s1`. The initial byte of `s2` overwrites the NUL character at the end of `s1`. A terminating NUL character is always appended to the result. If copying takes place between objects that overlap, the behavior is undefined.

The `strncat()` function shall not change the setting of `errno` on valid input.

### RETURN VALUE

The `strncat()` function shall return `s1`; no return value shall be reserved to indicate an error.

### ERRORS

No errors are defined.

### EXAMPLES

None.

### APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `strcat()`
- `strlcat()`
- XBD `<string.h>`

## CHANGE HISTORY

### First released in Issue 1.

Derived from Issue 1 of the SVID.

### Issue 6

The `strncat()` prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

### Issue 8

Austin Group Defect 448 is applied, adding a requirement that `strncat()` does not change the setting of `errno` on valid input.

Austin Group Defect 986 is applied, adding `strlcat()` to the SEE ALSO section.

---

## 1.237. strncmp

---

### SYNOPSIS

```
#include <string.h>

int strncmp(const char *s1, const char *s2, size_t n);
```

### DESCRIPTION

[Option Start] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard. [Option End]

The `strncmp()` function shall compare not more than `n` bytes (bytes that follow a NUL character are not compared) from the array pointed to by `s1` to the array pointed to by `s2`.

The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type `unsigned char`) that differ in the strings being compared.

[Option Start] The `strncmp()` function shall not change the setting of `errno` on valid input. [Option End]

### RETURN VALUE

Upon successful completion, `strncmp()` shall return an integer greater than, equal to, or less than 0, if the possibly null-terminated array pointed to by `s1` is greater than, equal to, or less than the possibly null-terminated array pointed to by `s2` respectively.

### ERRORS

No errors are defined.

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `strcmp()`
- XBD `<string.h>`

## CHANGE HISTORY

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

### Issue 6

Extensions beyond the ISO C standard are marked.

### Issue 8

Austin Group Defect 448 is applied, adding a requirement that `strncmp()` does not change the setting of `errno` on valid input.

## 1.238. `strncpy`

---

### SYNOPSIS

```
#include <string.h>

[CX] char *stpcpy(char *restrict s1, const char *restrict s2, size_t n)
char *strncpy(char *restrict s1, const char *restrict s2, size_t n)
```

### DESCRIPTION

For `strncpy()`: [CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The [CX] `stpcpy()` and `strncpy()` functions shall copy not more than `n` bytes (bytes that follow a NUL character are not copied) from the array pointed to by `s2` to the array pointed to by `s1`.

If the array pointed to by `s2` is a string that is shorter than `n` bytes, NUL characters shall be appended to the copy in the array pointed to by `s1`, until `n` bytes in all are written.

If copying takes place between objects that overlap, the behavior is undefined.

[CX] The `strncpy()` and `stpcpy()` functions shall not change the setting of `errno` on valid input.

### RETURN VALUE

[CX] If a NUL character is written to the destination, the `stpcpy()` function shall return the address of the first such NUL character. Otherwise, it shall return `&s1[n]`.

The `strncpy()` function shall return `s1`.

No return values are reserved to indicate an error.

### ERRORS

No errors are defined.

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

Applications must provide the space in *s1* for the *n* bytes to be transferred, as well as ensure that the *s2* and *s1* arrays do not overlap.

Character movement is performed differently in different implementations. Thus, overlapping moves may yield surprises.

If there is no NUL character byte in the first *n* bytes of the array pointed to by *s2*, the result is not null-terminated.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `strcpy()`
- `strlcat()`
- `wcsncpy()`

XBD

## CHANGE HISTORY

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

## **Issue 6**

The `strncpy()` prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

## **Issue 7**

The `stpncpy()` function is added from The Open Group Technical Standard, 2006, Extended API Set Part 1.

## **Issue 8**

Austin Group Defect 448 is applied, adding a requirement that `strncpy()` and `stpncpy()` do not change the setting of `errno` on valid input.

Austin Group Defect 986 is applied, adding `strlcat()` to the SEE ALSO section.

## 1.239. `strupbrk`

---

### SYNOPSIS

```
#include <string.h>

char *strupbrk(const char *s1, const char *s2);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `strupbrk()` function shall locate the first occurrence in the string pointed to by `s1` of any byte from the string pointed to by `s2`.

[CX] The `strupbrk()` function shall not change the setting of `errno` on valid input.

### RETURN VALUE

Upon successful completion, `strupbrk()` shall return a pointer to the byte or a null pointer if no byte from `s2` occurs in `s1`.

### ERRORS

No errors are defined.

---

*The following sections are informative.*

### EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `strchr()`
- `strrchr()`
- XBD `<string.h>`

## CHANGE HISTORY

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

### Issue 8

Austin Group Defect 448 is applied, adding a requirement that `strupbrk()` does not change the setting of `errno` on valid input.

## 1.240. strrchr — string scanning operation

---

### SYNOPSIS

```
#include <string.h>

char *strrchr(const char *s, int c);
```

### DESCRIPTION

The `strrchr()` function shall locate the last occurrence of `c` (converted to a `char`) in the string pointed to by `s`. The terminating NUL character is considered to be part of the string.

The `strrchr()` function shall not change the setting of `errno` on valid input.

### RETURN VALUE

Upon successful completion, `strrchr()` shall return a pointer to the byte or a null pointer if `c` does not occur in the string.

### ERRORS

No errors are defined.

### EXAMPLES

#### Finding the Base Name of a File

The following example uses `strrchr()` to get a pointer to the base name of a file. The `strrchr()` function searches backwards through the name of the file to find the last '/' character in `name`. This pointer (plus one) will point to the base name of the file.

```
#include <string.h>
...
const char *name;
char *basename;
...
```

```
basename = strrchr(name, '/') + 1;
```

```
...
```

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [strchr\(\)](#)
- [<string.h>](#)

## CHANGE HISTORY

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

### Issue 8

Austin Group Defect 448 is applied, adding a requirement that [strrchr\(\)](#) does not change the setting of [errno](#) on valid input.

---

## 1.241. `strspn` — get length of a substring

---

### SYNOPSIS

```
#include <string.h>

size_t strspn(const char *s1, const char *s2);
```

### DESCRIPTION

The `strspn()` function shall compute the length (in bytes) of the maximum initial segment of the string pointed to by `s1` which consists entirely of bytes from the string pointed to by `s2`.

The `strspn()` function shall not change the setting of `errno` on valid input.

### RETURN VALUE

The `strspn()` function shall return the computed length; no return value is reserved to indicate an error.

### ERRORS

No errors are defined.

### APPLICATION USAGE

None.

### RATIONALE

None.

### FUTURE DIRECTIONS

None.

## SEE ALSO

- `strcspn()`
- `<string.h>`

## CHANGE HISTORY

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

**Issue 5:** The RETURN VALUE section is updated to indicate that `strspn()` returns the length of `s`, and not `s` itself as was previously stated.

**Issue 7:** SD5-XSH-ERN-182 is applied.

**Issue 8:** Austin Group Defect 448 is applied, adding a requirement that `strspn()` does not change the setting of `errno` on valid input.

## 1.242. strstr — find a substring

---

### SYNOPSIS

```
#include <string.h>

char *strstr(const char *s1, const char *s2);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `strstr()` function shall locate the first occurrence in the string pointed to by `s1` of the sequence of bytes (excluding the terminating NUL character) in the string pointed to by `s2`.

[CX] The `strstr()` function shall not change the setting of `errno` on valid input.

### RETURN VALUE

Upon successful completion, `strstr()` shall return a pointer to the located string or a null pointer if the string is not found.

If `s2` points to a string with zero length, the function shall return `s1`.

### ERRORS

No errors are defined.

---

### EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `memmem()`
- `strchr()`
- XBD `<string.h>`

## CHANGE HISTORY

First released in Issue 3. Included for alignment with the ANSI C standard.

### Issue 8

Austin Group Defect 448 is applied, adding a requirement that `strstr()` does not change the setting of `errno` on valid input.

Austin Group Defect 1061 is applied, adding `memmem()` to the SEE ALSO section.

## 1.243. strtod, strtod, strtold — convert a string to a double-precision number

---

### SYNOPSIS

```
#include <stdlib.h>

double strtod(const char *restrict nptr, char **restrict endptr);
float strtodf(const char *restrict nptr, char **restrict endptr);
long double strtold(const char *restrict nptr, char **restrict endptr);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

These functions shall convert the initial portion of the string pointed to by `nptr` to `double`, `float`, and `long double` representation, respectively. First, they decompose the input string into three parts:

1. An initial, possibly empty, sequence of white-space bytes
2. A subject sequence interpreted as a floating-point constant or representing infinity or NaN
3. A final string of one or more unrecognized characters, including the terminating NUL character of the input string

Then they shall attempt to convert the subject sequence to a floating-point number, and return the result.

The expected form of the subject sequence is an optional '+' or '-' sign, then one of the following:

- A non-empty sequence of decimal digits optionally containing a radix character; then an optional exponent part consisting of the character 'e' or the character 'E', optionally followed by a '+' or '-' character, and then followed by one or more decimal digits
- A 0x or 0X, then a non-empty sequence of hexadecimal digits optionally containing a radix character; then an optional binary exponent part consisting

of the character 'p' or the character 'P', optionally followed by a '+' or '-' character, and then followed by one or more decimal digits

- One of INF or INFINITY, ignoring case
- One of NAN or NAN(n-char-sequenceopt), ignoring case in the NAN part, where:

```
n-char-sequence:  
    digit  
    nondigit  
    n-char-sequence digit  
    n-char-sequence nondigit
```

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space byte, that is of the expected form. The subject sequence contains no characters if the input string is not of the expected form.

If the subject sequence has the expected form for a floating-point number, the sequence of characters starting with the first digit or the decimal-point character (whichever occurs first) shall be interpreted as a floating constant of the C language, except that the radix character shall be used in place of a period, and that if neither an exponent part nor a radix character appears in a decimal floating-point number, or if a binary exponent part does not appear in a hexadecimal floating-point number, an exponent part of the appropriate type with value zero is assumed to follow the last digit in the string. If the subject sequence begins with a `,`, the sequence shall be interpreted as negated. A character sequence INF or INFINITY shall be interpreted as an infinity, if representable in the return type, else as if it were a floating constant that is too large for the range of the return type. A character sequence NAN or NAN(n-char-sequenceopt) shall be interpreted as a quiet NaN, if supported in the return type, else as if it were a subject sequence part that does not have the expected form; the meaning of the n-char sequences is implementation-defined. A pointer to the final string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

If the subject sequence has the hexadecimal form and `FLT_RADIX` is a power of 2, the value resulting from the conversion is correctly rounded.

[CX] The radix character is defined in the current locale (category `LC_NUMERIC`). In the `POSIX` locale, or in a locale where the radix character is not defined, the radix character shall default to a `('.)'`.

In other than the C [CX] or `POSIX` locale, additional locale-specific subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion shall be performed; the value of `nptr` is stored in the object pointed

to by `endptr`, provided that `endptr` is not a null pointer.

These functions shall not change the setting of `errno` if successful.

Since 0 is returned on error and is also a valid return on success, an application wishing to check for error situations should set `errno` to 0, then call `strtod()`, `strtof()`, or `strtold()`, then check `errno`.

## RETURN VALUE

Upon successful completion, these functions shall return the converted value. If no conversion could be performed, 0 shall be returned, and `errno` may be set to [EINVAL].

If the correct value would cause an overflow and default rounding is in effect,  $\pm\text{HUGE\_VAL}$ ,  $\pm\text{HUGE\_VALF}$ , or  $\pm\text{HUGE\_VALL}$  shall be returned (according to the sign of the value), and `errno` shall be set to [ERANGE].

If the correct value would cause an underflow, a value whose magnitude is no greater than the smallest normalized positive number in the return type shall be returned [CX] and `errno` set to [ERANGE].

## ERRORS

These functions shall fail if:

- **[ERANGE]** The value to be returned would cause overflow and default rounding is in effect [CX] or the value to be returned would cause underflow.

These functions may fail if:

- **[EINVAL]** [CX] No conversion could be performed.

## EXAMPLES

None.

## APPLICATION USAGE

If the subject sequence has the hexadecimal form and `FLT_RADIX` is not a power of 2, and the result is not exactly representable, the result should be one of the two numbers in the appropriate internal format that are adjacent to the hexadecimal floating source value, with the extra stipulation that the error should have a correct sign for the current rounding direction.

If the subject sequence has the decimal form and at most DECIMAL\_DIG (defined in `<math.h>`) significant digits, the result should be correctly rounded. If the subject sequence D has the decimal form and more than DECIMAL\_DIG significant digits, consider the two bounding, adjacent decimal strings L and U, both having DECIMAL\_DIG significant digits, such that the values of L, D, and U satisfy  $L \leq D \leq U$ . The result should be one of the (equal or adjacent) values that would be obtained by correctly rounding L and U according to the current rounding direction, with the extra stipulation that the error with respect to D should have a correct sign for the current rounding direction.

The changes to `strtod()` introduced by the ISO/IEC 9899:1999 standard can alter the behavior of well-formed applications complying with the ISO/IEC 9899:1990 standard and thus earlier versions of this standard. One such example would be:

```
int
what_kind_of_number (char *s)
{
    char *endp;
    double d;
    long l;

    d = strtod(s, &endp);
    if (s != endp && *endp == '\0')
        printf("It's a float with value %g\n", d);
    else
    {
        l = strtol(s, &endp, 0);
        if (s != endp && *endp == '\0')
            printf("It's an integer with value %ld\n", l);
        else
            return 1;
    }
    return 0;
}
```

If the function is called with:

```
what_kind_of_number ("0x10")
```

an ISO/IEC 9899:1990 standard-compliant library will result in the function printing:

```
It's an integer with value 16
```

With the ISO/IEC 9899:1999 standard, the result is:

```
It's a float with value 16
```

The change in behavior is due to the inclusion of floating-point numbers in hexadecimal notation without requiring that either a decimal point or the binary exponent be present.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `fscanf()`
- `isspace()`
- `localeconv()`
- `setlocale()`
- `strtol()`
- XBD [7. Locale](#)
- `<float.h>`
- `<stdlib.h>`

## CHANGE HISTORY

### First released in Issue 1

Derived from Issue 1 of the SVID.

### Issue 5

The DESCRIPTION is updated to indicate that `errno` is not changed if the function is successful.

## Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is added if no conversion could be performed.

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

- The `strtod()` function is updated.
- The `strtof()` and `strtold()` functions are added.
- The DESCRIPTION is extensively revised.

ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/61 is applied, correcting the second paragraph in the RETURN VALUE section. This change clarifies the sign of the return value.

## Issue 7

Austin Group Interpretation 1003.1-2001 #015 is applied.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0610 [302], XSH/TC1-2008/0611 [94], and XSH/TC1-2008/0612 [105] are applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0348 [584] and XSH/TC2-2008/0349 [796] are applied.

## Issue 8

Austin Group Defect 1163 is applied, clarifying the handling of white space in the input string.

Austin Group Defect 1213 is applied, correcting some typographic errors in the APPLICATION USAGE section.

Austin Group Defect 1302 is applied, aligning these functions with the ISO/IEC 9899:2018 standard.

Austin Group Defect 1686 is applied, adding CX shading to some text in the RETURN VALUE section.

---

## 1.244. strtod, strtod, strtold - convert a string to a double-precision number

---

### SYNOPSIS

```
#include <stdlib.h>

double strtod(const char *restrict nptr, char **restrict endptr);
float strtodf(const char *restrict nptr, char **restrict endptr);
long double strtold(const char *restrict nptr, char **restrict endptr);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

These functions shall convert the initial portion of the string pointed to by `nptr` to **double**, **float**, and **long double** representation, respectively. First, they decompose the input string into three parts:

1. An initial, possibly empty, sequence of white-space bytes
2. A subject sequence interpreted as a floating-point constant or representing infinity or NaN
3. A final string of one or more unrecognized characters, including the terminating NUL character of the input string

Then they shall attempt to convert the subject sequence to a floating-point number, and return the result.

The expected form of the subject sequence is an optional '+' or '-' sign, then one of the following:

- A non-empty sequence of decimal digits optionally containing a radix character; then an optional exponent part consisting of the character 'e' or the character 'E', optionally followed by a '+' or '-' character, and then followed by one or more decimal digits
- A 0x or 0X, then a non-empty sequence of hexadecimal digits optionally containing a radix character; then an optional binary exponent part consisting

of the character 'p' or the character 'P', optionally followed by a '+' or '-' character, and then followed by one or more decimal digits

- One of INF or INFINITY, ignoring case
- One of NAN or NAN(*n-char-sequenceopt*), ignoring case in the NAN part, where:

```
n-char-sequence:  
    digit  
    nondigit  
    n-char-sequence digit  
    n-char-sequence nondigit
```

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space byte, that is of the expected form. The subject sequence contains no characters if the input string is not of the expected form.

If the subject sequence has the expected form for a floating-point number, the sequence of characters starting with the first digit or the decimal-point character (whichever occurs first) shall be interpreted as a floating constant of the C language, except that the radix character shall be used in place of a period, and that if neither an exponent part nor a radix character appears in a decimal floating-point number, or if a binary exponent part does not appear in a hexadecimal floating-point number, an exponent part of the appropriate type with value zero is assumed to follow the last digit in the string. If the subject sequence begins with a `,` the sequence shall be interpreted as negated. A character sequence INF or INFINITY shall be interpreted as an infinity, if representable in the return type, else as if it were a floating constant that is too large for the range of the return type. A character sequence NAN or NAN(*n-char-sequenceopt*) shall be interpreted as a quiet NaN, if supported in the return type, else as if it were a subject sequence part that does not have the expected form; the meaning of the *n-char* sequences is implementation-defined. A pointer to the final string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

If the subject sequence has the hexadecimal form and `FLT_RADIX` is a power of 2, the value resulting from the conversion is correctly rounded.

[CX] The radix character is defined in the current locale (category `LC_NUMERIC`). In the `POSIX` locale, or in a locale where the radix character is not defined, the radix character shall default to a `('.)`.

In other than the C [CX] or `POSIX` locale, additional locale-specific subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion shall be performed; the value of `nptr` is stored in the object pointed

to by `endptr`, provided that `endptr` is not a null pointer.

These functions shall not change the setting of `errno` if successful.

Since 0 is returned on error and is also a valid return on success, an application wishing to check for error situations should set `errno` to 0, then call `strtod()`, `strtof()`, or `strtold()`, then check `errno`.

## RETURN VALUE

Upon successful completion, these functions shall return the converted value. If no conversion could be performed, 0 shall be returned, and `errno` may be set to [EINVAL].

If the correct value would cause an overflow and default rounding is in effect,  $\pm\text{HUGE\_VAL}$ ,  $\pm\text{HUGE\_VALF}$ , or  $\pm\text{HUGE\_VALL}$  shall be returned (according to the sign of the value), and `errno` shall be set to [ERANGE].

If the correct value would cause an underflow, a value whose magnitude is no greater than the smallest normalized positive number in the return type shall be returned [CX] and `errno` set to [ERANGE].

## ERRORS

These functions shall fail if:

### [ERANGE]

The value to be returned would cause overflow and default rounding is in effect [CX] or the value to be returned would cause underflow.

These functions may fail if:

### [EINVAL]

[CX] No conversion could be performed.

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

If the subject sequence has the hexadecimal form and FLT\_RADIX is not a power of 2, and the result is not exactly representable, the result should be one of the two numbers in the appropriate internal format that are adjacent to the hexadecimal floating source value, with the extra stipulation that the error should have a correct sign for the current rounding direction.

If the subject sequence has the decimal form and at most DECIMAL\_DIG (defined in `<float.h>`) significant digits, the result should be correctly rounded. If the subject sequence D has the decimal form and more than DECIMAL\_DIG significant digits, consider the two bounding, adjacent decimal strings L and U, both having DECIMAL\_DIG significant digits, such that the values of L, D, and U satisfy  $L \leq D \leq U$ . The result should be one of the (equal or adjacent) values that would be obtained by correctly rounding L and U according to the current rounding direction, with the extra stipulation that the error with respect to D should have a correct sign for the current rounding direction.

The changes to `strtod()` introduced by the ISO/IEC 9899:1999 standard can alter the behavior of well-formed applications complying with the ISO/IEC 9899:1990 standard and thus earlier versions of this standard. One such example would be:

```
int
what_kind_of_number (char *s)
{
    char *endp;
    double d;
    long l;

    d = strtod(s, &endp);
    if (s != endp && *endp == '\0')
        printf("It's a float with value %g\n", d);
    else
    {
        l = strtol(s, &endp, 0);
        if (s != endp && *endp == '\0')
            printf("It's an integer with value %ld\n", l);
        else
            return 1;
    }
    return 0;
}
```

If the function is called with:

```
what_kind_of_number ("0x10")
```

an ISO/IEC 9899:1990 standard-compliant library will result in the function printing:

It's an integer with value 16

With the ISO/IEC 9899:1999 standard, the result is:

It's a float with value 16

The change in behavior is due to the inclusion of floating-point numbers in hexadecimal notation without requiring that either a decimal point or the binary exponent be present.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

`fscanf()` , `isspace()` , `localeconv()` , `setlocale()` , `strtol()`

XBD 7. Locale, `<float.h>` , `<stdlib.h>`

## CHANGE HISTORY

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

### Issue 5

The DESCRIPTION is updated to indicate that `errno` is not changed if the function is successful.

### Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is added if no conversion could be performed.

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

- The `strtod()` function is updated.
- The `strtof()` and `strtold()` functions are added.
- The DESCRIPTION is extensively revised.

ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/61 is applied, correcting the second paragraph in the RETURN VALUE section. This change clarifies the sign of the return value.

## Issue 7

Austin Group Interpretation 1003.1-2001 #015 is applied.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0610 [302], XSH/TC1-2008/0611 [94], and XSH/TC1-2008/0612 [105] are applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0348 [584] and XSH/TC2-2008/0349 [796] are applied.

## Issue 8

Austin Group Defect 1163 is applied, clarifying the handling of white space in the input string.

Austin Group Defect 1213 is applied, correcting some typographic errors in the APPLICATION USAGE section.

Austin Group Defect 1302 is applied, aligning these functions with the ISO/IEC 9899:2018 standard.

Austin Group Defect 1686 is applied, adding CX shading to some text in the RETURN VALUE section.

---

## 1.245. strtoimax, strtoumax — convert string to integer type

---

### SYNOPSIS

```
#include <inttypes.h>

intmax_t strtoimax(const char *restrict nptr,
                    char **restrict endptr,
                    int base);

uintmax_t strtoumax(const char *restrict nptr,
                     char **restrict endptr,
                     int base);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

These functions shall be equivalent to the `strtol()`, `strtoll()`, `strtoul()`, and `strtoull()` functions, except that the initial portion of the string shall be converted to `intmax_t` and `uintmax_t` representation, respectively.

### RETURN VALUE

These functions shall return the converted value, if any.

If no conversion could be performed, zero shall be returned [CX] and `errno` may be set to [EINVAL].

[CX] If the value of `base` is not supported, 0 shall be returned and `errno` shall be set to [EINVAL].

If the correct value is outside the range of representable values, `{INTMAX_MAX}`, `{INTMAX_MIN}`, or `{UINTMAX_MAX}` shall be returned (according to the return type and sign of the value, if any), and `errno` shall be set to [ERANGE].

# ERRORS

These functions shall fail if:

- **[EINVAL]** [CX] The value of `base` is not supported.
- **[ERANGE]** The value to be returned is not representable.

These functions may fail if:

- **[EINVAL]** No conversion could be performed.
- 

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

Since the value of `*endptr` is unspecified if the value of `base` is not supported, applications should either ensure that `base` has a supported value (0 or between 2 and 36) before the call, or check for an [EINVAL] error before examining `*endptr`.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `strtol()`
- `strtoul()`
- XBD `<inttypes.h>`

# CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

## Issue 7

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0613 [453] and XSH/TC1-2008/0614 [453] are applied.

---

## 1.246. strtok, strtok\_r — split string into tokens

### SYNOPSIS

```
#include <string.h>

char *strtok(char *restrict s, const char *restrict sep);

[X] char *strtok_r(char *restrict s, const char *restrict sep,
                    char **restrict state);
```

### DESCRIPTION

For `strtok()`:

[X] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

A sequence of calls to `strtok()` breaks the string pointed to by `s` into a sequence of tokens, each of which is delimited by a byte from the string pointed to by `sep`. The first call in the sequence has `s` as its first argument, and is followed by calls with a null pointer as their first argument. The separator string pointed to by `sep` may be different from call to call.

The first call in the sequence searches the string pointed to by `s` for the first byte that is **not** contained in the current separator string pointed to by `sep`. If no such byte is found, then there are no tokens in the string pointed to by `s` and `strtok()` shall return a null pointer. If such a byte is found, it is the start of the first token.

The `strtok()` function then searches from there for a byte that **is** contained in the current separator string. If no such byte is found, the current token extends to the end of the string pointed to by `s`, and subsequent searches for a token shall return a null pointer. If such a byte is found, it is overwritten by a NUL character, which terminates the current token. The `strtok()` function saves a pointer to the following byte, from which the next search for a token shall start.

Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

The implementation shall behave as if no function defined in this volume of POSIX.1-2024 calls `strtok()`.

The `strtok()` function need not be thread-safe; however, `strtok()` shall avoid data races with all other functions.

[X] The `strtok_r()` function shall be equivalent to `strtok()`, except that `strtok_r()` shall be thread-safe and the argument `state` points to a user-provided pointer that allows `strtok_r()` to maintain state between calls which scan the same string. The application shall ensure that the pointer pointed to by `state` is unique for each string (`s`) being processed concurrently by `strtok_r()` calls. The application need not initialize the pointer pointed to by `state` to any particular value. The implementation shall not update the pointer pointed to by `state` to point (directly or indirectly) to resources, other than within the string `s`, that need to be freed or released by the caller.

[X] The `strtok()` and `strtok_r()` functions shall not change the setting of `errno` on valid input.

## RETURN VALUE

Upon successful completion, `strtok()` shall return a pointer to the first byte of a token. Otherwise, if there is no token, `strtok()` shall return a null pointer.

[X] The `strtok_r()` function shall return a pointer to the token found, or a null pointer when no token is found.

## ERRORS

No errors are defined.

## EXAMPLES

### Searching for Word Separators

The following example searches for tokens separated by space characters.

```
#include <string.h>
...
char *token;
char line[] = "LINE TO BE SEPARATED";
char *search = " ";

/* Token will point to "LINE". */
token = strtok(line, search);
```

```
/* Token will point to "T0". */
token = strtok(NULL, search);
```

## Find First two Fields in a Buffer

The following example uses `strtok()` to find two character strings (a key and data associated with that key) separated by any combination of space, tab, or newline characters at the start of the array of characters pointed to by `buffer`.

```
#include <string.h>
...
char *buffer;
...
struct element {
    char *key;
    char *data;
} e;
...
// Load the buffer...
...
// Get the key and its data...
e.key = strtok(buffer, " \t\n");
e.data = strtok(NULL, " \t\n");
// Process the rest of the contents of the buffer...
...
```

## APPLICATION USAGE

Note that if `sep` is the empty string, `strtok()` and `strtok_r()` return a pointer to the remainder of the string being tokenized.

The `strtok_r()` function is thread-safe and stores its state in a user-supplied buffer instead of possibly using a static data area that may be overwritten by an unrelated call from another thread.

## RATIONALE

The `strtok()` function searches for a separator string within a larger string. It returns a pointer to the last substring between separator strings. This function uses static storage to keep track of the current string position between calls. The new function, `strtok_r()`, takes an additional argument, `state`, to keep track of the current position in the string.

# FUTURE DIRECTIONS

None.

## SEE ALSO

XBD `<string.h>`

## CHANGE HISTORY

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

### Issue 5

The `strtok_r()` function is included for alignment with the POSIX Threads Extension.

A note indicating that the `strtok()` function need not be reentrant is added to the DESCRIPTION.

### Issue 6

Extensions beyond the ISO C standard are marked.

The `strtok_r()` function is marked as part of the Thread-Safe Functions option.

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

The `restrict` keyword is added to the `strtok()` and `strtok_r()` prototypes for alignment with the ISO/IEC 9899:1999 standard.

### Issue 7

Austin Group Interpretation 1003.1-2001 #156 is applied.

SD5-XSH-ERN-235 is applied, correcting an example.

The `strtok_r()` function is moved from the Thread-Safe Functions option to the Base.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0615 [177] is applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0350 [878] is applied.

## Issue 8

Austin Group Defect 448 is applied, adding a requirement that `strtok()` and `strtok_r()` do not change the setting of `errno` on valid input.

Austin Group Defect 1302 is applied, aligning the `strtok()` function with the ISO/IEC 9899:2018 standard.

## 1.247. strtok, strtok\_r — split string into tokens

---

### SYNOPSIS

```
#include <string.h>

char *strtok(char *restrict s, const char *restrict sep);

[CX] char *strtok_r(char *restrict s, const char *restrict sep,
                     char **restrict state);
```

### DESCRIPTION

For `strtok()` : [CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

A sequence of calls to `strtok()` breaks the string pointed to by `s` into a sequence of tokens, each of which is delimited by a byte from the string pointed to by `sep`. The first call in the sequence has `s` as its first argument, and is followed by calls with a null pointer as their first argument. The separator string pointed to by `sep` may be different from call to call.

The first call in the sequence searches the string pointed to by `s` for the first byte that is **not** contained in the current separator string pointed to by `sep`. If no such byte is found, then there are no tokens in the string pointed to by `s` and `strtok()` shall return a null pointer. If such a byte is found, it is the start of the first token.

The `strtok()` function then searches from there for a byte that **is** contained in the current separator string. If no such byte is found, the current token extends to the end of the string pointed to by `s`, and subsequent searches for a token shall return a null pointer. If such a byte is found, it is overwritten by a NUL character, which terminates the current token. The `strtok()` function saves a pointer to the following byte, from which the next search for a token shall start.

Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

The implementation shall behave as if no function defined in this volume of POSIX.1-2024 calls `strtok()`.

The `strtok()` function need not be thread-safe; however, `strtok()` shall avoid data races with all other functions.

[CX] The `strtok_r()` function shall be equivalent to `strtok()`, except that `strtok_r()` shall be thread-safe and the argument `state` points to a user-provided pointer that allows `strtok_r()` to maintain state between calls which scan the same string. The application shall ensure that the pointer pointed to by `state` is unique for each string (`s`) being processed concurrently by `strtok_r()` calls. The application need not initialize the pointer pointed to by `state` to any particular value. The implementation shall not update the pointer pointed to by `state` to point (directly or indirectly) to resources, other than within the string `s`, that need to be freed or released by the caller.

[CX] The `strtok()` and `strtok_r()` functions shall not change the setting of `errno` on valid input.

## RETURN VALUE

Upon successful completion, `strtok()` shall return a pointer to the first byte of a token. Otherwise, if there is no token, `strtok()` shall return a null pointer.

[CX] The `strtok_r()` function shall return a pointer to the token found, or a null pointer when no token is found.

## ERRORS

No errors are defined.

## EXAMPLES

### Searching for Word Separators

The following example searches for tokens separated by space characters.

```
#include <string.h>
...
char *token;
char line[] = "LINE TO BE SEPARATED";
char *search = " ";

/* Token will point to "LINE". */
token = strtok(line, search);
```

```
/* Token will point to "T0". */
token = strtok(NULL, search);
```

## Find First two Fields in a Buffer

The following example uses `strtok()` to find two character strings (a key and data associated with that key) separated by any combination of space, tab, or newline characters at the start of the array of characters pointed to by `buffer`.

```
#include <string.h>
...
char    *buffer;
...
struct element {
    char *key;
    char *data;
} e;
...
// Load the buffer...
...
// Get the key and its data...
e.key = strtok(buffer, " \t\n");
e.data = strtok(NULL, " \t\n");
// Process the rest of the contents of the buffer...
...
```

## APPLICATION USAGE

Note that if `sep` is the empty string, `strtok()` and `strtok_r()` return a pointer to the remainder of the string being tokenized.

The `strtok_r()` function is thread-safe and stores its state in a user-supplied buffer instead of possibly using a static data area that may be overwritten by an unrelated call from another thread.

## RATIONALE

The `strtok()` function searches for a separator string within a larger string. It returns a pointer to the last substring between separator strings. This function uses static storage to keep track of the current string position between calls. The new function, `strtok_r()`, takes an additional argument, `state`, to keep track of the current position in the string.

# FUTURE DIRECTIONS

None.

## SEE ALSO

XBD `<string.h>`

## CHANGE HISTORY

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

### Issue 5

The `strtok_r()` function is included for alignment with the POSIX Threads Extension.

A note indicating that the `strtok()` function need not be reentrant is added to the DESCRIPTION.

### Issue 6

Extensions beyond the ISO C standard are marked.

The `strtok_r()` function is marked as part of the Thread-Safe Functions option.

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

The `restrict` keyword is added to the `strtok()` and `strtok_r()` prototypes for alignment with the ISO/IEC 9899:1999 standard.

### Issue 7

Austin Group Interpretation 1003.1-2001 #156 is applied.

SD5-XSH-ERN-235 is applied, correcting an example.

The `strtok_r()` function is moved from the Thread-Safe Functions option to the Base.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0615 [177] is applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0350 [878] is applied.

## Issue 8

Austin Group Defect 448 is applied, adding a requirement that `strtok()` and `strtok_r()` do not change the setting of `errno` on valid input.

Austin Group Defect 1302 is applied, aligning the `strtok()` function with the ISO/IEC 9899:2018 standard.

## 1.248. strtol, strtoll — convert a string to a long integer

### SYNOPSIS

```
#include <stdlib.h>

long strtol(const char *restrict nptr, char **restrict endptr, int
long long strtoll(const char *restrict nptr, char **restrict endpt
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

These functions shall convert the initial portion of the string pointed to by `nptr` to a type `long` and `long long` representation, respectively. First, they decompose the input string into three parts:

1. An initial, possibly empty, sequence of white-space bytes
2. A subject sequence interpreted as an integer represented in some radix determined by the value of `base`
3. A final string of one or more unrecognized characters, including the terminating NUL character of the input string.

Then they shall attempt to convert the subject sequence to an integer, and return the result.

If the value of `base` is 0, the expected form of the subject sequence is that of a decimal constant, octal constant, or hexadecimal constant, any of which may be preceded by a '+' or '-' sign. A decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal constant consists of the prefix '0' optionally followed by a sequence of the digits '0' to '7' only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the decimal digits and letters 'a' (or 'A') to 'f' (or 'F') with values 10 to 15 respectively.

If the value of `base` is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by `base`, optionally preceded by a '+' or '-' sign. The letters from 'a' (or 'A') to 'z' (or 'Z') inclusive are ascribed the values 10 to 35; only letters whose

ascribed values are less than that of `base` are permitted. If the value of `base` is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space byte, that is of the expected form. The subject sequence shall contain no characters if the input string is empty or consists entirely of white-space bytes, or if the first non-white-space byte is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of `base` is 0, the sequence of characters starting with the first digit shall be interpreted as an integer constant. If the subject sequence has the expected form and the value of `base` is between 2 and 36, it shall be used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a `,` the resulting value shall be the negative of the converted value. A pointer to the final string shall be stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

In other than the C [CX] or POSIX locale, additional locale-specific subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of `nptr` shall be stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

These functions shall not change the setting of `errno` if successful.

Since 0, {LONG\_MIN} or {LLONG\_MIN}, and {LONG\_MAX} or {LLONG\_MAX} are returned on error and are also valid returns on success, an application wishing to check for error situations should set `errno` to 0, then call `strtol()` or `strtoll()`, then check `errno`.

## RETURN VALUE

Upon successful completion, these functions shall return the converted value, if any. If no conversion could be performed, 0 shall be returned [CX] and `errno` may be set to [EINVAL].

[CX] If the value of `base` is not supported, 0 shall be returned and `errno` shall be set to [EINVAL].

If the correct value is outside the range of representable values, {LONG\_MIN}, {LONG\_MAX}, {LLONG\_MIN}, or {LLONG\_MAX} shall be returned (according to the sign of the value), and `errno` set to [ERANGE].

# ERRORS

These functions shall fail if:

- **[EINVAL]** [CX] The value of `base` is not supported.
- **[ERANGE]** The value to be returned is not representable.

These functions may fail if:

- **[EINVAL]** No conversion could be performed.

## EXAMPLES

None.

## APPLICATION USAGE

Since the value of `*endptr` is unspecified if the value of `base` is not supported, applications should either ensure that `base` has a supported value (0 or between 2 and 36) before the call, or check for an [EINVAL] error before examining `*endptr`.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `fscanf()`
- `isalpha()`
- `strtod()`
- XBD `<stdlib.h>`

# CHANGE HISTORY

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

## Issue 5

The DESCRIPTION is updated to indicate that `errno` is not changed if the function is successful.

## Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is added if no conversion could be performed.

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

- The `strtol()` prototype is updated.
- The `strtoll()` function is added.

## Issue 7

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0616 [453], XSH/TC1-2008/0617 [105], XSH/TC1-2008/0618 [453], XSH/TC1-2008/0619 [453], and XSH/TC1-2008/0620 [453] are applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0351 [892], XSH/TC2-2008/0352 [584], XSH/TC2-2008/0353 [796], and XSH/TC2-2008/0354 [892] are applied.

## Issue 8

Austin Group Defect 700 is applied, clarifying how a subject sequence beginning with is converted.

Austin Group Defect 1163 is applied, clarifying the handling of white space in the input string.

---

## 1.249. strtold

### SYNOPSIS

```
#include <stdlib.h>

double strtod(const char *restrict nptr, char **restrict endptr);
float strtof(const char *restrict nptr, char **restrict endptr);
long double strtold(const char *restrict nptr, char **restrict endptr);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

These functions shall convert the initial portion of the string pointed to by `nptr` to **double**, **float**, and **long double** representation, respectively. First, they decompose the input string into three parts:

1. An initial, possibly empty, sequence of white-space bytes
2. A subject sequence interpreted as a floating-point constant or representing infinity or NaN
3. A final string of one or more unrecognized characters, including the terminating NUL character of the input string

Then they shall attempt to convert the subject sequence to a floating-point number, and return the result.

The expected form of the subject sequence is an optional '+' or '-' sign, then one of the following:

- A non-empty sequence of decimal digits optionally containing a radix character; then an optional exponent part consisting of the character 'e' or the character 'E', optionally followed by a '+' or '-' character, and then followed by one or more decimal digits
- A 0x or 0X, then a non-empty sequence of hexadecimal digits optionally containing a radix character; then an optional binary exponent part consisting of the character 'p' or the character 'P', optionally followed by a '+' or '-' character, and then followed by one or more decimal digits

- One of INF or INFINITY, ignoring case
- One of NAN or NAN(*n-char-sequence* *opt*), ignoring case in the NAN part, where:

```

n-char-sequence:
    digit
    nondigit
    n-char-sequence digit
    n-char-sequence nondigit

```

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space byte, that is of the expected form. The subject sequence contains no characters if the input string is not of the expected form.

If the subject sequence has the expected form for a floating-point number, the sequence of characters starting with the first digit or the decimal-point character (whichever occurs first) shall be interpreted as a floating constant of the C language, except that the radix character shall be used in place of a period, and that if neither an exponent part nor a radix character appears in a decimal floating-point number, or if a binary exponent part does not appear in a hexadecimal floating-point number, an exponent part of the appropriate type with value zero is assumed to follow the last digit in the string. If the subject sequence begins with a `,`, the sequence shall be interpreted as negated. A character sequence INF or INFINITY shall be interpreted as an infinity, if representable in the return type, else as if it were a floating constant that is too large for the range of the return type. A character sequence NAN or NAN(*n-char-sequence* *opt*) shall be interpreted as a quiet NaN, if supported in the return type, else as if it were a subject sequence part that does not have the expected form; the meaning of the *n*-char sequences is implementation-defined. A pointer to the final string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

If the subject sequence has the hexadecimal form and `FLT_RADIX` is a power of 2, the value resulting from the conversion is correctly rounded.

[CX] The radix character is defined in the current locale (category `LC_NUMERIC`). In the `POSIX` locale, or in a locale where the radix character is not defined, the radix character shall default to a `('.)`.

In other than the C [CX] or `POSIX` locale, additional locale-specific subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion shall be performed; the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

These functions shall not change the setting of `errno` if successful.

Since 0 is returned on error and is also a valid return on success, an application wishing to check for error situations should set `errno` to 0, then call `strtod()`, `strtof()`, or `strtold()`, then check `errno`.

## RETURN VALUE

Upon successful completion, these functions shall return the converted value. If no conversion could be performed, 0 shall be returned, and `errno` may be set to [EINVAL].

If the correct value would cause an overflow and default rounding is in effect,  $\pm\text{HUGE\_VAL}$ ,  $\pm\text{HUGE\_VALF}$ , or  $\pm\text{HUGE\_VALL}$  shall be returned (according to the sign of the value), and `errno` shall be set to [ERANGE].

If the correct value would cause an underflow, a value whose magnitude is no greater than the smallest normalized positive number in the return type shall be returned [CX] and `errno` set to [ERANGE].

## ERRORS

These functions shall fail if:

- [ERANGE]: The value to be returned would cause overflow and default rounding is in effect [CX] or the value to be returned would cause underflow.

These functions may fail if:

- [EINVAL]: [CX] No conversion could be performed.

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

If the subject sequence has the hexadecimal form and `FLT_RADIX` is not a power of 2, and the result is not exactly representable, the result should be one of the two numbers in the appropriate internal format that are adjacent to the hexadecimal

floating source value, with the extra stipulation that the error should have a correct sign for the current rounding direction.

If the subject sequence has the decimal form and at most DECIMAL\_DIG (defined in `<float.h>`) significant digits, the result should be correctly rounded. If the subject sequence D has the decimal form and more than DECIMAL\_DIG significant digits, consider the two bounding, adjacent decimal strings L and U, both having DECIMAL\_DIG significant digits, such that the values of L, D, and U satisfy  $L \leq D \leq U$ . The result should be one of the (equal or adjacent) values that would be obtained by correctly rounding L and U according to the current rounding direction, with the extra stipulation that the error with respect to D should have a correct sign for the current rounding direction.

The changes to `strtod()` introduced by the ISO/IEC 9899:1999 standard can alter the behavior of well-formed applications complying with the ISO/IEC 9899:1990 standard and thus earlier versions of this standard. One such example would be:

```
int
what_kind_of_number (char *s)
{
    char *endp;
    double d;
    long l;

    d = strtod(s, &endp);
    if (s != endp && *endp == '\0')
        printf("It's a float with value %g\n", d);
    else
    {
        l = strtol(s, &endp, 0);
        if (s != endp && *endp == '\0')
            printf("It's an integer with value %ld\n", l);
        else
            return 1;
    }
    return 0;
}
```

If the function is called with:

```
what_kind_of_number ("0x10")
```

an ISO/IEC 9899:1990 standard-compliant library will result in the function printing:

```
It's an integer with value 16
```

With the ISO/IEC 9899:1999 standard, the result is:

It's a float with value 16

The change in behavior is due to the inclusion of floating-point numbers in hexadecimal notation without requiring that either a decimal point or the binary exponent be present.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [fscanf\(\)](#)
- [isspace\(\)](#)
- [localeconv\(\)](#)
- [setlocale\(\)](#)
- [strtol\(\)](#)

XBD 7. Locale, [`<float.h>`](#), [`<stdlib.h>`](#)

## CHANGE HISTORY

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

### Issue 5

The DESCRIPTION is updated to indicate that `errno` is not changed if the function is successful.

### Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is added if no conversion could be performed.

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

- The `strtod()` function is updated.
- The `strtof()` and `strtold()` functions are added.
- The DESCRIPTION is extensively revised.

ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/61 is applied, correcting the second paragraph in the RETURN VALUE section. This change clarifies the sign of the return value.

## Issue 7

Austin Group Interpretation 1003.1-2001 #015 is applied.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0610 [302], XSH/TC1-2008/0611 [94], and XSH/TC1-2008/0612 [105] are applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0348 [584] and XSH/TC2-2008/0349 [796] are applied.

## Issue 8

Austin Group Defect 1163 is applied, clarifying the handling of white space in the input string.

Austin Group Defect 1213 is applied, correcting some typographic errors in the APPLICATION USAGE section.

Austin Group Defect 1302 is applied, aligning these functions with the ISO/IEC 9899:2018 standard.

Austin Group Defect 1686 is applied, adding CX shading to some text in the RETURN VALUE section.

---

## 1.250. strtoll

### SYNOPSIS

```
#include <stdlib.h>

long strtol(const char *restrict nptr, char **restrict endptr, int
long long strtoll(const char *restrict nptr, char **restrict endpt
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

These functions shall convert the initial portion of the string pointed to by `nptr` to a type `long` and `long long` representation, respectively. First, they decompose the input string into three parts:

1. An initial, possibly empty, sequence of white-space bytes
2. A subject sequence interpreted as an integer represented in some radix determined by the value of `base`
3. A final string of one or more unrecognized characters, including the terminating NUL character of the input string.

Then they shall attempt to convert the subject sequence to an integer, and return the result.

If the value of `base` is 0, the expected form of the subject sequence is that of a decimal constant, octal constant, or hexadecimal constant, any of which may be preceded by a '+' or '-' sign. A decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal constant consists of the prefix '0' optionally followed by a sequence of the digits '0' to '7' only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the decimal digits and letters 'a' (or 'A') to 'f' (or 'F') with values 10 to 15 respectively.

If the value of `base` is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by `base`, optionally preceded by a '+' or '-' sign. The letters from 'a' (or 'A') to 'z' (or 'Z') inclusive are ascribed the values 10 to 35; only letters whose

ascribed values are less than that of `base` are permitted. If the value of `base` is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space byte, that is of the expected form. The subject sequence shall contain no characters if the input string is empty or consists entirely of white-space bytes, or if the first non-white-space byte is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of `base` is 0, the sequence of characters starting with the first digit shall be interpreted as an integer constant. If the subject sequence has the expected form and the value of `base` is between 2 and 36, it shall be used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a `,`, the resulting value shall be the negative of the converted value. A pointer to the final string shall be stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

In other than the C [CX] or POSIX locale, additional locale-specific subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of `nptr` shall be stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

These functions shall not change the setting of `errno` if successful.

Since 0, {LONG\_MIN} or {LLONG\_MIN}, and {LONG\_MAX} or {LLONG\_MAX} are returned on error and are also valid returns on success, an application wishing to check for error situations should set `errno` to 0, then call `strtol()` or `strtoll()`, then check `errno`.

## RETURN VALUE

Upon successful completion, these functions shall return the converted value, if any. If no conversion could be performed, 0 shall be returned [CX] and `errno` may be set to [EINVAL].

[CX] If the value of `base` is not supported, 0 shall be returned and `errno` shall be set to [EINVAL].

If the correct value is outside the range of representable values, {LONG\_MIN}, {LONG\_MAX}, {LLONG\_MIN}, or {LLONG\_MAX} shall be returned (according to the sign of the value), and `errno` set to [ERANGE].

# ERRORS

These functions shall fail if:

- **[EINVAL]** [CX] The value of `base` is not supported.
- **[ERANGE]** The value to be returned is not representable.

These functions may fail if:

- **[EINVAL]** No conversion could be performed.

# EXAMPLES

None.

# APPLICATION USAGE

Since the value of `*endptr` is unspecified if the value of `base` is not supported, applications should either ensure that `base` has a supported value (0 or between 2 and 36) before the call, or check for an [EINVAL] error before examining `*endptr`.

# RATIONALE

None.

# FUTURE DIRECTIONS

None.

# SEE ALSO

- [fscanf\(\)](#)
- [isalpha\(\)](#)
- [strtod\(\)](#)
- XBD

# CHANGE HISTORY

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

## Issue 5

The DESCRIPTION is updated to indicate that `errno` is not changed if the function is successful.

## Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is added if no conversion could be performed.

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

- The `strtol()` prototype is updated.
- The `strtoll()` function is added.

## Issue 7

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0616 [453], XSH/TC1-2008/0617 [105], XSH/TC1-2008/0618 [453], XSH/TC1-2008/0619 [453], and XSH/TC1-2008/0620 [453] are applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0351 [892], XSH/TC2-2008/0352 [584], XSH/TC2-2008/0353 [796], and XSH/TC2-2008/0354 [892] are applied.

## Issue 8

Austin Group Defect 700 is applied, clarifying how a subject sequence beginning with is converted.

Austin Group Defect 1163 is applied, clarifying the handling of white space in the input string.

---

## 1.251. strtoul

---

### SYNOPSIS

```
#include <stdlib.h>

unsigned long strtoul(const char *restrict str,
                      char **restrict endptr, int base);
unsigned long long strtoull(const char *restrict str,
                           char **restrict endptr, int base);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

These functions shall convert the initial portion of the string pointed to by `str` to a type **unsigned long** and **unsigned long long** representation, respectively. First, they decompose the input string into three parts:

1. An initial, possibly empty, sequence of white-space bytes
2. A subject sequence interpreted as an integer represented in some radix determined by the value of `base`
3. A final string of one or more unrecognized characters, including the terminating NUL character of the input string

Then they shall attempt to convert the subject sequence to an unsigned integer, and return the result.

If the value of `base` is 0, the expected form of the subject sequence is that of a decimal constant, octal constant, or hexadecimal constant, any of which may be preceded by a '+' or '-' sign. A decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal constant consists of the prefix '0' optionally followed by a sequence of the digits '0' to '7' only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the decimal digits and letters 'a' (or 'A') to 'f' (or 'F') with values 10 to 15 respectively.

If the value of `base` is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by `base`, optionally preceded by a '+' or '-' sign. The letters from 'a' (or

'A') to 'z' (or 'Z') inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less than that of `base` are permitted. If the value of `base` is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space byte, that is of the expected form. The subject sequence shall contain no characters if the input string is empty or consists entirely of white-space bytes, or if the first non-white-space byte is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of `base` is 0, the sequence of characters starting with the first digit shall be interpreted as an integer constant. If the subject sequence has the expected form and the value of `base` is between 2 and 36, it shall be used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a `,` the resulting value shall be the negative of the converted value; this action shall be performed in the return type. A pointer to the final string shall be stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

In other than the C [CX] or POSIX locale, additional locale-specific subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion shall be performed; the value of `str` shall be stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

These functions shall not change the setting of `errno` if successful.

Since 0, `{ULONG_MAX}`, and `{ULLONG_MAX}` are returned on error and are also valid returns on success, an application wishing to check for error situations should set `errno` to 0, then call `strtoul()` or `strtoull()`, then check `errno`.

## RETURN VALUE

Upon successful completion, these functions shall return the converted value, if any. If no conversion could be performed, 0 shall be returned [CX] and `errno` may be set to [EINVAL].

[CX] If the value of `base` is not supported, 0 shall be returned and `errno` shall be set to [EINVAL].

If the correct value is outside the range of representable values, `{ULONG_MAX}` or `{ULLONG_MAX}` shall be returned and `errno` set to [ERANGE].

# ERRORS

These functions shall fail if:

- **[EINVAL]**: [CX] The value of `base` is not supported.
- **[ERANGE]**: The value to be returned is not representable.

These functions may fail if:

- **[EINVAL]**: [CX] No conversion could be performed.
- 

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

Since the value of `*endptr` is unspecified if the value of `base` is not supported, applications should either ensure that `base` has a supported value (0 or between 2 and 36) before the call, or check for an [EINVAL] error before examining `*endptr`.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `fscanf()`
- `isalpha()`
- `strtod()`

- `strtol()`

XBD `<stdlib.h>`

## CHANGE HISTORY

### First released in Issue 4

Derived from the ANSI C standard.

### Issue 5

The DESCRIPTION is updated to indicate that `errno` is not changed if the function is successful.

### Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EINVAL] error condition is added for when the value of `base` is not supported.
- In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is added if no conversion could be performed.

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

- The `strtoul()` prototype is updated.
- The `strtoull()` function is added.

### Issue 7

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0621 [105], XSH/TC1-2008/0622 [453], and XSH/TC1-2008/0623 [453] are applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0355 [584] and XSH/TC2-2008/0356 [796] are applied.

## Issue 8

Austin Group Defect 700 is applied, clarifying how a subject sequence beginning with is converted.

Austin Group Defect 1163 is applied, clarifying the handling of white space in the input string.

---

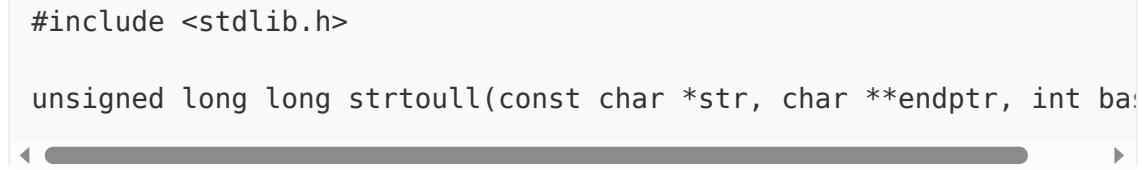
## 1.252. strtoull — convert a string to an unsigned long long integer

---

### SYNOPSIS

```
#include <stdlib.h>

unsigned long long strtoull(const char *str, char **endptr, int bas
```

A code block containing the C header and function declaration. It includes a horizontal scroll bar at the bottom.

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

These functions shall convert the initial portion of the string pointed to by `str` to a type **unsigned long** and **unsigned long long** representation, respectively. First, they decompose the input string into three parts:

1. An initial, possibly empty, sequence of white-space bytes
2. A subject sequence interpreted as an integer represented in some radix determined by the value of `base`
3. A final string of one or more unrecognized characters, including the terminating NUL character of the input string

Then they shall attempt to convert the subject sequence to an unsigned integer, and return the result.

If the value of `base` is 0, the expected form of the subject sequence is that of a decimal constant, octal constant, or hexadecimal constant, any of which may be preceded by a '+' or '-' sign. A decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal constant consists of the prefix '0' optionally followed by a sequence of the digits '0' to '7' only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the decimal digits and letters 'a' (or 'A') to 'f' (or 'F') with values 10 to 15 respectively.

If the value of `base` is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by `base`, optionally preceded by a '+' or '-' sign. The letters from 'a' (or 'A') to 'z' (or 'Z') inclusive are ascribed the values 10 to 35; only letters whose

ascribed values are less than that of `base` are permitted. If the value of `base` is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space byte, that is of the expected form. The subject sequence shall contain no characters if the input string is empty or consists entirely of white-space bytes, or if the first non-white-space byte is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of `base` is 0, the sequence of characters starting with the first digit shall be interpreted as an integer constant. If the subject sequence has the expected form and the value of `base` is between 2 and 36, it shall be used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a `,`, the resulting value shall be the negative of the converted value; this action shall be performed in the return type. A pointer to the final string shall be stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

In other than the C [CX] or POSIX locale, additional locale-specific subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion shall be performed; the value of `str` shall be stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

These functions shall not change the setting of `errno` if successful.

Since 0, `{ULONG_MAX}`, and `{ULLONG_MAX}` are returned on error and are also valid returns on success, an application wishing to check for error situations should set `errno` to 0, then call `strtoul()` or `strtoull()`, then check `errno`.

## RETURN VALUE

Upon successful completion, these functions shall return the converted value, if any. If no conversion could be performed, 0 shall be returned [CX] and `errno` may be set to [EINVAL].

[CX] If the value of `base` is not supported, 0 shall be returned and `errno` shall be set to [EINVAL].

If the correct value is outside the range of representable values, `{ULONG_MAX}` or `{ULLONG_MAX}` shall be returned and `errno` set to [ERANGE].

# ERRORS

The `strtoull()` function shall fail if:

- **[EINVAL]** — The value of `base` is not supported.
  - **[ERANGE]** — The correct value is outside the range of representable values.
- 

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

The `strtoull()` function is useful for converting string representations of unsigned integers to their numeric values, with proper error checking and handling of various number bases.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `strtoul()`
- `strtoimax()`
- `strtoumax()`
- `<stdlib.h>`

# CHANGE HISTORY

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

## Issue 8

Austin Group Defect 1541 is applied, aligning the functionality with the ISO C standard.

---

## 1.253. strtoumax

---

### SYNOPSIS

```
#include <inttypes.h>

intmax_t strtoimax(const char *restrict nptr,
                    char **restrict endptr,
                    int base);
uintmax_t strtoumax(const char *restrict nptr,
                     char **restrict endptr,
                     int base);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

These functions shall be equivalent to the `strtol()`, `strtoll()`, `strtoul()`, and `strtoull()` functions, except that the initial portion of the string shall be converted to `intmax_t` and `uintmax_t` representation, respectively.

### RETURN VALUE

These functions shall return the converted value, if any.

If no conversion could be performed, zero shall be returned [CX] and `errno` may be set to [EINVAL].

[CX] If the value of `base` is not supported, 0 shall be returned and `errno` shall be set to [EINVAL].

If the correct value is outside the range of representable values, `{INTMAX_MAX}`, `{INTMAX_MIN}`, or `{UINTMAX_MAX}` shall be returned (according to the return type and sign of the value, if any), and `errno` shall be set to [ERANGE].

### ERRORS

These functions shall fail if:

- **[EINVAL]** [CX] The value of `base` is not supported.
- **[ERANGE]** The value to be returned is not representable.

These functions may fail if:

- **[EINVAL]** No conversion could be performed.
- 

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

Since the value of `*endptr` is unspecified if the value of `base` is not supported, applications should either ensure that `base` has a supported value (0 or between 2 and 36) before the call, or check for an [EINVAL] error before examining `*endptr`.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `strtol()`
- `strtoul()`
- XBD `<inttypes.h>`

# CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

## Issue 7

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0613 [453] and XSH/TC1-2008/0614 [453] are applied.

---

## 1.254. `strxfrm`, `strxfrm_l` — string transformation

### SYNOPSIS

```
#include <string.h>

size_t strxfrm(char *restrict s1, const char *restrict s2, size_t n);

/* [CX] Extension */
size_t strxfrm_l(char *restrict s1, const char *restrict s2,
                  size_t n, locale_t locale);
```

### DESCRIPTION

For `strxfrm()`: [CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `strxfrm()` [CX] and `strxfrm_l()` functions shall transform the string pointed to by `s2` and place the resulting string into the array pointed to by `s1`. The transformation is such that if `strcmp()` is applied to two transformed strings, it shall return a value greater than, equal to, or less than 0, corresponding to the result of `strcoll()` [CX] or `strcoll_l()`, respectively, applied to the same two original strings [CX] with the same locale. No more than `n` bytes are placed into the resulting array pointed to by `s1`, including the terminating NUL character. If `n` is 0, `s1` is permitted to be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

[CX] The `strxfrm()` and `strxfrm_l()` functions shall not change the setting of `errno` if successful.

Since no return value is reserved to indicate an error, an application wishing to check for error situations should set `errno` to 0, then call `strxfrm()` [CX] or `strxfrm_l()`, then check `errno`.

[CX] The behavior is undefined if the `locale` argument to `strxfrm_l()` is the special locale object `LC_GLOBAL_LOCALE` or is not a valid locale object handle.

## RETURN VALUE

Upon successful completion, `strxfrm()` [CX] and `strxfrm_l()` shall return the length of the transformed string (not including the terminating NUL character). If the value returned is `n` or more, the contents of the array pointed to by `s1` are unspecified.

On error, `strxfrm()` [CX] and `strxfrm_l()` may set `errno` but no return value is reserved to indicate an error.

## ERRORS

These functions may fail if:

- **[EINVAL]** [CX] The string pointed to by the `s2` argument contains characters outside the domain of the collating sequence.
- 

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

The transformation function is such that two transformed strings can be ordered by `strcmp()` as appropriate to collating sequence information in the current locale (category `LC_COLLATE`).

The fact that when `n` is 0 `s1` is permitted to be a null pointer is useful to determine the size of the `s1` array prior to making the transformation.

## RATIONALE

None.

# FUTURE DIRECTIONS

None.

## SEE ALSO

- `strcmp()`
- `strcoll()`
- `<string.h>`

## CHANGE HISTORY

First released in Issue 3. Included for alignment with the ISO C standard.

### Issue 5

The DESCRIPTION is updated to indicate that `errno` does not change if the function is successful.

### Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is added if no conversion could be performed.

The `strxfrm()` prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

### Issue 7

The `strxfrm_l()` function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0624 [283], XSH/TC1-2008/0625 [283], and XSH/TC1-2008/0626 [302] are applied.

---

## 1.255. sysconf

---

### SYNOPSIS

```
#include <unistd.h>

long sysconf(int name);
```

### DESCRIPTION

The `sysconf()` function provides a method for the application to determine the current value of a configurable system limit or option (variable). The implementation shall support all of the variables listed in the following table and may support others.

The `name` argument represents the system variable to be queried. The following table lists the minimal set of system variables from `<limits.h>` or `<unistd.h>` that can be returned by `sysconf()`, and the symbolic constants defined in `<unistd.h>` that are the corresponding values used for `name`.

### System Variables

Variable	Value of Name
{AIO_LISTIO_MAX}	_SC_AIO_LISTIO_MAX
{AIO_MAX}	_SC_AIO_MAX
{AIO_PRIO_DELTA_MAX}	_SC_AIO_PRIO_DELTA_MAX
{ARG_MAX}	_SC_ARG_MAX
{ATEXIT_MAX}	_SC_ATEXIT_MAX
{BC_BASE_MAX}	_SC_BC_BASE_MAX
{BC_DIM_MAX}	_SC_BC_DIM_MAX
{BC_SCALE_MAX}	_SC_BC_SCALE_MAX

Variable	Value of Name
{BC_STRING_MAX}	_SC_BC_STRING_MAX
{CHILD_MAX}	_SC_CHILD_MAX
Clock ticks/second	_SC_CLK_TCK
{COLL_WEIGHTS_MAX}	_SC_COLL_WEIGHTS_MAX
{DELAYTIMER_MAX}	_SC_DELAYTIMER_MAX
{EXPR_NEST_MAX}	_SC_EXPR_NEST_MAX
{HOST_NAME_MAX}	_SC_HOST_NAME_MAX
{IOV_MAX}	_SC_IOV_MAX
{LINE_MAX}	_SC_LINE_MAX
{LOGIN_NAME_MAX}	_SC_LOGIN_NAME_MAX
{NGROUPS_MAX}	_SC_NGROUPS_MAX
Initial size of <code>getgrgid_r()</code> and <code>getgrnam_r()</code> data buffers	_SC_GETGR_R_SIZE_MAX
Initial size of <code>getpwuid_r()</code> and <code>getpwnam_r()</code> data buffers	_SC_GETPW_R_SIZE_MAX
{MQ_OPEN_MAX}	_SC_MQ_OPEN_MAX
{MQ_PRIO_MAX}	_SC_MQ_PRIO_MAX
Maximum number of execution units that can be made available to run threads†	_SC_NPROCESSORS_CONF
Maximum number of execution units currently available to run threads†	_SC_NPROCESSORS_ONLN
Highest supported signal number +1	_SC_NSIG
{OPEN_MAX}	_SC_OPEN_MAX

Variable	Value of Name
{PAGE_SIZE}	_SC_PAGE_SIZE
{PAGESIZE}	_SC_PAGESIZE
{PTHREAD_DESTRUCTOR_ITERATIONS}	_SC_THREAD_DESTRUCTOR_ITERATIONS
{PTHREAD_KEYS_MAX}	_SC_THREAD_KEYS_MAX
{PTHREAD_STACK_MIN}	_SC_THREAD_STACK_MIN
{PTHREAD_THREADS_MAX}	_SC_THREAD_THREADS_MAX
{RE_DUP_MAX}	_SC_RE_DUP_MAX
{RTSIG_MAX}	_SC_RTSIG_MAX
{SEM_NSEMS_MAX}	_SC_SEM_NSEMS_MAX
{SEM_VALUE_MAX}	_SC_SEM_VALUE_MAX
{SIGQUEUE_MAX}	_SC_SIGQUEUE_MAX
{STREAM_MAX}	_SC_STREAM_MAX
{SYMLOOP_MAX}	_SC_SYMLOOP_MAX
{TIMER_MAX}	_SC_TIMER_MAX
{TTY_NAME_MAX}	_SC_TTY_NAME_MAX
{TZNAME_MAX}	_SC_TZNAME_MAX
_POSIX_ADVISORY_INFO	_SC_ADVISORY_INFO
_POSIX_BARRIERS	_SC_BARRIERS
_POSIX_ASYNCHRONOUS_IO	_SC_ASYNCHRONOUS_IO
_POSIX_CLOCK_SELECTION	_SC_CLOCK_SELECTION
_POSIX_CPUTIME	_SC_CPUTIME

Variable	Value of Name
_POSIX_DEVICE_CONTROL	_SC_DEVICE_CONTROL
_POSIX_FSYNC	_SC_FSYNC
_POSIX_IPV6	_SC_IPV6
_POSIX_JOB_CONTROL	_SC_JOB_CONTROL
_POSIX_MAPPED_FILES	_SC_MAPPED_FILES
_POSIX_MEMLOCK	_SC_MEMLOCK
_POSIX_MEMLOCK_RANGE	_SC_MEMLOCK_RANGE
_POSIX_MEMORY_PROTECTION	_SC_MEMORY_PROTECTION
_POSIX_MESSAGE_PASSING	_SC_MESSAGE_PASSING
_POSIX_MONOTONIC_CLOCK	_SC_MONOTONIC_CLOCK
_POSIX_PRIORITIZED_IO	_SC_PRIORITIZED_IO
_POSIX_PRIORITY_SCHEDULING	_SC_PRIORITY_SCHEDULING
_POSIX_RAW_SOCKETS	_SC_RAW_SOCKETS
_POSIX_READER_WRITER_LOCKS	_SC_READER_WRITER_LOCKS
_POSIX_REALTIME_SIGNALS	_SC_REALTIME_SIGNALS
_POSIX_REGEXP	_SC_REGEXP
_POSIX_SAVED_IDS	_SC_SAVED_IDS
_POSIX_SEMAPHORES	_SC_SEMAPHORES
_POSIX_SHARED_MEMORY_OBJECTS	_SC_SHARED_MEMORY_OBJECTS
_POSIX_SHELL	_SC_SHELL
_POSIX_SPAWN	_SC_SPAWN

Variable	Value of Name
_POSIX_SPIN_LOCKS	_SC_SPIN_LOCKS
_POSIX_SPORADIC_SERVER	_SC_SPORADIC_SERVER
_POSIX_SS_REPL_MAX	_SC_SS_REPL_MAX
_POSIX_SYNCHRONIZED_IO	_SC_SYNCHRONIZED_IO
_POSIX_THREAD_ATTR_STACKADDR	_SC_THREAD_ATTR_STACKADDR
_POSIX_THREAD_ATTR_STACKSIZE	_SC_THREAD_ATTR_STACKSIZE
_POSIX_THREAD_CPUTIME	_SC_THREAD_CPUTIME
_POSIX_THREAD_PRIO_INHERIT	_SC_THREAD_PRIO_INHERIT
_POSIX_THREAD_PRIO_PROTECT	_SC_THREAD_PRIO_PROTECT
_POSIX_THREAD_PRIORITY_SCHEDULING	_SC_THREAD_PRIORITY_SCHEDULING
_POSIX_THREAD_PROCESS_SHARED	_SC_THREAD_PROCESS_SHARED
_POSIX_THREAD_ROBUST_PRIO_INHERIT	_SC_THREAD_ROBUST_PRIO_INHERIT
_POSIX_THREAD_ROBUST_PRIO_PROTECT	_SC_THREAD_ROBUST_PRIO_PROTECT
_POSIX_THREAD_SAFE_FUNCTIONS	_SC_THREAD_SAFE_FUNCTIONS
_POSIX_THREAD_SPORADIC_SERVER	_SC_THREAD_SPORADIC_SERVER
_POSIX_THREADS	_SC_THREADS
_POSIX_TIMEOUTS	_SC_TIMEOUTS
_POSIX_TIMERS	_SC_TIMERS
_POSIX_TYPED_MEMORY_OBJECTS	_SC_TYPED_MEMORY_OBJECTS
_POSIX_VERSION	_SC_VERSION
_POSIX_V8_ILP32_OFF32	_SC_V8_ILP32_OFF32

Variable	Value of Name
_POSIX_V8_ILP32_OFFBIG	_SC_V8_ILP32_OFFBIG
_POSIX_V8_LP64_OFF64	_SC_V8_LP64_OFF64
_POSIX_V8_LPBIG_OFFBIG	_SC_V8_LPBIG_OFFBIG
_POSIX_V7_ILP32_OFF32	_SC_V7_ILP32_OFF32
_POSIX_V7_ILP32_OFFBIG	_SC_V7_ILP32_OFFBIG
_POSIX_V7_LP64_OFF64	_SC_V7_LP64_OFF64
_POSIX_V7_LPBIG_OFFBIG	_SC_V7_LPBIG_OFFBIG
_POSIX2_C_BIND	_SC_2_C_BIND
_POSIX2_C_DEV	_SC_2_C_DEV
_POSIX2_CHAR_TERM	_SC_2_CHAR_TERM
_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
_XOPEN_CRYPT	_SC_XOPEN_CRYPT
_XOPEN_ENH_I18N	_SC_XOPEN_ENH_I18N
_XOPEN_REALTIME	_SC_XOPEN_REALTIME
_XOPEN_REALTIME_THREADS	_SC_XOPEN_REALTIME_THREADS
_XOPEN_SHM	_SC_XOPEN_SHM
_XOPEN_UNIX	_SC_XOPEN_UNIX

Variable	Value of Name
_XOPEN_UUCP	_SC_XOPEN_UUCP
_XOPEN_VERSION	_SC_XOPEN_VERSION

† The nature of an execution unit and the precise conditions under which an execution unit is considered to be available, or can be made available, or how many threads it can execute in parallel, are implementation-defined.

## RETURN VALUE

If `name` is an invalid value, `sysconf()` shall return -1 and set `errno` to indicate the error. If the variable corresponding to `name` is described in `<limits.h>` as a maximum or minimum value and the variable has no limit, `sysconf()` shall return -1 without changing the value of `errno`. Note that indefinite limits do not imply infinite limits; see `<limits.h>`.

Otherwise, `sysconf()` shall return the current variable value on the system. The value returned shall not be more restrictive than the corresponding value described to the application when it was compiled with the implementation's `<limits.h>` or `<unistd.h>`. The value returned for `name` arguments other than `_SC_NPROCESSORS_ONLN` shall not change during the lifetime of the calling process, except that `sysconf(_SC_OPEN_MAX)` may return different values before and after a call to `setrlimit()` which changes the RLIMIT\_NOFILE soft limit.

If the variable corresponding to `name` is dependent on an unsupported option, the results are unspecified.

## ERRORS

The `sysconf()` function shall fail if:

- **[EINVAL]** - The value of the `name` argument is invalid.

---

## APPLICATION USAGE

As -1 is a permissible return value in a successful situation, an application wishing to check for error situations should set `errno` to 0, then call `sysconf()`, and, if it returns -1, check to see if `errno` is non-zero.

Application developers should check whether an option, such as `_POSIX_SPORADIC_SERVER`, is supported prior to obtaining and using values for related variables, such as `_POSIX_SS_REPL_MAX`.

Although the queries `_SC_NPROCESSORS_CONF` and `_SC_NPROCESSORS_ONLN` provide a way for a class of "heavy-load" application to estimate the optimal number of threads that can be created to maximize throughput, real-world environments have complications that affect the actual efficiency that can be achieved. For example:

- There may be more than one "heavy-load" application running on the system.
- The system may be on battery power, and applications should co-ordinate with the system to ensure that a long-running task can pause, resume, and successfully complete even in the event of a power outage.

In case a portable "heavy-load" application wants to avoid the use of extensions, its developers may wish to create threads based on the logical partition of the long-running task, or utilize heuristics such as the ratio between CPU time and real time.

## RATIONALE

This functionality was added in response to requirements of application developers and of system vendors who deal with many international system configurations. It is closely related to `pathconf()` and `fpathconf()`.

Although a conforming application can run on all systems by never demanding more resources than the minimum values published in this volume of POSIX.1-2024, it is useful for that application to be able to use the actual value for the quantity of a resource available on any given system. To do this, the application makes use of the value of a symbolic constant in `<limits.h>` or `<unistd.h>`.

However, once compiled, the application must still be able to cope if the amount of resource available is increased. To that end, an application may need a means of determining the quantity of a resource, or the presence of an option, at execution time.

Two examples are offered:

1. Applications may wish to act differently on systems with or without job control.

Applications vendors who wish to distribute only a single binary package to all instances of a computer architecture would be forced to assume job control is never available if it were to rely solely on the `<unistd.h>` value published in this volume of POSIX.1-2024.

2. International applications vendors occasionally require knowledge of the number of clock ticks per second. Without these facilities, they would be required to either distribute their applications partially in source form or to have 50 Hz and 60 Hz versions for the various countries in which they operate.

It is the knowledge that many applications are actually distributed widely in executable form that leads to this facility. If limited to the most restrictive values in the headers, such applications would have to be prepared to accept the most limited environments offered by the smallest microcomputers. Although this is entirely portable, there was a consensus that they should be able to take advantage of the facilities offered by large systems, without the restrictions associated with source and object distributions.

During the discussions of this feature, it was pointed out that it is almost always possible for an application to discern what a value might be at runtime by suitably testing the various functions themselves. And, in any event, it could always be written to adequately deal with error returns from the various functions. In the end, it was felt that this imposed an unreasonable level of complication and sophistication on the application developer.

This runtime facility is not meant to provide ever-changing values that applications have to check multiple times. The values are seen as changing no more frequently than once per system initialization, such as by a system administrator or operator with an automatic configuration program. This volume of POSIX.1-2024 specifies that they shall not change within the lifetime of the process.

Some values apply to the system overall and others vary at the file system or directory level. The latter are described in [fpathconf\(\)](#).

Note that all values returned must be expressible as integers. String values were considered, but the additional flexibility of this approach was rejected due to its added complexity of implementation and use.

Some values, such as {PATH\_MAX}, are sometimes so large that they must not be used to, say, allocate arrays. The [sysconf\(\)](#) function returns a negative value to show that this symbolic constant is not even defined in this case.

Similar to [pathconf\(\)](#), this permits the implementation not to have a limit. When one resource is infinite, returning an error indicating that some other resource limit has been reached is conforming behavior.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [confstr\(\)](#)
  - [fpathconf\(\)](#)
  - [<limits.h>](#)
  - [<unistd.h>](#)
  - [getconf](#)
-

## 1.256. time — get time

---

### SYNOPSIS

```
#include <time.h>

time_t time(time_t *tloc);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `time()` function shall return the value of time [CX] in seconds since the Epoch.

The `tloc` argument points to an area where the return value is also stored. If `tloc` is a null pointer, no value is stored.

### RETURN VALUE

Upon successful completion, `time()` shall return the value of time. Otherwise, `(time_t)-1` shall be returned.

### ERRORS

The `time()` function may fail if:

- **[EOVERFLOW]** [CX] The number of seconds since the Epoch will not fit in an object of type `time_t`.

*The following sections are informative.*

# EXAMPLES

## Getting the Current Time

The following example uses the `time()` function to calculate the time elapsed, in seconds, since the Epoch, `localtime()` to convert that value to a broken-down time, and `asctime()` to convert the broken-down time values into a printable string.

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

int main(void)
{
    time_t result;

    result = time(NULL);
    printf("%s%ju secs since the Epoch\n",
           asctime(localtime(&result)),
           (uintmax_t)result);
    return(0);
}
```

This example writes the current time to `stdout` in a form like this:

```
Wed Jun 26 10:32:15 1996
835810335 secs since the Epoch
```

## Timing an Event

The following example gets the current time, prints it out in the user's format, and prints the number of minutes to an event being timed.

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

...
time_t now;
int minutes_to_event;
...

time(&now);
minutes_to_event = ...;
printf("The time is ");
puts(asctime(localtime(&now)));
printf("There are %d minutes to the event.\n",
       minutes_to_event);
...
```

# APPLICATION USAGE

None.

## RATIONALE

The `time()` function returns a value in seconds while `clock_gettime()` returns a `struct timespec` (seconds and nanoseconds) and is therefore capable of returning more precise times. The `times()` function is also capable of more precision than `time()` as it returns a value in clock ticks, although it returns the elapsed time since an arbitrary point such as system boot time, not since the epoch.

Earlier versions of this standard allowed the width of `time_t` to be less than 64 bits. A 32-bit signed integer (as used in many historical implementations) fails in the year 2038, and although a 32-bit unsigned integer does not fail until 2106 the preferred solution is to make `time_t` wider rather than to make it unsigned.

On some systems the `time()` function is implemented using a system call that does not return an error condition in addition to the return value. On these systems it is impossible to differentiate between valid and invalid return values and hence overflow conditions cannot be reliably detected.

The use of the `<time.h>` header instead of `<sys/types.h>` allows compatibility with the ISO C standard.

Many historical implementations (including Version 7) and the 1984 /usr/group standard use `long` instead of `time_t`. This volume of POSIX.1-2024 uses the latter type in order to agree with the ISO C standard.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `asctime()`
- `clock()`
- `clock_getres()`
- `ctime()`
- `difftime()`

- `futimens()`
- `gmtime()`
- `localtime()`
- `mktime()`
- `strftime()`
- `strptime()`
- `times()`

XBD `<time.h>`

## CHANGE HISTORY

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

### Issue 6

Extensions beyond the ISO C standard are marked.

The EXAMPLES, RATIONALE, and FUTURE DIRECTIONS sections are added.

### Issue 7

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0663 [106], XSH/TC1-2008/0664 [350], XSH/TC1-2008/0665 [106], XSH/TC1-2008/0666 [350], and XSH/TC1-2008/0667 [350] are applied.

### Issue 8

Austin Group Defect 1330 is applied, removing obsolescent interfaces.

Austin Group Defect 1462 is applied, changing the RATIONALE and FUTURE DIRECTIONS sections.

## 1.257. timer\_create - create a per-process timer

---

### SYNOPSIS

```
#include <signal.h>
#include <time.h>

int timer_create(clockid_t clockid,
                 struct sigevent *restrict evp,
                 timer_t *restrict timerid);
```

### DESCRIPTION

The `timer_create()` function shall create a per-process timer using the specified clock, `clock_id`, as the timing base. The `timer_create()` function shall return, in the location referenced by `timerid`, a timer ID of type `timer_t` used to identify the timer in timer requests. This timer ID shall be unique within the calling process until the timer is deleted. The particular clock, `clock_id`, is defined in `<time.h>`. The timer whose ID is returned shall be in a disarmed state upon return from `timer_create()`.

The `evp` argument, if non-NULL, points to a `sigevent` structure. This structure, allocated by the application, defines the asynchronous notification to occur as specified in Signal Generation and Delivery when the timer expires. If the `evp` argument is NULL, the effect is as if the `evp` argument pointed to a `sigevent` structure with the `sigev_notify` member having the value `SIGEV_SIGNAL`, the `sigev_signo` having a default signal number, and the `sigev_value` member having the value of the timer ID.

Each implementation shall define a set of clocks that can be used as timing bases for per-process timers. All implementations shall support `CLOCK_REALTIME` and `CLOCK_MONOTONIC` as values for `clock_id`.

Per-process timers shall not be inherited by a child process across a `fork()` and shall be disarmed and deleted by an `exec`.

If `_POSIX_CPUTIME` is defined, implementations shall support `clock_id` values representing the CPU-time clock of the calling process.

If `_POSIX_THREAD_CPUTIME` is defined, implementations shall support `clock_id` values representing the CPU-time clock of the calling thread.

It is implementation-defined whether a `timer_create()` function will succeed if the value defined by `clock_id` corresponds to the CPU-time clock of a process

or thread different from the process or thread invoking the function.

If `evp->sigev_notify` is `SIGEV_THREAD` and `sev->sigev_notify_attributes` is not `NULL`, if the attribute pointed to by `sev->sigev_notify_attributes` has a thread stack address specified by a call to `pthread_attr_setstack()`, the results are unspecified if the signal is generated more than once.

## RETURN VALUE

If the call succeeds, `timer_create()` shall return zero and update the location referenced by `timerid` to a `timer_t`, which can be passed to the per-process timer calls. If an error occurs, the function shall return a value of `-1` and set `errno` to indicate the error. The value of `timerid` is undefined if an error occurs.

## ERRORS

The `timer_create()` function shall fail if:

- **[EAGAIN]**
- The system lacks sufficient signal queuing resources to honor the request.
- The calling process has already created all of the timers it is allowed by this implementation.
- **[EINVAL]**
- The specified clock ID is not defined.
- **[ENOTSUP]**
- The implementation does not support the creation of a timer attached to the CPU-time clock that is specified by `clock_id` and associated with a process or thread different from the process or thread invoking `timer_create()`.

## EXAMPLES

None.

## APPLICATION USAGE

If a timer is created which has `evp->sigev_notify` set to SIGEV\_THREAD and the attribute pointed to by `evp->sigev_notify_attributes` has a thread stack address specified by a call to `pthread_attr_setstack()`, the memory dedicated as a thread stack cannot be recovered. The reason for this is that the threads created in response to a timer expiration are created detached, or in an unspecified way if the thread attribute's `detachstate` is PTHREAD\_CREATE\_JOINABLE. In neither case is it valid to call `pthread_join()`, which makes it impossible to determine the lifetime of the created thread which thus means the stack memory cannot be reused.

## RATIONALE

### Periodic Timer Overrun and Resource Allocation

The specified timer facilities may deliver realtime signals (that is, queued signals) on implementations that support this option. Since realtime applications cannot afford to lose notifications of asynchronous events, like timer expirations or asynchronous I/O completions, it must be possible to ensure that sufficient resources exist to deliver the signal when the event occurs. In general, this is not a difficulty because there is a one-to-one correspondence between a request and a subsequent signal generation. If the request cannot allocate the signal delivery resources, it can fail the call with an [EAGAIN] error.

Periodic timers are a special case. A single request can generate an unspecified number of signals. This is not a problem if the requesting process can service the signals as fast as they are generated, thus making the signal delivery resources available for delivery of subsequent periodic timer expiration signals. But, in general, this cannot be assured—processing of periodic timer signals may "overrun"; that is, subsequent periodic timer expirations may occur before the currently pending signal has been delivered.

Also, for signals, according to the POSIX.1-1990 standard, if subsequent occurrences of a pending signal are generated, it is implementation-defined whether a signal is delivered for each occurrence. This is not adequate for some realtime applications. So a mechanism is required to allow applications to detect how many timer expirations were delayed without requiring an indefinite amount of system resources to store the delayed expirations.

The specified facilities provide for an overrun count. The overrun count is defined as the number of extra timer expirations that occurred between the time a timer expiration signal is generated and the time the signal is delivered. The signal-

catching function, if it is concerned with overruns, can retrieve this count on entry. With this method, a periodic timer only needs one "signal queuing resource" that can be allocated at the time of the `timer_create()` function call.

A function is defined to retrieve the overrun count so that an application need not allocate static storage to contain the count, and an implementation need not update this storage asynchronously on timer expirations. But, for some high-frequency periodic applications, the overhead of an additional system call on each timer expiration may be prohibitive. The functions, as defined, permit an implementation to maintain the overrun count in user space, associated with the `timerid`. The `timer_getoverrun()` function can then be implemented as a macro that uses the `timerid` argument (which may just be a pointer to a user space structure containing the counter) to locate the overrun count with no system call overhead. Other implementations, less concerned with this class of applications, can avoid the asynchronous update of user space by maintaining the count in a system structure at the cost of the extra system call to obtain it.

## Timer Expiration Signal Parameters

The realtime signals functionality supports an application-specific datum that is delivered to the extended signal handler. This value is explicitly specified by the application, along with the signal number to be delivered, in a `sigevent` structure. The type of the application-defined value can be either an integer constant or a pointer. This explicit specification of the value, as opposed to always sending the timer ID, was selected based on existing practice.

It is common practice for realtime applications (on non-POSIX systems or realtime extended POSIX systems) to use the parameters of event handlers as the case label of a switch statement or as a pointer to an application-defined data structure. Since timer\_ids are dynamically allocated by the `timer_create()` function, they can be used for neither of these functions without additional application overhead in the signal handler; for example, to search an array of saved timer IDs to associate the ID with a constant or application data structure.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `clock_getres()`
- `timer_delete()`

- `timer_getoverrun()`
- `<signal.h>`
- `<time.h>`

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

### Issue 6

The `timer_create()` function is marked as part of the Timers option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Timers option.

CPU-time clocks are added for alignment with IEEE Std 1003.1d-1999.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding the requirement for the CLOCK\_MONOTONIC clock under the Monotonic Clock option.

The `restrict` keyword is added to the `timer_create()` prototype for alignment with the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/138 is applied, updating the DESCRIPTION and APPLICATION USAGE sections to describe the case when a timer is created with the notification method set to SIGEV\_THREAD.

### Issue 7

The `timer_create()` function is moved from the Timers option to the Base.

### Issue 8

Austin Group Defect 1116 is applied, removing a reference to the Realtime Signals Extension option that existed in earlier versions of this standard.

Austin Group Defect 1346 is applied, requiring support for Monotonic Clock.

---

## 1.258. timer\_delete

---

### NAME

```
#include <time.h>

int timer_delete(timer_t timerid);
```

### DESCRIPTION

The `timer_delete()` function deletes the specified timer, `timerid`, previously created by the `timer_create()` function. If the timer is armed when `timer_delete()` is called, the behavior shall be as if the timer is automatically disarmed before removal. The disposition of pending signals for the deleted timer is unspecified.

The behavior is undefined if the value specified by the `timerid` argument to `timer_delete()` does not correspond to a timer ID returned by `timer_create()` but not yet deleted by `timer_delete()`.

### RETURN VALUE

If successful, the `timer_delete()` function shall return a value of zero. Otherwise, the function shall return a value of -1 and set `errno` to indicate the error.

### ERRORS

No errors are defined.

---

*The following sections are informative.*

### EXAMPLES

None.

### APPLICATION USAGE

None.

### RATIONALE

If an implementation detects that the value specified by the `timerid` argument to `timer_delete()` does not correspond to a timer ID returned by `timer_create()` but not yet deleted by `timer_delete()`, it is recommended that the function should fail and report an [EINVAL] error.

### FUTURE DIRECTIONS

None.

## SEE ALSO

- `timer_create()`
- `<time.h>`

## CHANGE HISTORY

**First released in Issue 5.** Included for alignment with the POSIX Realtime Extension.

### Issue 6

The `timer_delete()` function is marked as part of the Timers option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Timers option.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/139 is applied, updating the ERRORS section so that the [EINVAL] error becomes optional.

### Issue 7

The `timer_delete()` function is moved from the Timers option to the Base.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0369 [659] is applied.

---

## 1.259. timer\_getoverrun, timer\_gettime, timer\_settime — per-process timers

---

### SYNOPSIS

```
#include <time.h>

int timer_getoverrun(timer_t timerid);
int timer_gettime(timer_t timerid, struct itimerspec *value);
int timer_settime(timer_t timerid, int flags,
                  const struct itimerspec *restrict value,
                  struct itimerspec *restrict ovalue);
```

### DESCRIPTION

The `timer_gettime()` function shall store the amount of time until the specified timer, `timerid`, expires and the reload value of the timer into the space pointed to by the `value` argument. The `it_value` member of this structure shall contain the amount of time before the timer expires, or zero if the timer is disarmed. This value is returned as the interval until timer expiration, even if the timer was armed with absolute time. The `it_interval` member of `value` shall contain the reload value last set by `timer_settime()`.

The `timer_settime()` function shall set the time until the next expiration of the timer specified by `timerid` from the `it_value` member of the `value` argument and arm the timer if the `it_value` member of `value` is non-zero. If the specified timer was already armed when `timer_settime()` is called, this call shall reset the time until next expiration to the `value` specified. If the `it_value` member of `value` is zero, the timer shall be disarmed. The effect of disarming or resetting a timer with pending expiration notifications is unspecified.

If the flag `TIMER_ABSTIME` is not set in the argument `flags`, `timer_settime()` shall behave as if the time until next expiration is set to be equal to the interval specified by the `it_value` member of `value`. That is, the timer shall expire in `it_value` nanoseconds from when the call is made. If the flag `TIMER_ABSTIME` is set in the argument `flags`, `timer_settime()` shall behave as if the time until next expiration is set to be equal to the difference between the absolute time specified by the `it_value` member of `value` and the current value of the clock associated with `timerid`. That is, the timer shall expire when the clock reaches the value specified by the `it_value` member of

`value`. If the specified time has already passed, the function shall succeed and the expiration notification shall be made.

The reload value of the timer shall be set to the value specified by the `it_interval` member of `value`. When a timer is armed with a non-zero `it_interval`, a periodic (or repetitive) timer is specified.

Time values that are between two consecutive non-negative integer multiples of the resolution of the specified timer shall be rounded up to the larger multiple of the resolution. Quantization error shall not cause the timer to expire earlier than the rounded time value.

If the argument `ovalue` is not NULL, the `timer_settime()` function shall store, in the location referenced by `ovalue`, a value representing the previous amount of time before the timer would have expired, or zero if the timer was disarmed, together with the previous timer reload value. Timers shall not expire before their scheduled time.

Only a single signal shall be queued to the process for a given timer at any point in time. When a timer for which a signal is still pending expires, no signal shall be queued, and a timer overrun shall occur. When a timer expiration signal is delivered to or accepted by a process, the `timer_getoverrun()` function shall return the timer expiration overrun count for the specified timer. The overrun count returned contains the number of extra timer expirations that occurred between the time the signal was generated (queued) and when it was delivered or accepted, up to but not including an implementation-defined maximum of `{DELAYTIMER_MAX}`. If the number of such extra expirations is greater than or equal to `{DELAYTIMER_MAX}`, then the overrun count shall be set to `{DELAYTIMER_MAX}`. The value returned by `timer_getoverrun()` shall apply to the most recent expiration signal delivery or acceptance for the timer. If no expiration signal has been delivered for the timer, the return value of `timer_getoverrun()` is unspecified.

The behavior is undefined if the value specified by the `timerid` argument to `timer_getoverrun()`, `timer_gettime()`, or `timer_settime()` does not correspond to a timer ID returned by `timer_create()` but not yet deleted by `timer_delete()`.

## RETURN VALUE

If the `timer_getoverrun()` function succeeds, it shall return the timer expiration overrun count as explained above.

If the `timer_gettime()` or `timer_settime()` functions succeed, a value of 0 shall be returned.

If an error occurs for any of these functions, the value -1 shall be returned, and `errno` set to indicate the error.

## ERRORS

The `timer_settime()` function shall fail if:

- **[EINVAL]**

A `value` structure specified a nanosecond value less than zero or greater than or equal to 1000 million, and the `it_value` member of that structure did not specify zero seconds and nanoseconds.

The `timer_settime()` function may fail if:

- **[EINVAL]**

The `it_interval` member of `value` is not zero and the timer was created with notification by creation of a new thread (`sigev_sigev_notify` was `SIGEV_THREAD`) and a fixed stack address has been set in the thread attribute pointed to by `sigev_notify_attributes`.

## APPLICATION USAGE

Using fixed stack addresses is problematic when timer expiration is signaled by the creation of a new thread. Since it cannot be assumed that the thread created for one expiration is finished before the next expiration of the timer, it could happen that two threads use the same memory as a stack at the same time. This is invalid and produces undefined results.

## RATIONALE

Practical clocks tick at a finite rate, with rates of 100 hertz and 1000 hertz being common. The inverse of this tick rate is the clock resolution, also called the clock granularity, which in either case is expressed as a time duration, being 10 milliseconds and 1 millisecond respectively for these common rates. The granularity of practical clocks implies that if one reads a given clock twice in rapid succession, one may get the same time value twice; and that timers must wait for the next clock tick after the theoretical expiration time, to ensure that a timer never returns too soon. Note also that the granularity of the clock may be significantly coarser than the resolution of the data format used to set and get time and interval values. Also note that some implementations may choose to adjust time and/or interval values to exactly match the ticks of the underlying clock.

This volume of POSIX.1-2024 defines functions that allow an application to determine the implementation-supported resolution for the clocks and requires an implementation to document the resolution supported for timers and `nanosleep()` if they differ from the supported clock resolution. This is more of a procurement issue than a runtime application issue.

If an implementation detects that the value specified by the `timerid` argument to `timer_getoverrun()`, `timer_gettime()`, or `timer_settime()` does not correspond to a timer ID returned by `timer_create()` but not yet deleted by `timer_delete()`, it is recommended that the function should fail and report an [EINVAL] error.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `clock_getres()`
- `timer_create()`
- `<time.h>`

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

### Issue 6

The `timer_getoverrun()`, `timer_gettime()`, and `timer_settime()` functions are marked as part of the Timers option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Timers option.

The [EINVAL] error condition is updated to include the following: "and the `it_value` member of that structure did not specify zero seconds and nanoseconds." This change is for IEEE PASC Interpretation 1003.1 #89.

The DESCRIPTION for `timer_getoverrun()` is updated to clarify that "If no expiration signal has been delivered for the timer, or if the Realtime Signals

Extension is not supported, the return value of `timer_getoverrun()` is unspecified".

The `restrict` keyword is added to the `timer_settime()` prototype for alignment with the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/140 is applied, updating the ERRORS section so that the mandatory [EINVAL] error ("The `timerid` argument does not correspond to an ID returned by `timer_create()` but not yet deleted by `timer_delete()`") becomes optional.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/141 is applied, updating the ERRORS section to include an optional [EINVAL] error for the case when a timer is created with the notification method set to `SIGEV_THREAD`. APPLICATION USAGE text is also added.

## Issue 7

The `timer_getoverrun()`, `timer_gettime()`, and `timer_settime()` functions are moved from the Timers option to the Base.

Functionality relating to the Realtime Signals Extension option is moved to the Base.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0370 [659] is applied.

## 1.260. timer\_getoverrun, timer\_gettime, timer\_settime — per-process timers

### SYNOPSIS

```
#include <time.h>

int timer_getoverrun(timer_t timerid);
int timer_gettime(timer_t timerid, struct itimerspec *value);
int timer_settime(timer_t timerid, int flags,
                  const struct itimerspec *restrict value,
                  struct itimerspec *restrict ovalue);
```

### DESCRIPTION

The `timer_gettime()` function shall store the amount of time until the specified timer, `timerid`, expires and the reload value of the timer into the space pointed to by the `value` argument. The `it_value` member of this structure shall contain the amount of time before the timer expires, or zero if the timer is disarmed. This value is returned as the interval until timer expiration, even if the timer was armed with absolute time. The `it_interval` member of `value` shall contain the reload value last set by `timer_settime()`.

The `timer_settime()` function shall set the time until the next expiration of the timer specified by `timerid` from the `it_value` member of the `value` argument and arm the timer if the `it_value` member of `value` is non-zero. If the specified timer was already armed when `timer_settime()` is called, this call shall reset the time until next expiration to the `value` specified. If the `it_value` member of `value` is zero, the timer shall be disarmed. The effect of disarming or resetting a timer with pending expiration notifications is unspecified.

If the flag `TIMER_ABSTIME` is not set in the argument `flags`, `timer_settime()` shall behave as if the time until next expiration is set to be equal to the interval specified by the `it_value` member of `value`. That is, the timer shall expire in `it_value` nanoseconds from when the call is made. If the flag `TIMER_ABSTIME` is set in the argument `flags`, `timer_settime()` shall behave as if the time until next expiration is set to be equal to the difference between the absolute time specified by the `it_value` member of `value` and the current value of the clock associated with `timerid`. That is, the timer shall expire when the clock reaches the value specified by the `it_value` member of

`value`. If the specified time has already passed, the function shall succeed and the expiration notification shall be made.

The reload value of the timer shall be set to the value specified by the `it_interval` member of `value`. When a timer is armed with a non-zero `it_interval`, a periodic (or repetitive) timer is specified.

Time values that are between two consecutive non-negative integer multiples of the resolution of the specified timer shall be rounded up to the larger multiple of the resolution. Quantization error shall not cause the timer to expire earlier than the rounded time value.

If the argument `ovalue` is not NULL, the `timer_settime()` function shall store, in the location referenced by `ovalue`, a value representing the previous amount of time before the timer would have expired, or zero if the timer was disarmed, together with the previous timer reload value. Timers shall not expire before their scheduled time.

Only a single signal shall be queued to the process for a given timer at any point in time. When a timer for which a signal is still pending expires, no signal shall be queued, and a timer overrun shall occur. When a timer expiration signal is delivered to or accepted by a process, the `timer_getoverrun()` function shall return the timer expiration overrun count for the specified timer. The overrun count returned contains the number of extra timer expirations that occurred between the time the signal was generated (queued) and when it was delivered or accepted, up to but not including an implementation-defined maximum of `{DELAYTIMER_MAX}`. If the number of such extra expirations is greater than or equal to `{DELAYTIMER_MAX}`, then the overrun count shall be set to `{DELAYTIMER_MAX}`. The value returned by `timer_getoverrun()` shall apply to the most recent expiration signal delivery or acceptance for the timer. If no expiration signal has been delivered for the timer, the return value of `timer_getoverrun()` is unspecified.

The behavior is undefined if the value specified by the `timerid` argument to `timer_getoverrun()`, `timer_gettime()`, or `timer_settime()` does not correspond to a timer ID returned by `timer_create()` but not yet deleted by `timer_delete()`.

## RETURN VALUE

If the `timer_getoverrun()` function succeeds, it shall return the timer expiration overrun count as explained above.

If the `timer_gettime()` or `timer_settime()` functions succeed, a value of 0 shall be returned.

If an error occurs for any of these functions, the value -1 shall be returned, and `errno` set to indicate the error.

## ERRORS

The `timer_settime()` function shall fail if:

- **[EINVAL]** A `value` structure specified a nanosecond value less than zero or greater than or equal to 1000 million, and the `it_value` member of that structure did not specify zero seconds and nanoseconds.

The `timer_settime()` function may fail if:

- **[EINVAL]** The `it_interval` member of `value` is not zero and the timer was created with notification by creation of a new thread (`sigev_notify` was `SIGEV_THREAD`) and a fixed stack address has been set in the thread attribute pointed to by `sigev_notify_attributes`.

## APPLICATION USAGE

Using fixed stack addresses is problematic when timer expiration is signaled by the creation of a new thread. Since it cannot be assumed that the thread created for one expiration is finished before the next expiration of the timer, it could happen that two threads use the same memory as a stack at the same time. This is invalid and produces undefined results.

## RATIONALE

Practical clocks tick at a finite rate, with rates of 100 hertz and 1000 hertz being common. The inverse of this tick rate is the clock resolution, also called the clock granularity, which in either case is expressed as a time duration, being 10 milliseconds and 1 millisecond respectively for these common rates. The granularity of practical clocks implies that if one reads a given clock twice in rapid succession, one may get the same time value twice; and that timers must wait for the next clock tick after the theoretical expiration time, to ensure that a timer never returns too soon. Note also that the granularity of the clock may be significantly coarser than the resolution of the data format used to set and get time and interval values. Also note that some implementations may choose to adjust time and/or interval values to exactly match the ticks of the underlying clock.

This volume of POSIX.1-2024 defines functions that allow an application to determine the implementation-supported resolution for the clocks and requires an implementation to document the resolution supported for timers and `nanosleep()` if they differ from the supported clock resolution. This is more of a procurement issue than a runtime application issue.

If an implementation detects that the value specified by the `timerid` argument to `timer_getoverrun()`, `timer_gettime()`, or `timer_settime()` does not correspond to a timer ID returned by `timer_create()` but not yet deleted by `timer_delete()`, it is recommended that the function should fail and report an [EINVAL] error.

## SEE ALSO

- `clock_getres()`
- `timer_create()`
- `<time.h>`

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

### Issue 6

The `timer_getoverrun()`, `timer_gettime()`, and `timer_settime()` functions are marked as part of the Timers option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Timers option.

The [EINVAL] error condition is updated to include the following: "and the `it_value` member of that structure did not specify zero seconds and nanoseconds." This change is for IEEE PASC Interpretation 1003.1 #89.

The DESCRIPTION for `timer_getoverrun()` is updated to clarify that "If no expiration signal has been delivered for the timer, or if the Realtime Signals Extension is not supported, the return value of `timer_getoverrun()` is unspecified".

The `restrict` keyword is added to the `timer_settime()` prototype for alignment with the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/140 is applied, updating the ERRORS section so that the mandatory [EINVAL] error ("The `timerid` argument does not correspond to an ID returned by `timer_create()` but not yet deleted by `timer_delete()`") becomes optional.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/141 is applied, updating the ERRORS section to include an optional [EINVAL] error for the case when a timer is created with the notification method set to `SIGEV_THREAD`. APPLICATION USAGE text is also added.

## Issue 7

The `timer_getoverrun()`, `timer_gettime()`, and `timer_settime()` functions are moved from the Timers option to the Base.

Functionality relating to the Realtime Signals Extension option is moved to the Base.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0370 [659] is applied.

## 1.261. timer\_getoverrun, timer\_gettime, timer\_settime — per-process timers

### SYNOPSIS

```
#include <time.h>

int timer_getoverrun(timer_t timerid);
int timer_gettime(timer_t timerid, struct itimerspec *value);
int timer_settime(timer_t timerid, int flags,
                  const struct itimerspec *restrict value,
                  struct itimerspec *restrict ovalue);
```

### DESCRIPTION

The `timer_gettime()` function shall store the amount of time until the specified timer, `timerid`, expires and the reload value of the timer into the space pointed to by the `value` argument. The `it_value` member of this structure shall contain the amount of time before the timer expires, or zero if the timer is disarmed. This value is returned as the interval until timer expiration, even if the timer was armed with absolute time. The `it_interval` member of `value` shall contain the reload value last set by `timer_settime()`.

The `timer_settime()` function shall set the time until the next expiration of the timer specified by `timerid` from the `it_value` member of the `value` argument and arm the timer if the `it_value` member of `value` is non-zero. If the specified timer was already armed when `timer_settime()` is called, this call shall reset the time until next expiration to the `value` specified. If the `it_value` member of `value` is zero, the timer shall be disarmed. The effect of disarming or resetting a timer with pending expiration notifications is unspecified.

If the flag `TIMER_ABSTIME` is not set in the argument `flags`, `timer_settime()` shall behave as if the time until next expiration is set to be equal to the interval specified by the `it_value` member of `value`. That is, the timer shall expire in `it_value` nanoseconds from when the call is made. If the flag `TIMER_ABSTIME` is set in the argument `flags`, `timer_settime()` shall behave as if the time until next expiration is set to be equal to the difference between the absolute time specified by the `it_value` member of `value` and the current value of the clock associated with `timerid`. That is, the timer shall expire when the clock reaches the value specified by the `it_value` member of

`value`. If the specified time has already passed, the function shall succeed and the expiration notification shall be made.

The reload value of the timer shall be set to the value specified by the `it_interval` member of `value`. When a timer is armed with a non-zero `it_interval`, a periodic (or repetitive) timer is specified.

Time values that are between two consecutive non-negative integer multiples of the resolution of the specified timer shall be rounded up to the larger multiple of the resolution. Quantization error shall not cause the timer to expire earlier than the rounded time value.

If the argument `ovalue` is not NULL, the `timer_settime()` function shall store, in the location referenced by `ovalue`, a value representing the previous amount of time before the timer would have expired, or zero if the timer was disarmed, together with the previous timer reload value. Timers shall not expire before their scheduled time.

Only a single signal shall be queued to the process for a given timer at any point in time. When a timer for which a signal is still pending expires, no signal shall be queued, and a timer overrun shall occur. When a timer expiration signal is delivered to or accepted by a process, the `timer_getoverrun()` function shall return the timer expiration overrun count for the specified timer. The overrun count returned contains the number of extra timer expirations that occurred between the time the signal was generated (queued) and when it was delivered or accepted, up to but not including an implementation-defined maximum of `{DELAYTIMER_MAX}`. If the number of such extra expirations is greater than or equal to `{DELAYTIMER_MAX}`, then the overrun count shall be set to `{DELAYTIMER_MAX}`. The value returned by `timer_getoverrun()` shall apply to the most recent expiration signal delivery or acceptance for the timer. If no expiration signal has been delivered for the timer, the return value of `timer_getoverrun()` is unspecified.

The behavior is undefined if the value specified by the `timerid` argument to `timer_getoverrun()`, `timer_gettime()`, or `timer_settime()` does not correspond to a timer ID returned by `timer_create()` but not yet deleted by `timer_delete()`.

## RETURN VALUE

If the `timer_getoverrun()` function succeeds, it shall return the timer expiration overrun count as explained above.

If the `timer_gettime()` or `timer_settime()` functions succeed, a value of 0 shall be returned.

If an error occurs for any of these functions, the value -1 shall be returned, and `errno` set to indicate the error.

## ERRORS

The `timer_settime()` function shall fail if:

- **EINVAL**
- A `value` structure specified a nanosecond value less than zero or greater than or equal to 1000 million, and the `it_value` member of that structure did not specify zero seconds and nanoseconds.

The `timer_settime()` function may fail if:

- **EINVAL**
- The `it_interval` member of `value` is not zero and the timer was created with notification by creation of a new thread (`sigev_sigev_notify` was `SIGEV_THREAD`) and a fixed stack address has been set in the thread attribute pointed to by `sigev_notify_attributes`.

## APPLICATION USAGE

Using fixed stack addresses is problematic when timer expiration is signaled by the creation of a new thread. Since it cannot be assumed that the thread created for one expiration is finished before the next expiration of the timer, it could happen that two threads use the same memory as a stack at the same time. This is invalid and produces undefined results.

## RATIONALE

Practical clocks tick at a finite rate, with rates of 100 hertz and 1000 hertz being common. The inverse of this tick rate is the clock resolution, also called the clock granularity, which in either case is expressed as a time duration, being 10 milliseconds and 1 millisecond respectively for these common rates. The granularity of practical clocks implies that if one reads a given clock twice in rapid succession, one may get the same time value twice; and that timers must wait for the next clock tick after the theoretical expiration time, to ensure that a timer never returns too soon. Note also that the granularity of the clock may be significantly coarser than the resolution of the data format used to set and get time and interval values. Also note that some implementations may choose to adjust time and/or interval values to exactly match the ticks of the underlying clock.

This volume of POSIX.1-2024 defines functions that allow an application to determine the implementation-supported resolution for the clocks and requires an implementation to document the resolution supported for timers and `nanosleep()` if they differ from the supported clock resolution. This is more of a procurement issue than a runtime application issue.

If an implementation detects that the value specified by the `timerid` argument to `timer_getoverrun()`, `timer_gettime()`, or `timer_settime()` does not correspond to a timer ID returned by `timer_create()` but not yet deleted by `timer_delete()`, it is recommended that the function should fail and report an [EINVAL] error.

## SEE ALSO

- `clock_getres()`
- `timer_create()`
- `<time.h>`

## CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

### Issue 6

The `timer_getoverrun()`, `timer_gettime()`, and `timer_settime()` functions are marked as part of the Timers option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Timers option.

The [EINVAL] error condition is updated to include the following: "and the `it_value` member of that structure did not specify zero seconds and nanoseconds." This change is for IEEE PASC Interpretation 1003.1 #89.

The DESCRIPTION for `timer_getoverrun()` is updated to clarify that "If no expiration signal has been delivered for the timer, or if the Realtime Signals Extension is not supported, the return value of `timer_getoverrun()` is unspecified".

The `restrict` keyword is added to the `timer_settime()` prototype for alignment with the ISO/IEC 9899:1999 standard.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/140 is applied, updating the ERRORS section so that the mandatory [EINVAL] error ("The `timerid` argument does not correspond to an ID returned by `timer_create()` but not yet deleted by `timer_delete()`") becomes optional.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/141 is applied, updating the ERRORS section to include an optional [EINVAL] error for the case when a timer is created with the notification method set to `SIGEV_THREAD`. APPLICATION USAGE text is also added.

## Issue 7

The `timer_getoverrun()`, `timer_gettime()`, and `timer_settime()` functions are moved from the Timers option to the Base.

Functionality relating to the Realtime Signals Extension option is moved to the Base.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0370 [659] is applied.

## 1.262. `tolower`, `tolower_l` — transliterate uppercase characters to lowercase

---

### SYNOPSIS

```
#include <ctype.h>

int tolower(int c);

[CX] int tolower_l(int c, locale_t locale);
```

### DESCRIPTION

For `tolower()`: [CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `tolower()` [CX] and `tolower_l()` functions have as a domain a type `int`, the value of which is representable as an `unsigned char` or the value of EOF. If the argument has any other value, the behavior is undefined. If the argument of `tolower()` [CX] or `tolower_l()` represents an uppercase letter, and there exists a corresponding lowercase letter as defined by character type information in the current locale [CX] or in the locale represented by `locale`, respectively (category `LC_CTYPE`), the result shall be the corresponding lowercase letter. All other arguments in the domain are returned unchanged.

[CX] The behavior is undefined if the `locale` argument to `tolower_l()` is the special locale object `LC_GLOBAL_LOCALE` or is not a valid locale object handle.

### RETURN VALUE

Upon successful completion, the `tolower()` [CX] and `tolower_l()` functions shall return the lowercase letter corresponding to the argument passed; otherwise, they shall return the argument unchanged.

### ERRORS

No errors are defined.

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `setlocale()`
- `uselocale()`
- XBD 7. Locale
- `<ctype.h>`
- `<locale.h>`

## CHANGE HISTORY

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

### Issue 6

Extensions beyond the ISO C standard are marked.

### Issue 7

The `tolower_l()` function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0671 [283] and XSH/TC1-2008/0672 [283] are applied.

---

## 1.263. toupper, toupper\_l — transliterate lowercase characters to uppercase

---

### SYNOPSIS

```
#include <ctype.h>

int toupper(int c);

/* XSI option */
int toupper_l(int c, locale_t locale);
```

### DESCRIPTION

For `toupper()`:

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `toupper()` and `toupper_l()` functions have as a domain a type `int`, the value of which is representable as an `unsigned char` or the value of EOF. If the argument has any other value, the behavior is undefined.

If the argument of `toupper()` or `toupper_l()` represents a lowercase letter, and there exists a corresponding uppercase letter as defined by character type information in the current locale or in the locale represented by `locale`, respectively (category `LC_CTYPE`), the result shall be the corresponding uppercase letter.

All other arguments in the domain are returned unchanged.

The behavior is undefined if the `locale` argument to `toupper_l()` is the special locale object `LC_GLOBAL_LOCALE` or is not a valid locale object handle.

### RETURN VALUE

Upon successful completion, `toupper()` and `toupper_l()` shall return the uppercase letter corresponding to the argument passed; otherwise, they shall return the argument unchanged.

# ERRORS

No errors are defined.

## EXAMPLES

None.

## APPLICATION USAGE

None.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `setlocale()`
- `uselocale()`
- XBD 7. Locale
- `<ctype.h>`
- `<locale.h>`

## CHANGE HISTORY

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

### Issue 6

Extensions beyond the ISO C standard are marked.

## Issue 7

SD5-XSH-ERN-181 is applied, clarifying the RETURN VALUE section.

The `toupper_l()` function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0673 [283] and XSH/TC1-2008/0674 [283] are applied.

## 1.264. tzset - set timezone conversion information

---

### SYNOPSIS

```
#include <time.h>

[XSI] extern int daylight;
[XSI] extern long timezone;

[CX] extern char *tzname[2];
[CX] void tzset(void);
```

### DESCRIPTION

The `tzset()` function shall use the value of the environment variable `TZ` to set time conversion information used by `ctime()`, `localtime()`, `mktime()`, and `strftime()`. If `TZ` is absent from the environment, implementation-defined default timezone information shall be used.

The `tzset()` function shall set the external variable `tzname` as follows:

```
tzname[0] = "std";
tzname[1] = "dst";
```

where `std` and `dst` are as described in XBD 8. Environment Variables.

[XSI] The `tzset()` function also shall set the external variable `daylight` to 0 if Daylight Saving Time conversions should never be applied for the timezone in use; otherwise, non-zero. The external variable `timezone` shall be set to the difference, in seconds, between Coordinated Universal Time (UTC) and local standard time.

If a thread accesses `tzname`, [XSI] `daylight`, or `timezone` directly while another thread is in a call to `tzset()`, or to any function that is required or allowed to set timezone information as if by calling `tzset()`, the behavior is undefined.

### RETURN VALUE

The `tzset()` function shall not return a value.

# ERRORS

No errors are defined.

## EXAMPLES

Example `TZ` variables and their timezone differences are given in the table below:

<code>TZ</code>	<code>timezone</code>
EST5EDT	5*60*60
GMT0	0*60*60
JST-9	-9*60*60
MET-1MEST	-1*60*60
MST7MDT	7*60*60
PST8PDT	8*60*60

## APPLICATION USAGE

Since the `ctime()`, `localtime()`, `mktime()`, `strftime()`, and `strftime_l()` functions are required to set timezone information as if by calling `tzset()`, there is no need for an explicit `tzset()` call before using these functions. However, portable applications should call `tzset()` explicitly before using `localtime_r()` because setting timezone information is optional for that function.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- `ctime()`
- `localtime()`
- `mktime()`
- `strftime()`
- XBD 8. Environment Variables
- `<time.h>`

## CHANGE HISTORY

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

### Issue 6

The example is corrected.

### Issue 7

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0377 [880] is applied.

### Issue 8

Austin Group Defect 1253 is applied, changing "Daylight Savings" to "Daylight Saving".

Austin Group Defect 1410 is applied, removing the `ctime_r()` function.

## 1.265. uname - get system name

---

### Synopsis

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

int uname(struct utsname *name);
```

### Description

The `uname()` function shall store information identifying the current system in the structure pointed to by `name`.

The `uname()` function uses the `utsname` structure defined in `<sys/utsname.h>`.

The `uname()` function shall return a string naming the current system in the character array `sysname`. Similarly, `nodename` shall contain the name of this node within an implementation-defined communications network. The arrays `release` and `version` shall further identify the operating system. The array `machine` shall contain a name that identifies the hardware that the system is running on.

The format of each member is implementation-defined.

### Return Value

Upon successful completion, `uname()` shall return 0; otherwise, it shall return -1 and set `errno` to indicate the error.

### Errors

No errors are defined.

# Examples

## Example 1: Get System Information

```
#include <stdio.h>
#include <sys/utsname.h>

int main(void)
{
    struct utsname name;

    if (uname(&name) == -1) {
        perror("uname");
        return 1;
    }

    printf("System name: %s\n", name.sysname);
    printf("Node name:   %s\n", name.nodename);
    printf("Release:     %s\n", name.release);
    printf("Version:     %s\n", name.version);
    printf("Machine:     %s\n", name.machine);

    return 0;
}
```

## Application Usage

The values of the structure members are not constrained to have any relation to the version of this volume of POSIX.1-2024 implemented in the operating system. An application should instead depend on `_POSIX_VERSION` and related constants defined in `<unistd.h>`.

This volume of POSIX.1-2024 does not define the sizes of the members of the structure and permits them to be of different sizes, although most implementations define them all to be the same size: eight bytes plus one byte for the string terminator. That size for `nodename` is not enough for use with many networks.

## Rationale

The `uname()` function originated in System III, System V, and related implementations, and it does not exist in Version 7 or 4.3 BSD. The values it returns are set at system compile time in those historical implementations.

4.3 BSD has `gethostname()` and `gethostid()`, which return a symbolic name and a numeric value, respectively. There are related `sethostname()` and `sethostid()` functions that are used to set the values the other two functions return. The former functions are included in this specification, the latter are not.

## Future Directions

None.

## See Also

- `gethostname()`
- `gethostid()`
- `<sys/utsname.h>`
- `<unistd.h>`

## Change History

- First released in Issue 1. Derived from System V documentation.
- Copyright © 2018-2024, The Open Group. All Rights Reserved.

SPDX-License-Identifier: CC-BY-4.0

## 1.266. ungetc — push byte back into input stream

---

### SYNOPSIS

```
#include <stdio.h>

int ungetc(int c, FILE *stream);
```

### DESCRIPTION

The `ungetc()` function shall push the byte specified by `c` (converted to an **unsigned char**) back onto the input stream pointed to by `stream`. The pushed-back bytes shall be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by `stream`) to a file-positioning function (`fseek()`, `fseeko()`, `fsetpos()`, or `rewind()`) or `fflush()` shall discard any pushed-back bytes for the stream. The external storage corresponding to the stream shall be unchanged.

One byte of push-back shall be provided. If `ungetc()` is called too many times on the same stream without an intervening read or file-positioning operation on that stream, the operation may fail.

If the value of `c` equals that of the macro EOF, the operation shall fail and the input stream shall be left unchanged.

A successful call to `ungetc()` shall clear the end-of-file indicator for the stream. The file-position indicator for the stream shall be decremented by each successful call to `ungetc()`; if its value was 0 before a call, its value is unspecified after the call. The value of the file-position indicator after all pushed-back bytes have been read shall be the same as it was before the bytes were pushed back.

### RETURN VALUE

Upon successful completion, `ungetc()` shall return the byte pushed back after conversion. Otherwise, it shall return EOF.

### ERRORS

No errors are defined.

# APPLICATION USAGE

None.

## RATIONALE

The ISO C standard includes the text "The value of the file position indicator for the stream after reading or discarding all pushed-back characters shall be the same as it was before the characters were pushed back." POSIX.1 omits "or discarding" from this because it is redundant—in the ISO C standard the discarding is done by file positioning functions and does not affect the position set by those functions. In particular, a relative seek using `fseek()` or `fseeko()` with SEEK\_CUR adjusts the position relative to the position on entry to the function, not the position after the pushed-back bytes have been discarded. POSIX.1 also requires `fflush()` to discard pushed back bytes in situations where the ISO C standard says the behavior of `fflush()` is undefined.

## FUTURE DIRECTIONS

The ISO C standard states that the use of `ungetc()` on a binary stream where the file position indicator is zero prior to the call is an obsolescent feature. In POSIX.1 there is no distinction between binary and text streams, so this applies to all streams. This feature may be removed in a future version of this standard.

## SEE ALSO

- 2.5 Standard I/O Streams
- `fseek()`
- `getc()`
- `fsetpos()`
- `read()`
- `rewind()`
- `setbuf()`
- XBD `<stdio.h>`

# CHANGE HISTORY

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

## Issue 7

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0687 [87,93], XSH/TC1-2008/0688 [87], and XSH/TC1-2008/0689 [14] are applied.

## Issue 8

Austin Group Defect 701 is applied, clarifying how the file-position indicator for the stream is updated.

Austin Group Defect 1302 is applied, changing the FUTURE DIRECTIONS section.

## 1.267. unsetenv

---

### SYNOPSIS

```
#include <stdlib.h>

int unsetenv(const char *name);
```

### DESCRIPTION

The `unsetenv()` function shall remove an environment variable from the environment of the calling process. The `name` argument points to a string, which is the name of the variable to be removed. The named argument shall not contain an '=' character. If the named variable does not exist in the current environment, the environment shall be unchanged and the function is considered to have completed successfully.

The `unsetenv()` function shall update the list of pointers to which `environ` points.

The `unsetenv()` function need not be thread-safe.

### RETURN VALUE

Upon successful completion, zero shall be returned. Otherwise, -1 shall be returned, `errno` set to indicate the error, and the environment shall be unchanged.

### ERRORS

The `unsetenv()` function shall fail if:

- `[EINVAL]`
- The `name` argument points to an empty string, or points to a string containing an '=' character.

### EXAMPLES

None.

# APPLICATION USAGE

None.

## RATIONALE

Refer to the RATIONALE section in [setenv\(\)](#).

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [getenv\(\)](#)
- [setenv\(\)](#)
- XBD [<stdlib.h>](#), [<sys/types.h>](#)

## CHANGE HISTORY

First released in Issue 6. Derived from the IEEE P1003.1a draft standard.

### Issue 7

Austin Group Interpretation 1003.1-2001 #156 is applied.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0698 [167] and XSH/TC1-2008/0699 [185] are applied.

---

## 1.268. va\_arg

---

### SYNOPSIS

```
#include <stdarg.h>

type va_arg(va_list ap, type);
void va_copy(va_list dest, va_list src);
void va_end(va_list ap);
void va_start(va_list ap, argN);
```

### DESCRIPTION

Refer to XBD [`<stdarg.h>`](#)

[return to top of page](#)

---

## 1.269. va\_copy

---

### SYNOPSIS

```
#include <stdarg.h>

type va_arg(va_list ap, type);
void va_copy(va_list dest, va_list src);
void va_end(va_list ap);
void va_start(va_list ap, argN);
```

### DESCRIPTION

Refer to XBD

---

## 1.270. va\_end

---

### SYNOPSIS

```
#include <stdarg.h>

void va_end(va_list ap);
```

### DESCRIPTION

The `va_end()` macro is used to clean up after traversal of a variable argument list. It invalidates the `va_list` object `ap` for use (unless `va_start()` or `va_copy()` is invoked again).

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

### REQUIREMENTS

Each invocation of the `va_start()` and `va_copy()` macros shall be matched by a corresponding invocation of the `va_end()` macro in the same function.

### USAGE RULES

- The object `ap` may be passed as an argument to another function
- If that function invokes the `va_arg()` macro with parameter `ap`, the value of `ap` in the calling function is unspecified and shall be passed to the `va_end()` macro prior to any further reference to `ap`
- Neither the `va_copy()` nor `va_start()` macro shall be invoked to reinitialize `ap` without an intervening invocation of the `va_end()` macro for the same `ap`

## EXAMPLE

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

#define MAXARGS 31

/*
 * execl is called by
 * execl(file, arg1, arg2, ..., (char *)());
 */
int execl(const char *file, const char *args, ...)
{
    va_list ap;
    char *array[MAXARGS + 1];
    int argno = 0;

    va_start(ap, args);
    while (args != 0 && argno < MAXARGS)
    {
        array[argno++] = args;
        args = va_arg(ap, const char *);
    }
    array[argno] = (char *) 0;
    va_end(ap);
    return execv(file, array);
}
```

## RELATED FUNCTIONS

- [va\\_start\(\)](#) - Initialize a variable argument list
- [va\\_arg\(\)](#) - Retrieve the next argument from a variable argument list
- [va\\_copy\(\)](#) - Copy a variable argument list state

## SEE ALSO

- [printf\(\)](#)
- [execl\(\)](#)

## 1.271. va\_start, va\_arg, va\_copy, va\_end - handle variable argument list

---

### SYNOPSIS

```
#include <stdarg.h>

type va_arg(va_list ap, type);
void va_copy(va_list dest, va_list src);
void va_end(va_list ap);
void va_start(va_list ap, argN);
```

### DESCRIPTION

The `<stdarg.h>` header shall contain a set of macros which allows portable functions that accept variable argument lists to be written. Functions that have variable argument lists (such as `printf()`) but do not use these macros are inherently non-portable, as different systems use different argument-passing conventions.

The `<stdarg.h>` header shall define the `va_list` type for variables used to traverse the list.

#### `va_start()`

The `va_start()` macro is invoked to initialize `ap` to the beginning of the list before any calls to `va_arg()`.

The parameter `argN` is the identifier of the rightmost parameter in the variable parameter list in the function definition (the one just before the `...`). If the parameter `argN` is declared with the `register` storage class, with a function type or array type, or with a type that is not compatible with the type that results after application of the default argument promotions, the behavior is undefined.

#### `va_copy()`

The `va_copy()` macro initializes `dest` as a copy of `src`, as if the `va_start()` macro had been applied to `dest` followed by the same sequence of uses of the `va_arg()` macro as had previously been used to reach the present state of `src`. Neither the `va_copy()` nor `va_start()` macro shall be invoked

to reinitialize `dest` without an intervening invocation of the `va_end()` macro for the same `dest`.

## va\_arg()

The `va_arg()` macro shall return the next argument in the list pointed to by `ap`. Each invocation of `va_arg()` modifies `ap` so that the values of successive arguments are returned in turn. The `type` parameter shall be a type name specified such that the type of a pointer to an object that has the specified type can be obtained simply by postfixing a '\*' to type.

If there is no actual next argument, or if `type` is not compatible with the type of the actual next argument (as promoted according to the default argument promotions), the behavior is undefined, except for the following cases:

- One type is a signed integer type, the other type is the corresponding unsigned integer type, and the value is representable in both types.
- One type is a pointer to `void` and the other is a pointer to a character type.
- [XSI] Both types are pointers.

Different types can be mixed, but it is up to the routine to know what type of argument is expected.

## va\_end()

The `va_end()` macro is used to clean up; it invalidates `ap` for use (unless `va_start()` or `va_copy()` is invoked again).

# USAGE

Each invocation of the `va_start()` and `va_copy()` macros shall be matched by a corresponding invocation of the `va_end()` macro in the same function.

Multiple traversals, each bracketed by `va_start()` ... `va_end()`, are possible.

The object `ap` may be passed as an argument to another function; if that function invokes the `va_arg()` macro with parameter `ap`, the value of `ap` in the calling function is unspecified and shall be passed to the `va_end()` macro prior to any further reference to `ap`.

# EXAMPLE

This example is a possible implementation of `execl()`:

```
#include <stdarg.h>

#define MAXARGS 31

/*
 * execl is called by
 * execl(file, arg1, arg2, ..., (char *)0);
 */
int execl(const char *file, const char *args, ...)
{
    va_list ap;
    char *array[MAXARGS + 1];
    int argno = 0;

    va_start(ap, args);
    while (args != 0 && argno < MAXARGS)
    {
        array[argno++] = args;
        args = va_arg(ap, const char *);
    }
    array[argno] = (char *) 0;
    va_end(ap);
    return execv(file, array);
}
```

## SEE ALSO

- [printf\(\)](#)
  - [execl\(\)](#)
  - [execv\(\)](#)
-

## 1.272. vfprintf

### SYNOPSIS

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

[CX] int vasprintf(char **restrict ptr, const char *restrict format,
                    va_list ap);
int vdprintf(int fildes, const char *restrict format, va_list ap);

int vfprintf(FILE *restrict stream, const char *restrict format,
              va_list ap);
int vprintf(const char *restrict format, va_list ap);
int vsnprintf(char *restrict s, size_t n, const char *restrict format,
              va_list ap);
int vsprintf(char *restrict s, const char *restrict format, va_list ap);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The [CX] `vasprintf()`, `vdprintf()`, `vfprintf()`, `vprintf()`, `vsnprintf()`, and `vsprintf()` functions shall be equivalent to the [CX] `asprintf()`, `dprintf()`, `fprintf()`, `printf()`, `snprintf()`, and `sprintf()` functions respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by `<stdarg.h>`.

These functions shall not invoke the `va_end` macro. As these functions invoke the `va_arg` macro, the value of `ap` after the return is unspecified.

### RETURN VALUE

Refer to `fprintf()`.

# ERRORS

Refer to [fprintf\(\)](#).

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

Applications using these functions should call [va\\_end\(ap\)](#) afterwards to clean up.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

[2.5 Standard I/O Streams](#), [fprintf\(\)](#)

XBD [<stdarg.h>](#), [<stdio.h>](#)

## CHANGE HISTORY

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

### Issue 5

The [vsnprintf\(\)](#) function is added.

## Issue 6

The `vfprintf()`, `vprintf()`, `vsnprintf()`, and `vsprintf()` functions are updated for alignment with the ISO/IEC 9899:1999 standard.

## Issue 7

The `vdprintf()` function is added to complement the `dprintf()` function from The Open Group Technical Standard, 2006, Extended API Set Part 1.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0703 [14] is applied.

## Issue 8

Austin Group Defect 1496 is applied, adding the `vasprintf()` function.

## 1.273. vfscanf, vscanf, vsscanf — format input of a stdarg argument list

### SYNOPSIS

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

int vfscanf(FILE *restrict stream,
             const char *restrict format,
             va_list arg);

int vscanf(const char *restrict format,
           va_list arg);

int vsscanf(const char *restrict s,
            const char *restrict format,
            va_list arg);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `vscanf()`, `vfscanf()`, and `vsscanf()` functions shall be equivalent to the `scanf()`, `fscanf()`, and `sscanf()` functions, respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined in the `<stdarg.h>` header. These functions shall not invoke the `va_end` macro. As these functions invoke the `va_arg` macro, the value of `ap` after the return is unspecified.

### RETURN VALUE

Refer to `fscanf()`.

### ERRORS

Refer to `fscanf()`.

---

## EXAMPLES

None.

## APPLICATION USAGE

Applications using these functions should call `va_end(ap)` afterwards to clean up.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [2.5 Standard I/O Streams](#)
- `fscanf()`
- XBD `<stdarg.h>`, `<stdio.h>`

## CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

### Issue 7

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0704 [14] is applied.

---

## 1.274. vprintf

### SYNOPSIS

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

int vasprintf(char **restrict ptr, const char *restrict format,
    va_list ap);

int vdprintf(int fildes, const char *restrict format, va_list ap);

int vfprintf(FILE *restrict stream, const char *restrict format,
    va_list ap);

int vprintf(const char *restrict format, va_list ap);

int vsnprintf(char *restrict s, size_t n, const char *restrict format,
    va_list ap);

int vsprintf(char *restrict s, const char *restrict format, va_list ap);
```

### DESCRIPTION

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `vasprintf()`, `vdprintf()`, `vfprintf()`, `vprintf()`, `vsnprintf()`, and `vsprintf()` functions shall be equivalent to the `asprintf()`, `dprintf()`, `fprintf()`, `printf()`, `snprintf()`, and `sprintf()` functions respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by `<stdarg.h>`.

These functions shall not invoke the `va_end` macro. As these functions invoke the `va_arg` macro, the value of `ap` after the return is unspecified.

### RETURN VALUE

Refer to `fprintf()`.

# ERRORS

Refer to `fprintf()`.

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

Applications using these functions should call `va_end(ap)` afterwards to clean up.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [2.5 Standard I/O Streams](#)
- [fprintf\(\)](#)
- XBD `<stdarg.h>`, `<stdio.h>`

## CHANGE HISTORY

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

### Issue 5

The `vsnprintf()` function is added.

## Issue 6

The `vfprintf()`, `vprintf()`, `vsnprintf()`, and `vsprintf()` functions are updated for alignment with the ISO/IEC 9899:1999 standard.

## Issue 7

The `vdprintf()` function is added to complement the `dprintf()` function from The Open Group Technical Standard, 2006, Extended API Set Part 1.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0703 [14] is applied.

## Issue 8

Austin Group Defect 1496 is applied, adding the `vasprintf()` function.

## 1.275. vfscanf, vscanf, vsscanf — format input of a stdarg argument list

---

### SYNOPSIS

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

int vfscanf(FILE *restrict stream, const char *restrict format, va_list arg);
int vscanf(const char *restrict format, va_list arg);
int vsscanf(const char *restrict s, const char *restrict format, va_list arg);
```

### DESCRIPTION

The `vscanf()`, `vfscanf()`, and `vsscanf()` functions shall be equivalent to the `scanf()`, `fscanf()`, and `sscanf()` functions, respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined in the `<stdarg.h>` header. These functions shall not invoke the `va_end` macro. As these functions invoke the `va_arg` macro, the value of `ap` after the return is unspecified.

### RETURN VALUE

Refer to `fscanf()`.

### ERRORS

Refer to `fscanf()`.

### EXAMPLES

None.

## APPLICATION USAGE

Applications using these functions should call `va_end(ap)` afterwards to clean up.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [2.5 Standard I/O Streams](#)
- [fscanf\(\)](#)
- XBD `<stdarg.h>`
- XBD `<stdio.h>`

## CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

### Issue 7

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0704 [14] is applied.

## 1.276. vsnprintf

---

### SYNOPSIS

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

/* XSI extension */
int vasprintf(char **restrict ptr, const char *restrict format,
              va_list ap);
int vdprintf(int fildes, const char *restrict format, va_list ap);

int vfprintf(FILE *restrict stream, const char *restrict format,
              va_list ap);
int vprintf(const char *restrict format, va_list ap);
int vsnprintf(char *restrict s, size_t n, const char *restrict format,
              va_list ap);
int vsprintf(char *restrict s, const char *restrict format,
              va_list ap);
```

### DESCRIPTION

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `vasprintf()`, `vdprintf()`, `vfprintf()`, `vprintf()`, `vsnprintf()`, and `vsprintf()` functions shall be equivalent to the `asprintf()`, `dprintf()`, `fprintf()`, `printf()`, `snprintf()`, and `sprintf()` functions respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by `<stdarg.h>`.

These functions shall not invoke the `va_end` macro. As these functions invoke the `va_arg` macro, the value of `ap` after the return is unspecified.

### RETURN VALUE

Refer to `fprintf()`.

# ERRORS

Refer to [fprintf\(\)](#).

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

Applications using these functions should call [va\\_end\(ap\)](#) afterwards to clean up.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- 2.5 Standard I/O Streams
- [fprintf\(\)](#)

XBD [<stdarg.h>](#), [<stdio.h>](#)

## CHANGE HISTORY

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

## Issue 5

The `vsprintf()` function is added.

## Issue 6

The `vfprintf()`, `vprintf()`, `vsnprintf()`, and `vsprintf()` functions are updated for alignment with the ISO/IEC 9899:1999 standard.

## Issue 7

The `vdprintf()` function is added to complement the `dprintf()` function from The Open Group Technical Standard, 2006, Extended API Set Part 1.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0703 [14] is applied.

## Issue 8

Austin Group Defect 1496 is applied, adding the `vasprintf()` function.

---

## 1.277. vsprintf - Format Output of a stdarg Argument List

### Synopsis

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

/* XSI extension */
int vasprintf(char **restrict ptr, const char *restrict format,
              va_list ap);
int vdprintf(int fildes, const char *restrict format, va_list ap);

int vfprintf(FILE *restrict stream, const char *restrict format,
              va_list ap);
int vprintf(const char *restrict format, va_list ap);
int vsnprintf(char *restrict s, size_t n, const char *restrict format,
              va_list ap);
int vsprintf(char *restrict s, const char *restrict format, va_list ap);
```

### Description

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `vasprintf()`, `vdprintf()`, `vfprintf()`, `vprintf()`, `vsnprintf()`, and `vsprintf()` functions shall be equivalent to the `asprintf()`, `dprintf()`, `fprintf()`, `printf()`, `snprintf()`, and `sprintf()` functions respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by `<stdarg.h>`.

These functions shall not invoke the `va_end` macro. As these functions invoke the `va_arg` macro, the value of `ap` after the return is unspecified.

### Return Value

Refer to `fprintf()`.

## Errors

Refer to `fprintf()`.

### 1.277.1. Examples

None.

### 1.277.2. Application Usage

Applications using these functions should call `va_end(ap)` afterwards to clean up.

### 1.277.3. Rationale

None.

### 1.277.4. Future Directions

None.

### 1.277.5. See Also

- [2.5 Standard I/O Streams](#)
- `fprintf()`
- `<stdarg.h>`
- `<stdio.h>`

# 1.277.6. Change History

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

## Issue 5

The `vsnprintf()` function is added.

## Issue 6

The `vfprintf()`, `vprintf()`, `vsnprintf()`, and `vsprintf()` functions are updated for alignment with the ISO/IEC 9899:1999 standard.

## Issue 7

The `vdprintf()` function is added to complement the `dprintf()` function from The Open Group Technical Standard, 2006, Extended API Set Part 1.

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0703 [14] is applied.

## Issue 8

Austin Group Defect 1496 is applied, adding the `vasprintf()` function.

---

## 1.278. vsscanf

---

### SYNOPSIS

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

int vfscanf(FILE *restrict stream,
             const char *restrict format,
             va_list arg);

int vscanf(const char *restrict format,
           va_list arg);

int vsscanf(const char *restrict s,
            const char *restrict format,
            va_list arg);
```

### DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2024 defers to the ISO C standard.

The `vscanf()`, `vfscanf()`, and `vsscanf()` functions shall be equivalent to the `scanf()`, `fscanf()`, and `sscanf()` functions, respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined in the `<stdarg.h>` header. These functions shall not invoke the `va_end` macro. As these functions invoke the `va_arg` macro, the value of `ap` after the return is unspecified.

### RETURN VALUE

Refer to `fscanf()`.

### ERRORS

Refer to `fscanf()`.

---

*The following sections are informative.*

## EXAMPLES

None.

## APPLICATION USAGE

Applications using these functions should call `va_end(ap)` afterwards to clean up.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

- [2.5 Standard I/O Streams](#)
- [fscanf\(\)](#)
- XBD `<stdarg.h>`, `<stdio.h>`

## CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

### Issue 7

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0704 [14] is applied.

## 1.279. write

---

### SYNOPSIS

```
#include <unistd.h>

ssize_t pwrite(int fildes, const void *buf, size_t nbytes, off_t off);
ssize_t write(int fildes, const void *buf, size_t nbytes);
```

### DESCRIPTION

The `write()` function shall attempt to write `nbyte` bytes from the buffer pointed to by `buf` to the file associated with the open file descriptor, `fildes`.

Before any action described below is taken, and if `nbyte` is zero and the file is a regular file, the `write()` function may detect and return errors as described below. In the absence of errors, or if error detection is not performed, the `write()` function shall return zero and have no other results. If `nbyte` is zero and the file is not a regular file, the results are unspecified.

On a regular file or other file capable of seeking, the actual writing of data shall proceed from the position in the file indicated by the file offset associated with `fildes`. Before successful return from `write()`, the file offset shall be incremented by the number of bytes actually written. On a regular file, if the position of the last byte written is greater than or equal to the length of the file, the length of the file shall be set to this position plus one.

On a file not capable of seeking, writing shall always take place starting at the current position. The value of a file offset associated with such a device is undefined.

If the `O_APPEND` flag of the file status flags is set, the file offset shall be set to the end of the file prior to each write and no intervening file modification operation shall occur between changing the file offset and the write operation.

If a `write()` requests that more bytes be written than there is room for (for example, the file size limit of the process or the physical end of a medium), only as many bytes as there is room for shall be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write of 512 bytes will return 20. The next write of a non-zero number of bytes would give a failure return (except as noted below).

If the request would cause the file size to exceed the soft file size limit for the process and there is no room for any bytes to be written, the request shall fail [XSI] and the implementation shall generate a SIGXFSZ signal for the thread.

If `write()` is interrupted by a signal before it writes any data, it shall return -1 with `errno` set to [EINTR].

If `write()` is interrupted by a signal after it successfully writes some data, it shall return the number of bytes written.

If the value of `nbyte` is greater than {SSIZE\_MAX}, the result is implementation-defined.

After a `write()` to a regular file has successfully returned:

- Any successful `read()` from each byte position in the file that was modified by that write shall return the data specified by the `write()` for that position until such byte positions are again modified.
- Any subsequent successful `write()` to the same byte position in the file shall overwrite that file data.

Write requests to a pipe or FIFO shall be handled in the same way as a regular file with the following exceptions:

- There is no file offset associated with a pipe or FIFO, hence each write request shall append to the end of the pipe or FIFO.
- Write requests of {PIPE\_BUF} bytes or less shall not be interleaved with data from other threads performing write operations on the same pipe or FIFO. Writes of greater than {PIPE\_BUF} bytes may have data interleaved, on arbitrary boundaries, with write operations by other threads, whether or not the O\_NONBLOCK flag of the file status flags is set.
- If the O\_NONBLOCK flag is clear, a write request may cause the thread to block, but on normal completion it shall return `nbyte`.
- If the O\_NONBLOCK flag is set, `write()` requests shall be handled differently, in the following ways:
  - The `write()` function shall not block the thread.
  - A write request for {PIPE\_BUF} or fewer bytes shall have the following effect: if there is sufficient space available in the pipe or FIFO, `write()` shall transfer all the data and return the number of bytes requested. Otherwise, `write()` shall transfer no data and return -1 with `errno` set to [EAGAIN].
  - A write request for more than {PIPE\_BUF} bytes shall cause one of the following:

- When at least one byte can be written, transfer what it can and return the number of bytes written. When all data previously written to the pipe or FIFO is read, it shall transfer at least {PIPE\_BUF} bytes.
- When no data can be written, transfer no data, and return -1 with `errno` set to [EAGAIN].

When attempting to write to a file descriptor (other than a pipe or FIFO) that supports non-blocking writes and cannot accept the data immediately:

- If the O\_NONBLOCK flag is clear, `write()` shall block the calling thread until the data can be accepted.
- If the O\_NONBLOCK flag is set, `write()` shall not block the thread. If some data can be written without blocking the thread, `write()` shall write what it can and return the number of bytes written. Otherwise, it shall return -1 and set `errno` to [EAGAIN].

Upon successful completion, where `nbyte` is greater than 0, `write()` shall mark for update the last data modification and last file status change timestamps 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.

For regular files, no data transfer shall occur past the offset maximum established in the open file description associated with `fildes`.

If `fildes` refers to a socket, `write()` shall be equivalent to `send()` with no flags set.

[SIO] If the O\_DSYNC bit has been set, write I/O operations on the file descriptor shall complete as defined by synchronized I/O data integrity completion.

If the O\_SYNC bit has been set, write I/O operations on the file descriptor shall complete as defined by synchronized I/O file integrity completion.

[SHM] If `fildes` refers to a shared memory object, the result of the `write()` function is unspecified.

[TYM] If `fildes` refers to a typed memory object, the result of the `write()` function is unspecified.

The `pwrite()` function shall be equivalent to `write()`, except that it writes into a given position and does not change the file offset (regardless of whether O\_APPEND is set). The first three arguments to `pwrite()` are the same as `write()` with the addition of a fourth argument `offset` for the desired position inside the file. An attempt to perform a `pwrite()` on a file that is incapable of seeking shall result in an error.

## RETURN VALUE

Upon successful completion, these functions shall return the number of bytes actually written to the file associated with `fildes`. This number shall never be greater than `nbyte`. Otherwise, -1 shall be returned and `errno` set to indicate the error.

## ERRORS

These functions shall fail if:

### [EAGAIN]

The file is neither a pipe, nor a FIFO, nor a socket, the `O_NONBLOCK` flag is set for the file descriptor, and the thread would be delayed in the `write()` operation.

### [EBADF]

The `fildes` argument is not a valid file descriptor open for writing.

### [EFBIG]

An attempt was made to write a file that exceeds the implementation-defined maximum file size and there was no room for any bytes to be written.

### [EFBIG]

An attempt was made to write a file that exceeds the file size limit of the process, and there was no room for any bytes to be written. [XSI] A `SIGXFSZ` signal shall also be generated for the thread.

### [EFBIG]

The file is a regular file, `nbyte` 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 `fildes`.

### [EINTR]

The write operation was terminated due to the receipt of a signal, and no data was transferred.

### [EIO]

The process is a member of a background process group attempting to write to its controlling terminal, `TOSTOP` is set, the calling thread is not blocking `SIGTTOU`, the process is not ignoring `SIGTTOU`, and the process group of the process is orphaned. This error may also be returned under implementation-defined conditions.

### [ENOSPC]

There was no free space remaining on the device containing the file.

The `pwrite()` function shall fail if:

## **[EINVAL]**

The file is a regular file or block special file, and the `offset` argument is negative. The file offset shall remain unchanged.

## **[ESPIPE]**

The file is incapable of seeking.

The `write()` function shall fail if:

## **[EAGAIN]**

The file is a pipe or FIFO, the `O_NONBLOCK` flag is set for the file descriptor, and the thread would be delayed in the write operation.

## **[EAGAIN] or [EWOULDBLOCK]**

The file is a socket, the `O_NONBLOCK` flag is set for the file descriptor, and the thread would be delayed in the write operation.

## **[ECONNRESET]**

A write was attempted on a socket that is not connected.

## **[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 shall also be sent to the thread.

## **[EPIPE]**

A write was attempted on a socket that is shut down for writing, or is no longer connected. In the latter case, if the socket is of type `SOCK_STREAM`, a `SIGPIPE` signal shall also be sent to the thread.

These functions may fail if:

## **[EIO]**

A physical I/O error has occurred.

## **[ENOBUFS]**

Insufficient resources were available in the system to perform the operation.

## **[ENXIO]**

A request was made of a nonexistent device, or the request was outside the capabilities of the device.

The `write()` function may fail if:

## **[EACCES]**

A write was attempted on a socket and the calling process does not have appropriate privileges.

## **[ENETDOWN]**

A write was attempted on a socket and the local network interface used to reach the destination is down.

## [ENETUNREACH]

A write was attempted on a socket and no route to the network is present.

---

*The following sections are informative.*

## EXAMPLES

### Writing from a Buffer

The following example writes data from the buffer pointed to by `buf` to the file associated with the file descriptor `fd`.

```
#include <sys/types.h>
#include <string.h>
...
char buf[20];
size_t nbytes;
ssize_t bytes_written;
int fd;
...
strcpy(buf, "This is a test\n");
nbytes = strlen(buf);

bytes_written = write(fd, buf, nbytes);
...
```

## APPLICATION USAGE

None.

## RATIONALE

See also the RATIONALE section in `read()`.

An attempt to write to a pipe or FIFO has several major characteristics:

- **Atomic/non-atomic:** A write is atomic if the whole amount written in one operation is not interleaved with data from any other thread. This is useful when there are multiple writers sending data to a single reader. Applications need to know how large a write request can be expected to be performed atomically. This maximum is called {PIPE\_BUF}. This volume of POSIX.1-

2024 does not say whether write requests for more than {PIPE\_BUF} bytes are atomic, but requires that writes of {PIPE\_BUF} or fewer bytes shall be atomic.

- **Blocking/immediate:** Blocking is only possible with O\_NONBLOCK clear. If there is enough space for all the data requested to be written immediately, the implementation should do so. Otherwise, the calling thread may block; that is, pause until enough space is available for writing. The effective size of a pipe or FIFO (the maximum amount that can be written in one operation without blocking) may vary dynamically, depending on the implementation, so it is not possible to specify a fixed value for it.

- **Complete/partial/deferred:** A write request:

```
int fildes;
size_t nbyte;
ssize_t ret;
char *buf;

ret = write(fildes, buf, nbyte);
```

may return:

\*\*Complete\*\*  
`ret` = `nbyte`

\*\*Partial\*\*  
`ret` < `nbyte`

This shall never happen if `nbyte` ≤ {PIPE\_BUF}. If it does happen

\*\*Deferred\*\*  
`ret` = -1, `errno` = [EAGAIN]

This error indicates that a later request may succeed. It does not

Partial and deferred writes are only possible with O\_NONBLOCK set.



The relations of these properties are shown in the following tables:

**Write to a Pipe or FIFO with O\_NONBLOCK clear**

Immediately Writable:	None	Some	nbyte
<b>nbyte</b> ≤	Atomic blocking nbyte	Atomic blocking nbyte	Atomic immediate nbyte
<b>nbyte</b> >	Blocking nbyte	Blocking nbyte	Blocking nbyte

If the `O_NONBLOCK` flag is clear, a write request shall block if the amount writable immediately is less than that requested. If the flag is set (by `fcntl()`), a write request shall never block.

#### Write to a Pipe or FIFO with `O_NONBLOCK` set

Immediately Writable:	None	Some	nbyte
<b>nbyte</b> ≤	-1, [EAGAIN]	-1, [EAGAIN]	Atomic nbyte
<b>nbyte</b> >	-1, [EAGAIN]	< nbyte or -1, [EAGAIN]	≤ nbyte or -1, [EAGAIN]

There is no exception regarding partial writes when `O_NONBLOCK` is set. With the exception of writing to an empty pipe or FIFO, this volume of POSIX.1-2024 does not specify exactly when a partial write is performed since that would require specifying internal details of the implementation. Every application should be prepared to handle partial writes when `O_NONBLOCK` is set and the requested amount is greater than `{PIPE_BUF}`, just as every application should be prepared to handle partial writes on other kinds of file descriptors.

The intent of forcing writing at least one byte if any can be written is to assure that each write makes progress if there is any room in the pipe or FIFO. If the pipe or FIFO is empty, `{PIPE_BUF}` bytes must be written; if not, at least some progress must have been made.

Where this volume of POSIX.1-2024 requires -1 to be returned and `errno` set to [EAGAIN], most historical implementations return zero (with the `O_NDELAY` flag set, which is the historical predecessor of `O_NONBLOCK`, but is not itself in this volume of POSIX.1-2024). The error indications in this volume of POSIX.1-2024 were chosen so that an application can distinguish these cases from end-of-file. While `write()` cannot receive an indication of end-of-file, `read()` can, and the two functions have similar return values. Also, some existing systems (for example, Eighth Edition) permit a write of zero bytes to mean that the reader should get an

end-of-file indication; for those systems, a return value of zero from `write()` indicates a successful write of an end-of-file indication.

Implementations are allowed, but not required, to perform error checking for `write()` requests of zero bytes.

The concept of a {PIPE\_MAX} limit (indicating the maximum number of bytes that can be written to a pipe or FIFO in a single operation) was considered, but rejected, because this concept would unnecessarily limit application writing.

See also the discussion of O\_NONBLOCK in `read()`.

Writes can be serialized with respect to other reads and writes. If a `read()` of file data can be proven (by any means) to occur after a `write()` of the data, it must reflect that `write()`, even if the calls are made by different threads. A similar requirement applies to multiple write operations to the same file position. This is needed to guarantee the propagation of data from `write()` calls to subsequent `read()` calls. This requirement is particularly significant for networked file systems, where some caching schemes violate these semantics.

Note that this is specified in terms of `read()` and `write()`. The XSI extensions `readv()` and `writev()` also obey these semantics. A new "high-performance" write analog that did not follow these serialization requirements would also be permitted by this wording. This volume of POSIX.1-2024 is also silent about any effects of application-level caching (such as that done by stdio).

This volume of POSIX.1-2024 does not specify the value of the file offset after an error is returned; there are too many cases. For programming errors, such as [EBADF], the concept is meaningless since no file is involved. For errors that are detected immediately, such as [EAGAIN], clearly the pointer should not change. After an interrupt or hardware error, however, an updated value would be very useful and is the behavior of many implementations.

This volume of POSIX.1-2024 does not specify the behavior of concurrent writes to a regular file from multiple threads, except that each write is atomic (see 2.9.7 Thread Interactions with File Operations). Applications should use some form of concurrency control.

This volume of POSIX.1-2024 intentionally does not specify any `pwrite()` errors related to pipes, FIFOs, and sockets other than [ESPIPE].

## FUTURE DIRECTIONS

None.

## SEE ALSO

`chmod()`, `creat()`, `dup()`, `fcntl()`, `getrlimit()`, `lseek()`,  
`open()`, `pipe()`, `read()`, `writev()`

XBD `<limits.h>`, `<sys/uio.h>`, `<unistd.h>`

## CHANGE HISTORY

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

### Issue 5

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

Large File Summit extensions are added.

The `pwrite()` function is added.

### Issue 6

The DESCRIPTION states that the `write()` function does not block the thread. Previously this said "process" rather than "thread".

The DESCRIPTION and ERRORS sections are updated so that references to STREAMS are marked as part of the XSI STREAMS Option Group.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION now states that if `write()` is interrupted by a signal after it has successfully written some data, it returns the number of bytes written. In the POSIX.1-1988 standard, it was optional whether `write()` returned the number of bytes written, or whether it returned -1 with `errno` set to [EINTR]. This is a FIPS requirement.
- The following changes are made to support large files:
  - For regular files, no data transfer occurs past the offset maximum established in the open file description associated with the `fd`.
  - A second [EFBIG] error condition is added.
- The [EIO] error condition is added.
- The [EPIPE] error condition is added for when a pipe has only one end open.

- The [ENXIO] optional error condition is added.

Text referring to sockets is added to the DESCRIPTION.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The effect of reading zero bytes is clarified.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that `write()` results are unspecified for typed memory objects.

The following error conditions are added for operations on sockets: [EAGAIN], [EWOULDBLOCK], [ECONNRESET], [ENOTCONN], and [EPIPE].

The [EIO] error is made optional.

The [ENOBUFS] error is added for sockets.

The following error conditions are added for operations on sockets: [EACCES], [ENETDOWN], and [ENETUNREACH].

The `writev()` function is split out into a separate reference page.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/146 is applied, updating text in the ERRORS section from "a SIGPIPE signal is generated to the calling process" to "a SIGPIPE signal shall also be sent to the thread".

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/147 is applied, making a correction to the RATIONALE.

## Issue 7

The `pwrite()` function is moved from the XSI option to the Base.

Functionality relating to the XSI STREAMS option is marked obsolescent.

SD5-XSH-ERN-160 is applied, updating the DESCRIPTION to clarify the requirements for the `pwrite()` function, and to change the use of the phrase "file pointer" to "file offset".

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0742 [219], XSH/TC1-2008/0743 [215], XSH/TC1-2008/0744 [79], and XSH/TC1-2008/0745 [215] are applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0401 [676,710] and XSH/TC2-2008/0402 [966] are applied.

## Issue 8

Austin Group Defect 308 is applied, clarifying the handling of [EFBIG] errors.

Austin Group Defect 1330 is applied, removing obsolescent interfaces.

Austin Group Defect 1430 is applied, clarifying that requirements relating to data interleaving on pipes and FIFOs apply to write operations in other threads, not just other processes, and changing some uses of "pipe" to "pipe or FIFO".

Austin Group Defect 1669 is applied, removing XSI shading from part of the [EFBIG] error relating to the file size limit for the process.

---