ChorusOS man pages section 3POSIX: POSIX Library Functions

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900
U.S.A.

Part No: 806-3333
December 10, 1999

Contents

PREFACE

Overview

A man page is provided for both the naive user, and sophisticated user who is familiar with the ChorusOS™ operating system and is in need of on-line information. A man page is intended to answer concisely the question “What does it do?” The man pages in general comprise a reference manual. They are not intended to be a tutorial.

The following is a list of sections in the ChorusOS man pages and the information it references:

- *Section 1CC: User Utilities; Host and Target Utilities*
- *Section 1M: System Management Utilities*
- *Section 2DL: System Calls; Data Link Services*
- *Section 2K: System Calls; Kernel Services*
- *Section 2MON: System Calls; Monitoring Services*
- *Section 2POSIX: System Calls; POSIX System Calls*
- *Section 2RESTART: System Calls; Hot Restart and Persistent Memory*
- *Section 2SEG: System Calls; Virtual Memory Segment Services*
- *Section 3FTPD: Libraries; FTP Daemon*
- *Section 3M: Libraries; Mathematical Libraries*
- *Section 3POSIX: Libraries; POSIX Library Functions*
- *Section 3RPC: Libraries; RPC Services*
- *Section 3STDC: Libraries; Standard C Library Functions*
- *Section 3TELD: Libraries; Telnet Services*
- *Section 4CC: Files*

- *Section 5FEA: ChorusOS Features and APIs*
- *Section 7P: Protocols*
- *Section 7S: Services*
- *Section 9DDI: Device Driver Interfaces*
- *Section 9DKI: Driver to Kernel Interface*
- *Section 9DRV: Driver Implementations*

ChorusOS man pages are grouped in Reference Manuals, with one reference manual per section.

Below is a generic format for man pages. The man pages of each manual section generally follow this order, but include only needed headings. For example, if there are no bugs to report, there is no BUGS section. See the `intro` pages for more information and detail about each section, and `man`(1) for more information about man pages in general.

NAME	This section gives the names of the commands or functions documented, followed by a brief description of what they do.
SYNOPSIS	This section shows the syntax of commands or functions. When a command or file does not exist in the standard path, its full pathname is shown. Options and arguments are alphabetized, with single letter arguments first, and options with arguments next, unless a different argument order is required.

The following special characters are used in this section:

[ ]	The option or argument enclosed in these brackets is optional. If the brackets are omitted, the argument must be specified.
. . .	Ellipses. Several values may be provided for the previous argument, or the previous argument can be specified multiple times, for example, ‘ "`filename` . . ." .
|	Separator. Only one of the arguments separated by this character can be specified at time.
{ }	Braces. The options and/or arguments enclosed within braces are

	interdependent, such that everything enclosed must be treated as a unit.
FEATURES	This section provides the list of features which offer an interface. An API may be associated with one or more system features. The interface will be available if one of the associated features has been configured.
DESCRIPTION	This section defines the functionality and behavior of the service. Thus it describes concisely what the command does. It does not discuss OPTIONS or cite EXAMPLES.. Interactive commands, subcommands, requests, macros, functions and such, are described under USAGE.
OPTIONS	This lists the command options with a concise summary of what each option does. The options are listed literally and in the order they appear in the SYNOPSIS section. Possible arguments to options are discussed under the option, and where appropriate, default values are supplied.
OPERANDS	This section lists the command operands and describes how they affect the actions of the command.
OUTPUT	This section describes the output - standard output, standard error, or output files - generated by the command.
RETURN VALUES	If the man page documents functions that return values, this section lists these values and describes the conditions under which they are returned. If a function can return only constant values, such as 0 or –1, these values are listed in tagged paragraphs. Otherwise, a single paragraph describes the return values of each function. Functions declared void do not return values, so they are not discussed in RETURN VALUES.
ERRORS	On failure, most functions place an error code in the global variable `errno` indicating why they failed. This section lists alphabetically all error codes a function can generate and describes the conditions that cause each error. When more than one condition can cause the same error, each condition is described in a separate paragraph under the error code.

USAGE	This section is provided as a guidance on use. This section lists special rules, features and commands that require in-depth explanations. The subsections listed below are used to explain built-in functionality: Commands Modifiers Variables Expressions Input Grammar
EXAMPLES	This section provides examples of usage or of how to use a command or function. Wherever possible a complete example including command line entry and machine response is shown. Whenever an example is given, the prompt is shown as `example%` or if the user must be superuser, `example#`. Examples are followed by explanations, variable substitution rules, or returned values. Most examples illustrate concepts from the SYNOPSIS, DESCRIPTION, OPTIONS and USAGE sections.
ENVIRONMENT VARIABLES	This section lists any environment variables that the command or function affects, followed by a brief description of the effect.
EXIT STATUS	This section lists the values the command returns to the calling program or shell and the conditions that cause these values to be returned. Usually, zero is returned for successful completion and values other than zero for various error conditions.
FILES	This section lists all filenames referred to by the man page, files of interest, and files created or required by commands. Each is followed by a descriptive summary or explanation.
SEE ALSO	This section lists references to other man pages, in-house documentation and outside publications.
DIAGNOSTICS	This section lists diagnostic messages with a brief explanation of the condition causing the error.
WARNINGS	This section lists warnings about special conditions which could seriously affect your working conditions. This is not a list of diagnostics.
NOTES	This section lists additional information that does not belong anywhere else on the page. It takes the form of an aside to the user, covering points of special interest. Critical information is never covered here.

BUGS

This section describes known bugs and wherever possible, suggests workarounds.

CHAPTER

POSIX Library Functions

NAME

Intro – introduction to POSIX-compliant pthread and timer calls

DESCRIPTION

This section describes the API for the POSIX threads and timers in ChorusOS (see `intro`(2K)).

The POSIX threads provide a limited POSIX-compliant programming and execution environment including thread management and thread synchronization functions. It is complemented by the POSIX timers, which adds POSIX real-time clock and timer features. POSIX-THREADS is based on the draft standard P1003.1c (Threads Extension) Draft 8, POSIX-TIMERS on IEEE Std 1003.1b-1993.

NUMERICAL LIMITS

Symbols corresponding to POSIX numerical limits are defined in <*limits.h*>. However if a definition of one of the values defined by POSIX is omitted from <*limits.h*>, its actual value is provided by the *sysconf*(3POSIX) function.

SYMBOLIC CONSTANTS

As different POSIX profiles may be supported by the same set of header files, the POSIX-THREADS / POSIX-TIMERS option symbols are defined in <*unistd.h*>. An application may always choose to interrogate a value at run-time to take advantage of the current configuration using the *sysconf*(3POSIX) function.

The following POSIX option symbols are defined in the context of these features. For POSIX-THREADS:

```
_POSIX_SEMAPHORES
_POSIX_THREADS
_POSIX_THREAD_SAFE_FUNCTIONS
_POSIX_THREAD_PRIORITY_SCHEDULING
_POSIX_THREAD_ATTR_STACKSIZE
_POSIX_THREAD_ATTR_STACKADDR
```

For POSIX-TIMERS:

```
_POSIX_TIMERS
```

However, some features in the POSIX standards associated with these option symbols are not supported in ChorusOS. In other cases, the semantics provided are slightly different, as follows:

Thread cancellation
No interfaces related to thread cancellation are provided.

Priority queueing on synchronization objects
Threads are awakened by *sem_post*, *pthread_mutex_unlock*, and *pthread_cond_signal* in the order in which they had blocked on the respective synchronization object, rather than in priority order.

Resource limits
The symbols _SC_PTHREAD_THREADS_MAX and _SC_PTHREAD_KEYS_MAX respectively, define limits on the number of

threads and the number of thread-specific data keys per system rather than per process (actor) as defined in POSIX.

Supervisor thread stacks
Most functions are identical in user and supervisor actors. However, an application running in supervisor mode may not create a thread with a pre-allocated stack.

In all other areas, an attempt has been made to conform precisely to the specifications of the two POSIX standards.

A POSIX application runs as a Chorus actor. The POSIX interfaces described in this manual are seen as extensions to the standard ChorusOS programming environment. In several areas, ChorusOS and POSIX semantics interact.

CHORUS-POSIX INTERACTIONS: SCHEDULING

The SCHED_FIFO and SCHED_RR scheduling policies are provided as specified in POSIX 1003.1b. (A third policy, SCHED_OTHER, is identical to SCHED_RR.) The SCHED_FIFO policy provides strict preemptive scheduling within a range of priorities that may be obtained using *sched_get_priority_min* and *sched_get_priority_max*. Except when thread priorities are dynamically modified, a running thread continues executing until it blocks voluntarily or is preempted by a thread of higher priority, in which case it is queued at the head of the list of runnable threads corresponding to its priority level. The SCHED_RR policy is the same except that a thread that executes longer than a pre-defined quantum (obtainable using *sched_rr_get_interval*) may be descheduled and queued behind any other runnable threads waiting to execute at the same priority.

The POSIX threads are mapped one-to-one onto ChorusOS threads: the POSIX SCHED_FIFO and SCHED_RR scheduling policies are mapped onto the K_SCHED_FIFO and K_SCHED_RR scheduling classes, respectively. The priority mapping is defined below. Note that the priority systems scales are reversed:

- In the ChorusOS scheduling classes, a larger numeric priority value indicates a lower thread priority.
- Under the SCHED_FIFO and SCHED_RR policies, a higher priority value indicates a higher thread priority.

In the following, all ranges are shown in order of increasing thread priority. Symbols are defined in `<sched.h>`.

SCHED_FIFO	POSIX priorities in the range POSIX_FIFO_MIN (0) to POSIX_FIFO_MAX (255) are mapped one-to-one onto the CHORUS K_SCHED_FIFO class priority range K_FIFO_PRIOMIN (255) to K_FIFO_PRIOMAX (0).

SCHED_RR	POSIX priorities in the range POSIX_RR_MIN (0) to POSIX_RR_MAX (255) are mapped one-to-one onto the CHORUS K_SCHED_RR class priority range K_RR_PRIOMON (255) to K_RR_PRIOMAX (0).

The default policy and priority are determined as follows. The first invocation of a substantive function of either the POSIX-THREADS or the POSIX-TIMERS (if included) features triggers initialization of both features. (Calls that create or manipulate threads, thread attribute objects, or timers are considered substantive for this purpose.) At that point the calling thread, which is normally the initial thread of the actor, is transformed into a POSIX thread (`pthread`), as if it had been created using y *pthread_create* in the state PTHREAD_CREATE_DETACHED (see *pthread_create*(3POSIX), *pthread_attr_init*(3POSIX)). Its ChorusOS thread priority is obtained (see *threadScheduler*(2K)) and mapped into a POSIX policy and priority using the reverse of the mappings defined above. These policy and priority values become the defaults for threads created in PTHREAD_EXPLICIT_SCHED (see *pthread_attr_init*(3POSIX)). Note that this procedure can only operate correctly if the calling thread is in the ChorusOS K_SCHED_FIFO or K_SCHED_RR class. If the calling thread is in a different CHORUS scheduling class, then the default policy is set to SCHED_RR and the default priority is set to POSIX_RR_MIN.

All threads are scheduled on a system-wide basis, according to their priority (and scheduling class). Hence the POSIX option PTHREAD_SCOPE_PROCESS is not supported (see *pthread_attr_setscope*(3POSIX)).

CHORUS-POSIX INTERACTIONS: THREAD MANAGEMENT

POSIX applications are expected to use POSIX facilities for thread management wherever possible, rather than the corresponding ChorusOS facilities. However, some applications may involve threads which are not pthreads (not created by *pthread_create*), such as message handlers in supervisor actors. Furthermore, certain ChorusOS thread management services with no POSIX analogues may be useful in some applications.

When POSIX threads are created using the facilities in this manual, the POSIX thread identifier type *pthread_t* is equivalent to the ChorusOS thread local identifier. That is, the value returned by the *thread* argument of *pthread_create* is always equal to the *threadli* of the underlying ChorusOS thread.

As mentioned earlier, the first invocation of POSIX-THREADS or POSIX-TIMERS transforms the calling thread into a pthread. Other threads in the actor are not pthreads unless created by *pthread_create.* However, many POSIX services may be invoked by pure ChorusOS threads. In particular, POSIX mutexes, condition variables, and semaphores may be freely used for synchronization between and among POSIX threads and pure CHORUS threads. Certain POSIX functions have limitations regarding invocation from pure CHORUS threads.

pthread_create	May be used to create a POSIX thread; the PTHREAD_EXPLICIT_SCHED attribute is assumed. PTHREAD_INHERIT_SCHED is ignored if specified
pthread_exit	Deletes the calling thread, ignoring the status argument. (Non-pthreads cannot be joined using *pthread_join*)
timer_create	Requires that the caller be a POSIX thread; otherwise an error (ENOSYS) is returned

The following CHORUS Core Executive system calls may be useful in manipulating POSIX threads.

threadAbort and *threadAborted*
threadContext
threadName
threadStat

If a pthread is aborted while blocked or executing in certain POSIX functions, the result is the same as that defined in POSIX on arrival of a caught signal. In particular, *pthread_join*, *sem_wait*, *sem_trywait*, and *nanosleep* return EINTR and may take other actions in the event of a thread abort (see the corresponding manual pages for details). Otherwise, no support for thread abort is provided at the POSIX level.

INTERFACE TYPE DEFINITIONS

The following interface object types are used in the definition of the various POSIX functions. Formal definitions are found in <pthread.h>, <time.h>, <sched.h>, or <signal.h>.

pthread_t	POSIX thread identifier (equal to the corresponding *threadli*)
pthread_attr_t	POSIX thread attribute object (see *pthread_attr_init*(3POSIX))
pthread_mutex_t	Mutex, or blocking binary semaphore (see *pthread_mutex_init*(3POSIX))
pthread_mutexattr_t	Mutex attribute object; no attributes currently defined (see *pthread_mutexattr_init*(3POSIX))
pthread_cond_t	Condition variable (see *pthread_cond_init*(3POSIX))

pthread_condattr_t	Condition variable attribute object; no attributes currently defined (see *pthread_condattr_init*(3POSIX))
pthread_key_t	Key for lookup of per-thread data within actor (see *pthread_key_create(3POSIX))*
sem_t	Counting semaphore (see *sem_init*(3POSIX))
*void **	(standard C type) used for passing an arbitrary untyped value (not necessarily a pointer) in *pthread_exit*, *pthread_join*, *pthread_setspecific*, and *pthread_getspecific*
pthread_once_t	Variable used to record the initialization of a dynamic library (see *pthread_once*(3POSIX))
struct sched_param	Scheduling parameter structure, defined for each scheduling policy. For the policies currently supported, there is only one member, *sched_priority*.
clockid_t	Clock identifier. Only one clock, the system realtime clock, is supported. (see *clock_settime*(3POSIX))
struct sigevent	Structure defining action to take on asynchronous event notification. Contains a notification routine address and a single argument (see *timer_create*(3POSIX))
timer_t	Identifier of one-shot or periodic timer (see *timer_create*(3POSIX))
struct timespec	Standard time interval specification, in terms of seconds and additional nanoseconds.
struct itimerspec	Timer setting. Contains one *struct timespec* defining the time interval or absolute time, and another specifying the reload interval for a periodic timer.
size_t	Object size in bytes.

ERROR CODES

`WARNING:` error reporting conventions are inconsistent by design in the POSIX-THREADS and POSIX-TIMERS features (see `intro`(2K)). Error returns from POSIX-compliant interfaces follow the corresponding POSIX standards precisely, in which two different styles of return code are used:

1003.1b functions	(those beginning with *sem_*, *sched_*, *clock_*, and *timer_*; and *nanosleep*) return –1 in case of error and store the error code in *errno*.
1003.1c functions	(those beginning with *pthread_*) return the error code directly and do not set *errno*.

In addition, error codes are used differently in some cases. For example, an unsuccessful *sem_trywait* (1003.1b) returns EAGAIN, whereas an unsuccessful *pthread_mutex_try* (1003.1c) returns EBUSY. Invalid virtual addresses are flagged with EFAULT in 1003.1b functions and EINVAL in 1003.1c functions. Individual man pages contain information about the error codes returned by each routine.

The EFAULT error code will be returned where possible. However the library implementation of the POSIX interface accesses pointer arguments. Hence bad virtual addresses can cause a segmentation fault, instead of returning EFAULT error code.

1 EPERM Non-recoverable error
Attempted to perform a restricted operation for which the caller does not have the privilege.

3 ESRCH Object not found
A *pthread* call designated a non-existent target thread, or a thread which is not a pthread.

4 EINTR Interrupted function
The thread was aborted while blocked or executing in the function.

11 EAGAIN Resource temporarily exhausted
Some object or resource was unavailable, but the call may succeed if retried.

12 ENOMEM Insufficient memory available
Memory was unavailable, but the call may succeed if retried.

14 EFAULT Bad virtual address
A pointer argument contained an unmapped or inaccessible virtual address (user mode only).

16 EBUSY Object in use
Some object or resource was in use by another thread.

22 EINVAL Invalid argument
An argument was invalid, or a virtual address was unmapped or inaccessible (user mode only).

33 EDOM Math arg out of domain of function

34 ERANGE Math result not representable

45 EDEADLK Deadlock condition
A thread attempted to join (*pthread_join*) with itself.

80 ENOSYS Function not implemented
The function is not provided in the mode of the current actor.

108 ETIMEDOUT Timeout
The specified interval expired before the function completed.

133 ENOTSUP Option not supported
The requested option is not supported.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

Name	Description
`btree`(3POSIX)	btree database access method
`cfgetispeed`(3POSIX)	See `tcsetattr`(3POSIX)
`cfgetospeed`(3POSIX)	See `tcsetattr`(3POSIX)
`cfmakeraw`(3POSIX)	See `tcsetattr`(3POSIX)
`cfsetispeed`(3POSIX)	See `tcsetattr`(3POSIX)
`cfsetospeed`(3POSIX)	See `tcsetattr`(3POSIX)
`cfsetspeed`(3POSIX)	See `tcsetattr`(3POSIX)
`clock_getres`(3POSIX)	See `clock_settime`(3POSIX)
`clock_gettime`(3POSIX)	See `clock_settime`(3POSIX)
`clock_settime`(3POSIX)	get or set clock to specified value, or get clock resolution
`closedir`(3POSIX)	See `directory`(3POSIX)
`dbopen`(3POSIX)	database access methods
`directory`(3POSIX)	directory operations
`endnetent`(3POSIX)	See `getnetent`(3POSIX)
`endnetgrent`(3POSIX)	See `getnetgrent`(3POSIX)
`endprotoent`(3POSIX)	See `getprotoent`(3POSIX)

endservent(3POSIX)	See getservent(3POSIX)
err(3POSIX)	formatted error messages
getcwd(3POSIX)	get working directory pathname
getdiskbyname(3POSIX)	get generic disk description by its name
getmntinfo(3POSIX)	get information about mounted file systems
getnetbyaddr(3POSIX)	See getnetent(3POSIX)
getnetbyname(3POSIX)	See getnetent(3POSIX)
getnetent(3POSIX)	get network entry
getnetgrent(3POSIX)	netgroup database operations
getprotobyname(3POSIX)	See getprotoent(3POSIX)
getprotobynumber(3POSIX)	See getprotoent(3POSIX)
getprotoent(3POSIX)	get protocol entry
getservbyname(3POSIX)	See getservent(3POSIX)
getservbyport(3POSIX)	See getservent(3POSIX)
getservent(3POSIX)	get service entry
getwd(3POSIX)	See getcwd(3POSIX)
glob(3POSIX)	generate pathnames matching a pattern
globfree(3POSIX)	See glob(3POSIX)
hash(3POSIX)	hash database access method
innetgr(3POSIX)	See getnetgrent(3POSIX)
link_addr(3POSIX)	elementary address specification routines for link level access
link_ntoa(3POSIX)	See link_addr(3POSIX)
mpool(3POSIX)	shared memory buffer pool
nanosleep(3POSIX)	delay the current thread with high resolution
ns_addr(3POSIX)	Xerox NS address conversion routines

`ns_ntoa`(3POSIX)	See `ns_addr`(3POSIX)
`opendir`(3POSIX)	See `directory`(3POSIX)
`pthread_attr_destroy`(3POSIX)	See `pthread_attr_init`(3POSIX)
`pthread_attr_getdetachstate`(3POSIX)	See `pthread_attr_init`(3POSIX)
`pthread_attr_getinheritsched`(3POSIX)	See `pthread_attr_setscope`(3POSIX)
`pthread_attr_getschedparam`(3POSIX)	See `pthread_attr_setscope`(3POSIX)
`pthread_attr_getschedpolicy`(3POSIX)	See `pthread_attr_setscope`(3POSIX)
`pthread_attr_getscope`(3POSIX)	See `pthread_attr_setscope`(3POSIX)
`pthread_attr_getstackaddr`(3POSIX)	See `pthread_attr_init`(3POSIX)
`pthread_attr_getstacksize`(3POSIX)	See `pthread_attr_init`(3POSIX)
`pthread_attr_init`(3POSIX)	Initialize a thread attribute object; Destroy a thread attribute object; Set the stacksize attribute; Get the stacksize attribute; Set the stackaddr attribute; Get the stackaddr attribute; Set the detachstate attribute; Get the detachstate attribute
`pthread_attr_setdetachstate`(3POSIX)	See `pthread_attr_init`(3POSIX)
`pthread_attr_setinheritsched`(3POSIX)	See `pthread_attr_setscope`(3POSIX)
`pthread_attr_setschedparam`(3POSIX)	See `pthread_attr_setscope`(3POSIX)
`pthread_attr_setschedpolicy`(3POSIX)	See `pthread_attr_setscope`(3POSIX)
`pthread_attr_setscope`(3POSIX)	Set the contention scope attribute; Get the contention scope attribute; Set the scheduling inheritance attribute; Get the scheduling inheritance attribute;

	Set the scheduling policy attribute; Get the scheduling policy attribute; Set the scheduling parameter attribute; Get the scheduling parameter attribute
`pthread_attr_setstackaddr`(3POSIX)	See `pthread_attr_init`(3POSIX)
`pthread_attr_setstacksize`(3POSIX)	See `pthread_attr_init`(3POSIX)
`pthread_cond_broadcast`(3POSIX)	See `pthread_cond_init`(3POSIX)
`pthread_cond_destroy`(3POSIX)	See `pthread_cond_init`(3POSIX)
`pthread_cond_init`(3POSIX)	initialize and use a condition variable
`pthread_cond_signal`(3POSIX)	See `pthread_cond_init`(3POSIX)
`pthread_cond_timedwait`(3POSIX)	See `pthread_cond_init`(3POSIX)
`pthread_cond_wait`(3POSIX)	See `pthread_cond_init`(3POSIX)
`pthread_condattr_destroy`(3POSIX)	See `pthread_condattr_init`(3POSIX)
`pthread_condattr_init`(3POSIX)	initialize or destroy a condition variable attribute object
`pthread_create`(3POSIX)	create a thread
`pthread_equal`(3POSIX)	compare thread identifiers
`pthread_exit`(3POSIX)	terminate the calling thread
`pthread_getschedparam`(3POSIX)	See `pthread_setschedparam`(3POSIX)
`pthread_getspecific`(3POSIX)	See `pthread_setspecific`(3POSIX)
`pthread_join`(3POSIX)	wait for thread termination
`pthread_key_create`(3POSIX)	create or delete a thread-specific data key
`pthread_key_delete`(3POSIX)	See `pthread_key_create`(3POSIX)
`pthread_kill`(3POSIX)	send a deletion signal to a thread

pthread_mutex_destroy(3POSIX)	See pthread_mutex_init(3POSIX)
pthread_mutex_init(3POSIX)	initialize and use a mutex
pthread_mutex_lock(3POSIX)	See pthread_mutex_init(3POSIX)
pthread_mutex_trylock(3POSIX)	See pthread_mutex_init(3POSIX)
pthread_mutex_unlock(3POSIX)	See pthread_mutex_init(3POSIX)
pthread_mutexattr_destroy(3POSIX)	See pthread_mutexattr_init(3POSIX)
pthread_mutexattr_init(3POSIX)	initialize or destroy a mutex attribute object
pthread_once(3POSIX)	initialize a library dynamically
pthread_self(3POSIX)	get the identifier of the calling thread
pthread_setschedparam(3POSIX)	set or get current scheduling policy and parameters of a thread
pthread_setspecific(3POSIX)	set or get the thread-specific value associated with a key
pthread_yield(3POSIX)	yield processor to another thread
readdir(3POSIX)	See directory(3POSIX)
recno(3POSIX)	record number database access method
rewinddir(3POSIX)	See directory(3POSIX)
sched_get_priority_max(3POSIX)	get priority and time quantum information for scheduling policy
sched_get_priority_min(3POSIX)	See sched_get_priority_max(3POSIX)
sched_rr_get_interval(3POSIX)	See sched_get_priority_max(3POSIX)
sched_yield(3POSIX)	See pthread_yield(3POSIX)
seekdir(3POSIX)	See directory(3POSIX)
sem_destroy(3POSIX)	See sem_init(3POSIX)
sem_getvalue(3POSIX)	See sem_init(3POSIX)
sem_init(3POSIX)	initialize and use a semaphore

`sem_post`(3POSIX)	See `sem_init`(3POSIX)
`sem_trywait`(3POSIX)	See `sem_init`(3POSIX)
`sem_wait`(3POSIX)	See `sem_init`(3POSIX)
`setnetent`(3POSIX)	See `getnetent`(3POSIX)
`setnetgrent`(3POSIX)	See `getnetgrent`(3POSIX)
`setprotoent`(3POSIX)	See `getprotoent`(3POSIX)
`setservent`(3POSIX)	See `getservent`(3POSIX)
`sysconf`(3POSIX)	get configurable system variables
`sysctl`(3POSIX)	get or set system information
`sysctlbyname`(3POSIX)	See `sysctl`(3POSIX)
`tcgetattr`(3POSIX)	See `tcsetattr`(3POSIX)
`tcsetattr`(3POSIX)	manipulating the termios structure
`telldir`(3POSIX)	See `directory`(3POSIX)
`timer_create`(3POSIX)	create or delete a timer
`timer_delete`(3POSIX)	See `timer_create`(3POSIX)
`timer_getoverrun`(3POSIX)	See `timer_settime`(3POSIX)
`timer_gettime`(3POSIX)	See `timer_settime`(3POSIX)
`timer_settime`(3POSIX)	set and arm or disarm a timer, obtain remaining interval for an active timer, or obtain current overrun count for a timer
`verr`(3POSIX)	See `err`(3POSIX)
`verrx`(3POSIX)	See `err`(3POSIX)
`vwarn`(3POSIX)	See `err`(3POSIX)
`vwarnx`(3POSIX)	See `err`(3POSIX)
`warn`(3POSIX)	See `err`(3POSIX)
`warnx`(3POSIX)	See `err`(3POSIX)

NAME

btree – btree database access method

SYNOPSIS

#include <sys/types.h>

#include <db.h>

DESCRIPTION

`dbopen()` is the library interface to database files. One of the file formats supported is `btree` files. The general description of the database access methods is in `dbopen`(3POSIX). This manual page describes only the `btree`-specific information.

The `btree` data structure is a sorted, balanced tree structure storing associated key/data pairs.

The btree accesses a method-specific data structure provided to `dbopen()` by the `<db.h>` include file, whose structure is the following:

```
typedef struct {
     u_long  flags;
     u_int   cachesize;
     int     maxkeypage;
     int     minkeypage;
     u_int   psize;
     int     (*compare)(const DBT* key1, const DBT* key2);
     size_t  (*prefix)(const DBT* key1, const DBT* key2);
     int     lorder;
} BTREEINFO;
```

The elements of this structure are as follows:

flags The flag value is specified by *or*'ing any of the following values:

R_DUP permits duplicate keys in the tree, (in other words, allows insertion of the key even if it already exists in the tree).

The default behavior, as described in *dbopen*(3), is to overwrite a matching key when inserting a new key or to fail if the **R_NOOVERWRITE** flag is specified

The **R_DUP** flag is overridden by the **R_NOOVERWRITE** flag, and if the **R_NOOVERWRITE** flag is specified, attempts to insert duplicate keys into the tree will fail.

If the database contains duplicate keys, the order of retrieval of key/data pairs is undefined if the `get` routine is used. However, *seq* routine calls with the **R_CURSOR** flag set will always return the logical “first” of any group of duplicate keys.

cachesize A suggested maximum size (in bytes) of the memory cache. This value is `only` advisory, and the access method will allocate more memory rather than fail. As every search examines the root page of the tree, caching the most recently-used pages substantially improves access time. In addition, physical writes are delayed as long as

	possible, so a moderate cache can reduce the number of I/O operations significantly. Obviously, using a cache increases the likelihood of corruption or lost data if the system crashes while a tree is being modified. If *cachesize* is 0 (no size is specified) a default cache is used.
maxkeypage	The maximum number of keys which will be stored on any single page. Not currently implemented.
minkeypage	The minimum number of keys which will be stored on any single page. This value is used to determine which keys will be stored on overflow pages, (if a key or data item is longer than the page size divided by the minkeypage value, it will be stored on overflow pages instead of in the page itself). If *minkeypage* is 0 (no minimum number of keys is specified) a value of 2 is used.
psize	Page size is the size (in bytes) of the pages used for nodes in the tree. The minimum page size is 512 bytes and the maximum page size is 64K. If *psize* is 0 (no page size is specified) a page size is chosen based on the underlying file system I/O block size.
compare	This is the key comparison function. It must return an integer less than, equal to, or greater than zero if the first key argument is considered to be respectively less than, equal to, or greater than the second key argument. The same comparison function must be used on a given tree every time it is opened. If *compare* is NULL (no comparison function is specified), the keys are compared lexically, with shorter keys considered smaller than longer keys.
prefix	This is the prefix comparison function. If specified, this routine must return the number of bytes of the second key argument, which are needed to determine that it is larger than the first key argument. If the keys are equal, the key length should be returned. Note that the usefulness of this routine is very data-dependent, but in some data sets can produce significantly reduced tree sizes and search times. If *prefix* is NULL (no prefix function is specified), and no comparison function is specified, a default lexical comparison routine is used. If *prefix* is NULL and a comparison routine is specified, no prefix comparison is done.
lorder	The byte order for integers in the stored database meta data. The number should represent the order as an integer; for

example, big end-Ian order would be the number 4,321. If `lorder` is 0 (no order is specified) the current host order is used.

If the file already exists (and the O_TRUNC flag is not specified), the values specified for the parameters flags, lorder and psize are ignored in favor of the values used when the tree was created.

Forward sequential scans of a tree are from the smallest key to the largest.

Space freed by deleting key / data pairs from the tree is never reclaimed, although it is normally made available for reuse. This means that the btree storage structure is grow-only. The only solutions are to avoid excessive deletions, or to create a fresh tree periodically from a scan of an existing one.

Searches, insertions, and deletions in a btree will all complete in O lg base N where base is the average fill factor. Often, inserting ordered data into btrees results in a low fill factor. This implementation has been modified to make ordered insertion the best case, resulting in a better than usual page fill factor.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`dbopen`(3POSIX), `hash`(3POSIX), `mpool`(3POSIX), `recno`(3POSIX)

The Ubiquitous B-tree, Douglas Comer, ACM Comput. Surv. 11, 2 (June 1979), 121-138.

Prefix B-trees, Bayer and Unterauer, ACM Transactions on Database Systems, Vol. 2, 1 (March 1977), 11-26.

The Art of Computer Programming Vol. 3: Sorting and Searching, D.E. Knuth, 1968, pp 471-480.

BUGS

Only big and little endian byte order is supported.

RESTRICTIONS

These library calls do not support multi-threaded applications.

NAME

tcsetattr, tcgetattr, cfgetispeed, cfsetispeed, cfgetospeed, cfsetospeed, cfsetspeed, cfmakeraw – manipulating the termios structure

SYNOPSIS

#include <termios.h>
speed_t **cfgetispeed**(struct termios * *t*);

int **cfsetispeed**(struct termios * *t*, speed_t *speed*);

speed_t **cfgetospeed**(struct termios * *t*);

int **cfsetospeed**(struct termios * *t*, speed_t *speed*);

int **cfsetspeed**(struct termios * *t*, speed_t *speed*);

void **cfmakeraw**(struct termios * *t*);

int **tcgetattr**(int *fd*, struct termios * *t*);

int **tcsetattr**(int *fd*, int *action*, struct termios * *t*);

FEATURES

VTTY

DESCRIPTION

The `cfmakeraw()`, `tcgetattr()` and `tcsetattr()` functions are provided for getting and setting the termios structure.

The `cfgetispeed()`, `cfsetispeed()`, `cfgetospeed()`, `cfsetospeed()` and `cfsetspeed()` functions are provided for getting and setting the baud rate values in the termios structure. The effects of the functions on the terminal as described below do not become effective, nor are all errors detected, until the `tcsetattr()` function is called. Certain values for baud rates set in the termios structure and passed to `tcsetattr()` have special meanings. These are discussed in the portion of the manual page that describes the `tcsetattr()` function.

GETTING AND SETTING THE BAUD RATE

The input and output baud rates are found in the termios structure. The unsigned integer `speed_t` is `typdef` ed in the include file `<termios.h>`. The value of the integer corresponds directly to the baud rate being represented, however, the following symbolic values are defined:

```
#define B0          0
#define B50         50
#define B75         75
#define B110        110
#define B134        134
#define B150        150
#define B200        200
#define B300        300
#define B600        600
#define B1200       1200
#define B1800       1800
#define B2400       2400
#define B4800       4800
#define B9600       9600
```

```
#define B19200      19200
#define B38400      38400
#ifndef _POSIX_SOURCE
#define EXTA        19200
#define EXTB        38400
#endif  /*_POSIX_SOURCE */
```

The `cfgetispeed()` function returns the input baud rate in the termios structure referenced by *t* .

The `cfsetispeed()` function sets the input baud rate in the termios structure referenced by *t* to *speed* . The `cfgetospeed()` function returns the output baud rate in the termios structure referenced by *t* . The `cfsetospeed()` function sets the output baud rate in the termios structure referenced by *t* to *speed* .

The `cfsetspeed()` function sets both the input and output baud rate in the termios structure referenced by *t* to *speed* .

Upon successful completion, the functions `cfsetispeed()` , `cfsetospeed()` and `cfsetspeed()` return a value of `0` . Otherwise, a value of `-1` is returned and the global variable `errno` is set to indicate the error.

GETTING AND SETTING THE TERMIOS STATE

This section describes the functions that are used to control the general terminal interface. Unless otherwise noted for a specific command, these functions are restricted from use by background processes. Attempts to perform these operations will cause the process group to be sent a `SIGTTOU` signal. If the calling process is blocking or ignoring `SIGTTOU` signals, the process is allowed to perform the operation and the `SIGTTOU` signal is not sent. See RESTRICTIONS below.

In all the functions, although *fd* is an open file descriptor, the functions affect the underlying terminal file, not just the open file description associated with the particular file descriptor.

The `cfmakeraw()` function sets the flags stored in the termios structure to a state disabling all input and output processing, giving a raw I/O path. It should be noted that there is no function to reverse this effect. This is because there are a variety of processing options that could be re-enabled, and the correct method is for an application to snapshot the current terminal state using the `tcgetattr()` function, setting raw mode using `cfmakeraw()` and the subsequent `tcsetattr()` , and then using another `tcsetattr()` with the saved state to revert to the previous terminal state.

The `tcgetattr()` function copies the parameters associated with the terminal referenced by *fd* in the termios structure referenced by *t* . This function is allowed from a background process (see RESTRICTIONS); however, the terminal attributes may subsequently be changed by a foreground process.

The `tcsetattr()` function sets the parameters associated with the terminal from the termios structure referenced by *tp* . The *action* field is created by `or`-ing the following values, as specified in the include file `<termios.h>` .

TCSANOW	The change occurs immediately.
TCSADRAIN	The change occurs after all output written to *fd* has been transmitted to the terminal. This value of *action* should be used when changing parameters that affect output.
TCSAFLUSH	The change occurs after all output written to has been transmitted to the terminal. Additionally, any input that has been received but not read is discarded.
TCSASOFT	If this value is or'ed into the *action* value, the values of the *c_cflag* , *c_ispeed* , and *c_ospeed* fields are ignored.

The `0` baud rate is used to terminate the connection. If `0` is specified as the output speed to the function `tcsetattr()` , modem control will no longer be asserted on the terminal, disconnecting the terminal.

If `0` is specified as the input speed to the function `tcsetattr()` , the input baud rate will be set to the same value as that specified by the output baud rate.

If `tcsetattr()` is unable to make any of the requested changes, it returns `-1` and sets `errno` . Otherwise, it makes all of the requested changes it can. If the specified input and output baud rates differ and are a combination that is not supported, neither baud rate is changed.

Upon successful completion, the functions `tcgetattr()` and `tcsetattr()` return a value of `0` . Otherwise, they return `-1` and the global variable `errno` is set to indicate one of the following error conditions:

[EBADF]	The *fd* argument to `tcgetattr()` or `tcsetattr()` was not a valid file descriptor.
[EINTR]	The `tcsetattr()` function was interrupted by a signal. See RESTRICTIONS below.
[EINVAL]	The *action* argument to the `tcsetattr()` function was not valid, or an attempt was made to change an attribute represented in the termios structure to an unsupported value.
[ENOTTY]	The file associated with the *fd* argument to `tcgetattr()` or `tcsetattr()` is not a terminal.

STANDARDS

The `cfgetispeed()` , `cfsetispeed()` , `cfgetospeed()` , `cfsetospeed()` , `tcgetattr()` and `tcsetattr()` functions are expected to be compliant with the POSIX 1003.1-88 specification. The `cfmakeraw()` and `cfsetspeed()` functions, as well as the `TCSASOFT` option to the `tcsetattr()` function are extensions to the POSIX 1003.1-88 specification.

RESTRICTIONS

Signals and signals management are not supported.

These library functions (in `libbsd.a`) do not support multithreaded applications.

The background semantic is not supported.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

tcsetattr, tcgetattr, cfgetispeed, cfsetispeed, cfgetospeed, cfsetospeed, cfsetspeed, cfmakeraw – manipulating the termios structure

SYNOPSIS

#include <termios.h>
speed_t **cfgetispeed**(struct termios * *t*);

int **cfsetispeed**(struct termios * *t*, speed_t *speed*);

speed_t **cfgetospeed**(struct termios * *t*);

int **cfsetospeed**(struct termios * *t*, speed_t *speed*);

int **cfsetspeed**(struct termios * *t*, speed_t *speed*);

void **cfmakeraw**(struct termios * *t*);

int **tcgetattr**(int *fd*, struct termios * *t*);

int **tcsetattr**(int *fd*, int *action*, struct termios * *t*);

FEATURES

VTTY

DESCRIPTION

The `cfmakeraw()`, `tcgetattr()` and `tcsetattr()` functions are provided for getting and setting the termios structure.

The `cfgetispeed()`, `cfsetispeed()`, `cfgetospeed()`, `cfsetospeed()` and `cfsetspeed()` functions are provided for getting and setting the baud rate values in the termios structure. The effects of the functions on the terminal as described below do not become effective, nor are all errors detected, until the `tcsetattr()` function is called. Certain values for baud rates set in the termios structure and passed to `tcsetattr()` have special meanings. These are discussed in the portion of the manual page that describes the `tcsetattr()` function.

GETTING AND SETTING THE BAUD RATE

The input and output baud rates are found in the termios structure. The unsigned integer `speed_t` is `typdef` ed in the include file `<termios.h>`. The value of the integer corresponds directly to the baud rate being represented, however, the following symbolic values are defined:

```
#define B0          0
#define B50         50
#define B75         75
#define B110        110
#define B134        134
#define B150        150
#define B200        200
#define B300        300
#define B600        600
#define B1200       1200
#define B1800       1800
#define B2400       2400
#define B4800       4800
#define B9600       9600
```

```
#define B19200      19200
#define B38400      38400
#ifndef _POSIX_SOURCE
#define EXTA        19200
#define EXTB        38400
#endif  /*_POSIX_SOURCE */
```

The `cfgetispeed()` function returns the input baud rate in the termios structure referenced by *t* .

The `cfsetispeed()` function sets the input baud rate in the termios structure referenced by *t* to *speed* . The `cfgetospeed()` function returns the output baud rate in the termios structure referenced by *t* . The `cfsetospeed()` function sets the output baud rate in the termios structure referenced by *t* to *speed* .

The `cfsetspeed()` function sets both the input and output baud rate in the termios structure referenced by *t* to *speed* .

Upon successful completion, the functions `cfsetispeed()` , `cfsetospeed()` and `cfsetspeed()` return a value of `0` . Otherwise, a value of `-1` is returned and the global variable `errno` is set to indicate the error.

GETTING AND SETTING THE TERMIOS STATE

This section describes the functions that are used to control the general terminal interface. Unless otherwise noted for a specific command, these functions are restricted from use by background processes. Attempts to perform these operations will cause the process group to be sent a `SIGTTOU` signal. If the calling process is blocking or ignoring `SIGTTOU` signals, the process is allowed to perform the operation and the `SIGTTOU` signal is not sent. See RESTRICTIONS below.

In all the functions, although *fd* is an open file descriptor, the functions affect the underlying terminal file, not just the open file description associated with the particular file descriptor.

The `cfmakeraw()` function sets the flags stored in the termios structure to a state disabling all input and output processing, giving a raw I/O path. It should be noted that there is no function to reverse this effect. This is because there are a variety of processing options that could be re-enabled, and the correct method is for an application to snapshot the current terminal state using the `tcgetattr()` function, setting raw mode using `cfmakeraw()` and the subsequent `tcsetattr()` , and then using another `tcsetattr()` with the saved state to revert to the previous terminal state.

The `tcgetattr()` function copies the parameters associated with the terminal referenced by *fd* in the termios structure referenced by *t* . This function is allowed from a background process (see RESTRICTIONS); however, the terminal attributes may subsequently be changed by a foreground process.

The `tcsetattr()` function sets the parameters associated with the terminal from the termios structure referenced by *tp* . The *action* field is created by `or`-ing the following values, as specified in the include file `<termios.h>` .

TCSANOW	The change occurs immediately.
TCSADRAIN	The change occurs after all output written to *fd* has been transmitted to the terminal. This value of *action* should be used when changing parameters that affect output.
TCSAFLUSH	The change occurs after all output written to has been transmitted to the terminal. Additionally, any input that has been received but not read is discarded.
TCSASOFT	If this value is or'ed into the *action* value, the values of the *c_cflag* , *c_ispeed* , and *c_ospeed* fields are ignored.

The `0` baud rate is used to terminate the connection. If `0` is specified as the output speed to the function `tcsetattr()` , modem control will no longer be asserted on the terminal, disconnecting the terminal.

If `0` is specified as the input speed to the function `tcsetattr()` , the input baud rate will be set to the same value as that specified by the output baud rate.

If `tcsetattr()` is unable to make any of the requested changes, it returns `-1` and sets `errno` . Otherwise, it makes all of the requested changes it can. If the specified input and output baud rates differ and are a combination that is not supported, neither baud rate is changed.

Upon successful completion, the functions `tcgetattr()` and `tcsetattr()` return a value of `0` . Otherwise, they return `-1` and the global variable `errno` is set to indicate one of the following error conditions:

[EBADF]	The *fd* argument to `tcgetattr()` or `tcsetattr()` was not a valid file descriptor.
[EINTR]	The `tcsetattr()` function was interrupted by a signal. See RESTRICTIONS below.
[EINVAL]	The *action* argument to the `tcsetattr()` function was not valid, or an attempt was made to change an attribute represented in the termios structure to an unsupported value.
[ENOTTY]	The file associated with the *fd* argument to `tcgetattr()` or `tcsetattr()` is not a terminal.

STANDARDS

The `cfgetispeed()` , `cfsetispeed()` , `cfgetospeed()` , `cfsetospeed()` , `tcgetattr()` and `tcsetattr()` functions are expected to be compliant with the POSIX 1003.1-88 specification. The `cfmakeraw()` and `cfsetspeed()` functions, as well as the `TCSASOFT` option to the `tcsetattr()` function are extensions to the POSIX 1003.1-88 specification.

RESTRICTIONS

Signals and signals management are not supported.

These library functions (in `libbsd.a`) do not support multithreaded applications.

The background semantic is not supported.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

tcsetattr, tcgetattr, cfgetispeed, cfsetispeed, cfgetospeed, cfsetospeed, cfsetspeed, cfmakeraw – manipulating the termios structure

SYNOPSIS

#include <termios.h>
speed_t **cfgetispeed**(struct termios * *t*);

int **cfsetispeed**(struct termios * *t*, speed_t *speed*);

speed_t **cfgetospeed**(struct termios * *t*);

int **cfsetospeed**(struct termios * *t*, speed_t *speed*);

int **cfsetspeed**(struct termios * *t*, speed_t *speed*);

void **cfmakeraw**(struct termios * *t*);

int **tcgetattr**(int *fd*, struct termios * *t*);

int **tcsetattr**(int *fd*, int *action*, struct termios * *t*);

FEATURES

VTTY

DESCRIPTION

The `cfmakeraw()`, `tcgetattr()` and `tcsetattr()` functions are provided for getting and setting the termios structure.

The `cfgetispeed()`, `cfsetispeed()`, `cfgetospeed()`, `cfsetospeed()` and `cfsetspeed()` functions are provided for getting and setting the baud rate values in the termios structure. The effects of the functions on the terminal as described below do not become effective, nor are all errors detected, until the `tcsetattr()` function is called. Certain values for baud rates set in the termios structure and passed to `tcsetattr()` have special meanings. These are discussed in the portion of the manual page that describes the `tcsetattr()` function.

GETTING AND SETTING THE BAUD RATE

The input and output baud rates are found in the termios structure. The unsigned integer `speed_t` is `typdef` ed in the include file `<termios.h>`. The value of the integer corresponds directly to the baud rate being represented, however, the following symbolic values are defined:

```
#define B0          0
#define B50         50
#define B75         75
#define B110        110
#define B134        134
#define B150        150
#define B200        200
#define B300        300
#define B600        600
#define B1200       1200
#define B1800       1800
#define B2400       2400
#define B4800       4800
#define B9600       9600
```

```
#define B19200      19200
#define B38400      38400
#ifndef _POSIX_SOURCE
#define EXTA        19200
#define EXTB        38400
#endif  /*_POSIX_SOURCE */
```

The `cfgetispeed( )` function returns the input baud rate in the termios structure referenced by *t* .

The `cfsetispeed( )` function sets the input baud rate in the termios structure referenced by *t* to *speed* . The `cfgetospeed( )` function returns the output baud rate in the termios structure referenced by *t* . The `cfsetospeed( )` function sets the output baud rate in the termios structure referenced by *t* to *speed* .

The `cfsetspeed( )` function sets both the input and output baud rate in the termios structure referenced by *t* to *speed* .

Upon successful completion, the functions `cfsetispeed( )` , `cfsetospeed( )` and `cfsetspeed( )` return a value of `0` . Otherwise, a value of `-1` is returned and the global variable `errno` is set to indicate the error.

GETTING AND SETTING THE TERMIOS STATE

This section describes the functions that are used to control the general terminal interface. Unless otherwise noted for a specific command, these functions are restricted from use by background processes. Attempts to perform these operations will cause the process group to be sent a `SIGTTOU` signal. If the calling process is blocking or ignoring `SIGTTOU` signals, the process is allowed to perform the operation and the `SIGTTOU` signal is not sent. See RESTRICTIONS below.

In all the functions, although *fd* is an open file descriptor, the functions affect the underlying terminal file, not just the open file description associated with the particular file descriptor.

The `cfmakeraw( )` function sets the flags stored in the termios structure to a state disabling all input and output processing, giving a raw I/O path. It should be noted that there is no function to reverse this effect. This is because there are a variety of processing options that could be re-enabled, and the correct method is for an application to snapshot the current terminal state using the `tcgetattr( )` function, setting raw mode using `cfmakeraw( )` and the subsequent `tcsetattr( )` , and then using another `tcsetattr( )` with the saved state to revert to the previous terminal state.

The `tcgetattr( )` function copies the parameters associated with the terminal referenced by *fd* in the termios structure referenced by *t* . This function is allowed from a background process (see RESTRICTIONS); however, the terminal attributes may subsequently be changed by a foreground process.

The `tcsetattr( )` function sets the parameters associated with the terminal from the termios structure referenced by *tp* . The *action* field is created by `or`-ing the following values, as specified in the include file `<termios.h>` .

TCSANOW	The change occurs immediately.
TCSADRAIN	The change occurs after all output written to *fd* has been transmitted to the terminal. This value of *action* should be used when changing parameters that affect output.
TCSAFLUSH	The change occurs after all output written to has been transmitted to the terminal. Additionally, any input that has been received but not read is discarded.
TCSASOFT	If this value is or'ed into the *action* value, the values of the *c_cflag* , *c_ispeed* , and *c_ospeed* fields are ignored.

The `0` baud rate is used to terminate the connection. If `0` is specified as the output speed to the function `tcsetattr( )` , modem control will no longer be asserted on the terminal, disconnecting the terminal.

If `0` is specified as the input speed to the function `tcsetattr( )` , the input baud rate will be set to the same value as that specified by the output baud rate.

If `tcsetattr( )` is unable to make any of the requested changes, it returns `-1` and sets `errno` . Otherwise, it makes all of the requested changes it can. If the specified input and output baud rates differ and are a combination that is not supported, neither baud rate is changed.

Upon successful completion, the functions `tcgetattr( )` and `tcsetattr( )` return a value of `0` . Otherwise, they return `-1` and the global variable `errno` is set to indicate one of the following error conditions:

[EBADF]	The *fd* argument to `tcgetattr( )` or `tcsetattr( )` was not a valid file descriptor.
[EINTR]	The `tcsetattr( )` function was interrupted by a signal. See RESTRICTIONS below.
[EINVAL]	The *action* argument to the `tcsetattr( )` function was not valid, or an attempt was made to change an attribute represented in the termios structure to an unsupported value.
[ENOTTY]	The file associated with the *fd* argument to `tcgetattr( )` or `tcsetattr( )` is not a terminal.

STANDARDS

The `cfgetispeed( )` , `cfsetispeed( )` , `cfgetospeed( )` , `cfsetospeed( )` , `tcgetattr( )` and `tcsetattr( )` functions are expected to be compliant with the POSIX 1003.1-88 specification. The `cfmakeraw( )` and `cfsetspeed( )` functions, as well as the `TCSASOFT` option to the `tcsetattr( )` function are extensions to the POSIX 1003.1-88 specification.

RESTRICTIONS

Signals and signals management are not supported.

These library functions (in `libbsd.a`) do not support multithreaded applications.

The background semantic is not supported.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

tcsetattr, tcgetattr, cfgetispeed, cfsetispeed, cfgetospeed, cfsetospeed, cfsetspeed, cfmakeraw – manipulating the termios structure

SYNOPSIS

#include <termios.h>
speed_t **cfgetispeed**(struct termios * *t*);

int **cfsetispeed**(struct termios * *t*, speed_t *speed*);

speed_t **cfgetospeed**(struct termios * *t*);

int **cfsetospeed**(struct termios * *t*, speed_t *speed*);

int **cfsetspeed**(struct termios * *t*, speed_t *speed*);

void **cfmakeraw**(struct termios * *t*);

int **tcgetattr**(int *fd*, struct termios * *t*);

int **tcsetattr**(int *fd*, int *action*, struct termios * *t*);

FEATURES

VTTY

DESCRIPTION

The `cfmakeraw()`, `tcgetattr()` and `tcsetattr()` functions are provided for getting and setting the termios structure.

The `cfgetispeed()`, `cfsetispeed()`, `cfgetospeed()`, `cfsetospeed()` and `cfsetspeed()` functions are provided for getting and setting the baud rate values in the termios structure. The effects of the functions on the terminal as described below do not become effective, nor are all errors detected, until the `tcsetattr()` function is called. Certain values for baud rates set in the termios structure and passed to `tcsetattr()` have special meanings. These are discussed in the portion of the manual page that describes the `tcsetattr()` function.

GETTING AND SETTING THE BAUD RATE

The input and output baud rates are found in the termios structure. The unsigned integer `speed_t` is `typdef` ed in the include file `<termios.h>`. The value of the integer corresponds directly to the baud rate being represented, however, the following symbolic values are defined:

```
#define B0          0
#define B50         50
#define B75         75
#define B110        110
#define B134        134
#define B150        150
#define B200        200
#define B300        300
#define B600        600
#define B1200       1200
#define B1800       1800
#define B2400       2400
#define B4800       4800
#define B9600       9600
```

```
#define B19200      19200
#define B38400      38400
#ifndef _POSIX_SOURCE
#define EXTA        19200
#define EXTB        38400
#endif  /*_POSIX_SOURCE */
```

The `cfgetispeed()` function returns the input baud rate in the termios structure referenced by *t* .

The `cfsetispeed()` function sets the input baud rate in the termios structure referenced by *t* to *speed* . The `cfgetospeed()` function returns the output baud rate in the termios structure referenced by *t* . The `cfsetospeed()` function sets the output baud rate in the termios structure referenced by *t* to *speed* .

The `cfsetspeed()` function sets both the input and output baud rate in the termios structure referenced by *t* to *speed* .

Upon successful completion, the functions `cfsetispeed()` , `cfsetospeed()` and `cfsetspeed()` return a value of `0` . Otherwise, a value of `-1` is returned and the global variable `errno` is set to indicate the error.

GETTING AND SETTING THE TERMIOS STATE

This section describes the functions that are used to control the general terminal interface. Unless otherwise noted for a specific command, these functions are restricted from use by background processes. Attempts to perform these operations will cause the process group to be sent a `SIGTTOU` signal. If the calling process is blocking or ignoring `SIGTTOU` signals, the process is allowed to perform the operation and the `SIGTTOU` signal is not sent. See RESTRICTIONS below.

In all the functions, although *fd* is an open file descriptor, the functions affect the underlying terminal file, not just the open file description associated with the particular file descriptor.

The `cfmakeraw()` function sets the flags stored in the termios structure to a state disabling all input and output processing, giving a raw I/O path. It should be noted that there is no function to reverse this effect. This is because there are a variety of processing options that could be re-enabled, and the correct method is for an application to snapshot the current terminal state using the `tcgetattr()` function, setting raw mode using `cfmakeraw()` and the subsequent `tcsetattr()` , and then using another `tcsetattr()` with the saved state to revert to the previous terminal state.

The `tcgetattr()` function copies the parameters associated with the terminal referenced by *fd* in the termios structure referenced by *t* . This function is allowed from a background process (see RESTRICTIONS); however, the terminal attributes may subsequently be changed by a foreground process.

The `tcsetattr()` function sets the parameters associated with the terminal from the termios structure referenced by *tp* . The *action* field is created by `or`-ing the following values, as specified in the include file `<termios.h>` .

TCSANOW	The change occurs immediately.
TCSADRAIN	The change occurs after all output written to *fd* has been transmitted to the terminal. This value of *action* should be used when changing parameters that affect output.
TCSAFLUSH	The change occurs after all output written to has been transmitted to the terminal. Additionally, any input that has been received but not read is discarded.
TCSASOFT	If this value is or'ed into the *action* value, the values of the *c_cflag* , *c_ispeed* , and *c_ospeed* fields are ignored.

The `0` baud rate is used to terminate the connection. If `0` is specified as the output speed to the function `tcsetattr()` , modem control will no longer be asserted on the terminal, disconnecting the terminal.

If `0` is specified as the input speed to the function `tcsetattr()` , the input baud rate will be set to the same value as that specified by the output baud rate.

If `tcsetattr()` is unable to make any of the requested changes, it returns `-1` and sets `errno` . Otherwise, it makes all of the requested changes it can. If the specified input and output baud rates differ and are a combination that is not supported, neither baud rate is changed.

Upon successful completion, the functions `tcgetattr()` and `tcsetattr()` return a value of `0` . Otherwise, they return `-1` and the global variable `errno` is set to indicate one of the following error conditions:

[EBADF]	The *fd* argument to `tcgetattr()` or `tcsetattr()` was not a valid file descriptor.
[EINTR]	The `tcsetattr()` function was interrupted by a signal. See RESTRICTIONS below.
[EINVAL]	The *action* argument to the `tcsetattr()` function was not valid, or an attempt was made to change an attribute represented in the termios structure to an unsupported value.
[ENOTTY]	The file associated with the *fd* argument to `tcgetattr()` or `tcsetattr()` is not a terminal.

STANDARDS

The `cfgetispeed()` , `cfsetispeed()` , `cfgetospeed()` , `cfsetospeed()` , `tcgetattr()` and `tcsetattr()` functions are expected to be compliant with the POSIX 1003.1-88 specification. The `cfmakeraw()` and `cfsetspeed()` functions, as well as the `TCSASOFT` option to the `tcsetattr()` function are extensions to the POSIX 1003.1-88 specification.

RESTRICTIONS

Signals and signals management are not supported.

These library functions (in `libbsd.a`) do not support multithreaded applications.

The background semantic is not supported.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

tcsetattr, tcgetattr, cfgetispeed, cfsetispeed, cfgetospeed, cfsetospeed, cfsetspeed, cfmakeraw – manipulating the termios structure

SYNOPSIS

#include <termios.h>
speed_t **cfgetispeed**(struct termios * *t*);

int **cfsetispeed**(struct termios * *t*, speed_t *speed*);

speed_t **cfgetospeed**(struct termios * *t*);

int **cfsetospeed**(struct termios * *t*, speed_t *speed*);

int **cfsetspeed**(struct termios * *t*, speed_t *speed*);

void **cfmakeraw**(struct termios * *t*);

int **tcgetattr**(int *fd*, struct termios * *t*);

int **tcsetattr**(int *fd*, int *action*, struct termios * *t*);

FEATURES

VTTY

DESCRIPTION

The `cfmakeraw()` , `tcgetattr()` and `tcsetattr()` functions are provided for getting and setting the termios structure.

The `cfgetispeed()` , `cfsetispeed()` , `cfgetospeed()` , `cfsetospeed()` and `cfsetspeed()` functions are provided for getting and setting the baud rate values in the termios structure. The effects of the functions on the terminal as described below do not become effective, nor are all errors detected, until the `tcsetattr()` function is called. Certain values for baud rates set in the termios structure and passed to `tcsetattr()` have special meanings. These are discussed in the portion of the manual page that describes the `tcsetattr()` function.

GETTING AND SETTING THE BAUD RATE

The input and output baud rates are found in the termios structure. The unsigned integer `speed_t` is `typdef` ed in the include file `<termios.h>` . The value of the integer corresponds directly to the baud rate being represented, however, the following symbolic values are defined:

```
#define B0          0
#define B50         50
#define B75         75
#define B110        110
#define B134        134
#define B150        150
#define B200        200
#define B300        300
#define B600        600
#define B1200       1200
#define B1800       1800
#define B2400       2400
#define B4800       4800
#define B9600       9600
```

```
#define B19200      19200
#define B38400      38400
#ifndef _POSIX_SOURCE
#define EXTA        19200
#define EXTB        38400
#endif  /*_POSIX_SOURCE */
```

The `cfgetispeed( )` function returns the input baud rate in the termios structure referenced by *t* .

The `cfsetispeed( )` function sets the input baud rate in the termios structure referenced by *t* to *speed* . The `cfgetospeed( )` function returns the output baud rate in the termios structure referenced by *t* . The `cfsetospeed( )` function sets the output baud rate in the termios structure referenced by *t* to *speed* .

The `cfsetspeed( )` function sets both the input and output baud rate in the termios structure referenced by *t* to *speed* .

Upon successful completion, the functions `cfsetispeed( )` , `cfsetospeed( )` and `cfsetspeed( )` return a value of `0` . Otherwise, a value of `-1` is returned and the global variable `errno` is set to indicate the error.

GETTING AND SETTING THE TERMIOS STATE

This section describes the functions that are used to control the general terminal interface. Unless otherwise noted for a specific command, these functions are restricted from use by background processes. Attempts to perform these operations will cause the process group to be sent a `SIGTTOU` signal. If the calling process is blocking or ignoring `SIGTTOU` signals, the process is allowed to perform the operation and the `SIGTTOU` signal is not sent. See RESTRICTIONS below.

In all the functions, although *fd* is an open file descriptor, the functions affect the underlying terminal file, not just the open file description associated with the particular file descriptor.

The `cfmakeraw( )` function sets the flags stored in the termios structure to a state disabling all input and output processing, giving a raw I/O path. It should be noted that there is no function to reverse this effect. This is because there are a variety of processing options that could be re-enabled, and the correct method is for an application to snapshot the current terminal state using the `tcgetattr( )` function, setting raw mode using `cfmakeraw( )` and the subsequent `tcsetattr( )` , and then using another `tcsetattr( )` with the saved state to revert to the previous terminal state.

The `tcgetattr( )` function copies the parameters associated with the terminal referenced by *fd* in the termios structure referenced by *t* . This function is allowed from a background process (see RESTRICTIONS); however, the terminal attributes may subsequently be changed by a foreground process.

The `tcsetattr()` function sets the parameters associated with the terminal from the termios structure referenced by *tp* . The *action* field is created by `or`-ing the following values, as specified in the include file `<termios.h>` .

TCSANOW	The change occurs immediately.
TCSADRAIN	The change occurs after all output written to *fd* has been transmitted to the terminal. This value of *action* should be used when changing parameters that affect output.
TCSAFLUSH	The change occurs after all output written to has been transmitted to the terminal. Additionally, any input that has been received but not read is discarded.
TCSASOFT	If this value is or'ed into the *action* value, the values of the *c_cflag* , *c_ispeed* , and *c_ospeed* fields are ignored.

The `0` baud rate is used to terminate the connection. If `0` is specified as the output speed to the function `tcsetattr()` , modem control will no longer be asserted on the terminal, disconnecting the terminal.

If `0` is specified as the input speed to the function `tcsetattr()` , the input baud rate will be set to the same value as that specified by the output baud rate.

If `tcsetattr()` is unable to make any of the requested changes, it returns `-1` and sets `errno` . Otherwise, it makes all of the requested changes it can. If the specified input and output baud rates differ and are a combination that is not supported, neither baud rate is changed.

Upon successful completion, the functions `tcgetattr()` and `tcsetattr()` return a value of `0` . Otherwise, they return `-1` and the global variable `errno` is set to indicate one of the following error conditions:

[EBADF]	The *fd* argument to `tcgetattr()` or `tcsetattr()` was not a valid file descriptor.
[EINTR]	The `tcsetattr()` function was interrupted by a signal. See RESTRICTIONS below.
[EINVAL]	The *action* argument to the `tcsetattr()` function was not valid, or an attempt was made to change an attribute represented in the termios structure to an unsupported value.
[ENOTTY]	The file associated with the *fd* argument to `tcgetattr()` or `tcsetattr()` is not a terminal.

STANDARDS

The `cfgetispeed()` , `cfsetispeed()` , `cfgetospeed()` , `cfsetospeed()` , `tcgetattr()` and `tcsetattr()` functions are expected to be compliant with the POSIX 1003.1-88 specification. The `cfmakeraw()` and `cfsetspeed()` functions, as well as the `TCSASOFT` option to the `tcsetattr()` function are extensions to the POSIX 1003.1-88 specification.

RESTRICTIONS

Signals and signals management are not supported.

These library functions (in `libbsd.a`) do not support multithreaded applications.

The background semantic is not supported.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

tcsetattr, tcgetattr, cfgetispeed, cfsetispeed, cfgetospeed, cfsetospeed, cfsetspeed, cfmakeraw – manipulating the termios structure

SYNOPSIS

#include <termios.h>
speed_t **cfgetispeed**(struct termios * *t*);

int **cfsetispeed**(struct termios * *t*, speed_t *speed*);

speed_t **cfgetospeed**(struct termios * *t*);

int **cfsetospeed**(struct termios * *t*, speed_t *speed*);

int **cfsetspeed**(struct termios * *t*, speed_t *speed*);

void **cfmakeraw**(struct termios * *t*);

int **tcgetattr**(int *fd*, struct termios * *t*);

int **tcsetattr**(int *fd*, int *action*, struct termios * *t*);

FEATURES

VTTY

DESCRIPTION

The `cfmakeraw()`, `tcgetattr()` and `tcsetattr()` functions are provided for getting and setting the termios structure.

The `cfgetispeed()`, `cfsetispeed()`, `cfgetospeed()`, `cfsetospeed()` and `cfsetspeed()` functions are provided for getting and setting the baud rate values in the termios structure. The effects of the functions on the terminal as described below do not become effective, nor are all errors detected, until the `tcsetattr()` function is called. Certain values for baud rates set in the termios structure and passed to `tcsetattr()` have special meanings. These are discussed in the portion of the manual page that describes the `tcsetattr()` function.

GETTING AND SETTING THE BAUD RATE

The input and output baud rates are found in the termios structure. The unsigned integer `speed_t` is `typdef` ed in the include file `<termios.h>`. The value of the integer corresponds directly to the baud rate being represented, however, the following symbolic values are defined:

```
#define B0          0
#define B50         50
#define B75         75
#define B110        110
#define B134        134
#define B150        150
#define B200        200
#define B300        300
#define B600        600
#define B1200       1200
#define B1800       1800
#define B2400       2400
#define B4800       4800
#define B9600       9600
```

```
#define B19200      19200
#define B38400      38400
#ifndef _POSIX_SOURCE
#define EXTA        19200
#define EXTB        38400
#endif  /*_POSIX_SOURCE */
```

The `cfgetispeed()` function returns the input baud rate in the termios structure referenced by *t* .

The `cfsetispeed()` function sets the input baud rate in the termios structure referenced by *t* to *speed* . The `cfgetospeed()` function returns the output baud rate in the termios structure referenced by *t* . The `cfsetospeed()` function sets the output baud rate in the termios structure referenced by *t* to *speed* .

The `cfsetspeed()` function sets both the input and output baud rate in the termios structure referenced by *t* to *speed* .

Upon successful completion, the functions `cfsetispeed()` , `cfsetospeed()` and `cfsetspeed()` return a value of `0` . Otherwise, a value of `-1` is returned and the global variable `errno` is set to indicate the error.

GETTING AND SETTING THE TERMIOS STATE

This section describes the functions that are used to control the general terminal interface. Unless otherwise noted for a specific command, these functions are restricted from use by background processes. Attempts to perform these operations will cause the process group to be sent a `SIGTTOU` signal. If the calling process is blocking or ignoring `SIGTTOU` signals, the process is allowed to perform the operation and the `SIGTTOU` signal is not sent. See RESTRICTIONS below.

In all the functions, although *fd* is an open file descriptor, the functions affect the underlying terminal file, not just the open file description associated with the particular file descriptor.

The `cfmakeraw()` function sets the flags stored in the termios structure to a state disabling all input and output processing, giving a raw I/O path. It should be noted that there is no function to reverse this effect. This is because there are a variety of processing options that could be re-enabled, and the correct method is for an application to snapshot the current terminal state using the `tcgetattr()` function, setting raw mode using `cfmakeraw()` and the subsequent `tcsetattr()` , and then using another `tcsetattr()` with the saved state to revert to the previous terminal state.

The `tcgetattr()` function copies the parameters associated with the terminal referenced by *fd* in the termios structure referenced by *t* . This function is allowed from a background process (see RESTRICTIONS); however, the terminal attributes may subsequently be changed by a foreground process.

The `tcsetattr()` function sets the parameters associated with the terminal from the termios structure referenced by *tp* . The *action* field is created by `or`-ing the following values, as specified in the include file `<termios.h>` .

TCSANOW	The change occurs immediately.
TCSADRAIN	The change occurs after all output written to *fd* has been transmitted to the terminal. This value of *action* should be used when changing parameters that affect output.
TCSAFLUSH	The change occurs after all output written to has been transmitted to the terminal. Additionally, any input that has been received but not read is discarded.
TCSASOFT	If this value is or'ed into the *action* value, the values of the *c_cflag* , *c_ispeed* , and *c_ospeed* fields are ignored.

The `0` baud rate is used to terminate the connection. If `0` is specified as the output speed to the function `tcsetattr()` , modem control will no longer be asserted on the terminal, disconnecting the terminal.

If `0` is specified as the input speed to the function `tcsetattr()` , the input baud rate will be set to the same value as that specified by the output baud rate.

If `tcsetattr()` is unable to make any of the requested changes, it returns `-1` and sets `errno` . Otherwise, it makes all of the requested changes it can. If the specified input and output baud rates differ and are a combination that is not supported, neither baud rate is changed.

Upon successful completion, the functions `tcgetattr()` and `tcsetattr()` return a value of `0` . Otherwise, they return `-1` and the global variable `errno` is set to indicate one of the following error conditions:

[EBADF]	The *fd* argument to `tcgetattr()` or `tcsetattr()` was not a valid file descriptor.
[EINTR]	The `tcsetattr()` function was interrupted by a signal. See RESTRICTIONS below.
[EINVAL]	The *action* argument to the `tcsetattr()` function was not valid, or an attempt was made to change an attribute represented in the termios structure to an unsupported value.
[ENOTTY]	The file associated with the *fd* argument to `tcgetattr()` or `tcsetattr()` is not a terminal.

STANDARDS

The `cfgetispeed()` , `cfsetispeed()` , `cfgetospeed()` , `cfsetospeed()` , `tcgetattr()` and `tcsetattr()` functions are expected to be compliant with the POSIX 1003.1-88 specification. The `cfmakeraw()` and `cfsetspeed()` functions, as well as the `TCSASOFT` option to the `tcsetattr()` function are extensions to the POSIX 1003.1-88 specification.

RESTRICTIONS

Signals and signals management are not supported.

These library functions (in `libbsd.a`) do not support multithreaded applications.

The background semantic is not supported.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	**ATTRIBUTE VALUE**
Interface Stability	Evolving

NAME

clock_settime, clock_gettime, clock_getres – get or set clock to specified value, or get clock resolution

SYNOPSIS

#include <time.h>
int **clock_settime**(clockid_t *clock_id*, const struct timespec * *tp*);

int **clock_gettime**(clockid_t *clock_id*, struct timespec * *tp*);

int **clock_getres**(clockid_t *clock_id*, struct timespec * *res*);

DESCRIPTION

The *clock_settime* function sets 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 clock specified are truncated to the smaller multiple of the resolution. Only supervisor threads or threads of system actors (see *actorCreate* (2K)) are permitted to set a clock value.

The *clock_gettime* function stores the current value for the specified clock, *clock_id* , in *tp* .

The *clock_getres* function returns the resolution of the specified clock in *res* unless *res* is NULL, in which case only validity checking is performed. The clock resolution is platform-defined and is not settable by an application.

The only clock supported is the system-wide realtime clock, with the *clock_id* of CLOCK_REALTIME. This clock, as set by *clock_settime* , usu ally represents the time of day.

RETURN VALUE

Upon successful completion, *clock_settime* , *clock_gettime* , and *clock_getres* return zero.

Otherwise a value of –1 is returned and *errno* is set to indicate the error condition.

ERRORS

[EFAULT]	A pointer argument contains an address outside the current actor's address space.
[EINVAL]	The *clock_id* argument specifies a clock other than CLOCK_REALTIME. The *tp* argument is NULL (*clock_settime* and *clock_gettime* only). The time specification referenced by the *tp* argument contains an impossible value (*clock_settime* only).
[EPERM]	The current thread is neither a supervisor thread nor a thread of a system actor (*clock_settime* only).

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

clock_settime, clock_gettime, clock_getres – get or set clock to specified value, or get clock resolution

SYNOPSIS

```
#include <time.h>
int clock_settime(clockid_t clock_id, const struct timespec * tp);

int clock_gettime(clockid_t clock_id, struct timespec * tp);

int clock_getres(clockid_t clock_id, struct timespec * res);
```

DESCRIPTION

The *clock_settime* function sets 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 clock specified are truncated to the smaller multiple of the resolution. Only supervisor threads or threads of system actors (see *actorCreate* (2K)) are permitted to set a clock value.

The *clock_gettime* function stores the current value for the specified clock, *clock_id* , in *tp* .

The *clock_getres* function returns the resolution of the specified clock in *res* unless *res* is NULL, in which case only validity checking is performed. The clock resolution is platform-defined and is not settable by an application.

The only clock supported is the system-wide realtime clock, with the *clock_id* of CLOCK_REALTIME. This clock, as set by *clock_settime* , usu ally represents the time of day.

RETURN VALUE

Upon successful completion, *clock_settime* , *clock_gettime* , and *clock_getres* return zero.

Otherwise a value of –1 is returned and *errno* is set to indicate the error condition.

ERRORS

[EFAULT]	A pointer argument contains an address outside the current actor's address space.
[EINVAL]	The *clock_id* argument specifies a clock other than CLOCK_REALTIME. The *tp* argument is NULL (*clock_settime* and *clock_gettime* only). The time specification referenced by the *tp* argument contains an impossible value (*clock_settime* only).
[EPERM]	The current thread is neither a supervisor thread nor a thread of a system actor (*clock_settime* only).

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

clock_settime, clock_gettime, clock_getres – get or set clock to specified value, or get clock resolution

SYNOPSIS

#include <time.h>
int **clock_settime**(clockid_t *clock_id*, const struct timespec * *tp*);

int **clock_gettime**(clockid_t *clock_id*, struct timespec * *tp*);

int **clock_getres**(clockid_t *clock_id*, struct timespec * *res*);

DESCRIPTION

The *clock_settime* function sets 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 clock specified are truncated to the smaller multiple of the resolution. Only supervisor threads or threads of system actors (see *actorCreate* (2K)) are permitted to set a clock value.

The *clock_gettime* function stores the current value for the specified clock, *clock_id* , in *tp* .

The *clock_getres* function returns the resolution of the specified clock in *res* unless *res* is NULL, in which case only validity checking is performed. The clock resolution is platform-defined and is not settable by an application.

The only clock supported is the system-wide realtime clock, with the *clock_id* of CLOCK_REALTIME. This clock, as set by *clock_settime* , usu ally represents the time of day.

RETURN VALUE

Upon successful completion, *clock_settime* , *clock_gettime* , and *clock_getres* return zero.

Otherwise a value of –1 is returned and *errno* is set to indicate the error condition.

ERRORS

[EFAULT]	A pointer argument contains an address outside the current actor’s address space.
[EINVAL]	The *clock_id* argument specifies a clock other than CLOCK_REALTIME. The *tp* argument is NULL (*clock_settime* and *clock_gettime* only). The time specification referenced by the *tp* argument contains an impossible value (*clock_settime* only).
[EPERM]	The current thread is neither a supervisor thread nor a thread of a system actor (*clock_settime* only).

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

directory, opendir, readdir, telldir, seekdir, rewinddir, closedir – directory operations

SYNOPSIS

#include <sys/types.h>
#include <dirent.h>
DIR * **opendir**(const char * *filename*);

struct dirent * **readdir**(DIR * *dirp*);

long **telldir**(const DIR * *dirp*);

void **seekdir**(DIR * *dirp*, long *loc*);

void **rewinddir**(DIR * *dirp*);

int **closedir**(DIR * *dirp*);

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

The *opendir* function opens the directory named by *filename* , associates a directory stream with it and returns a pointer to be used to identify the directory stream in subsequent operations. The NULL pointer is returned if *filename* cannot be accessed, or if it cannot *malloc(3STDC)* enough memory to hold all of it.

The *readdir* function returns a pointer to the next directory entry. It returns NULL upon reaching the end of the directory or detecting an invalid *seekdir* operation.

The *telldir* function returns the current location associated with the named directory stream.

The *seekdir* function sets the position of the next *readdir* operation on the directory stream. The new position reverts to the one associated with the directory stream when the *telldir* operation was performed. Values returned by *telldir* are valid only for the lifetime of the DIR pointer, *dirp* , from which they are derived. If the directory is closed and then reopened, the *telldir* value may be invalidated due to undetected directory compaction. It is safe to use a previous *telldir* value immediately after a call to *opendir* and before any calls to *readdir* .

The *rewinddir* function resets the position of the named directory stream to the beginning of the directory.

The *closedir* function closes the named directory stream and frees the structure associated with the *dirp* pointer, returning 0 on success. On failure, –1 is returned and the global variable *errno* is set to indicate the error.

Sample code which searches a directory for the “name” entry is:

```
len = strlen(name);
dirp = opendir(".");
if (dirp) {
   while ((dp = readdir(dirp)) != NULL)
 if (dp->d_namlen == len && !strcmp(dp->d_name, name)) {
  (void) closedir(dirp);
  return FOUND;
         }
   (void) closedir(dirp);
   }
return NOT_FOUND;
```

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`open`(2POSIX) , `close`(2POSIX) , `read`(2POSIX) , `lseek`(2POSIX)

HISTORY

The *opendir, readdir, telldir, seekdir, rewinddir* and *closedir* functions appeared in 4.2 BSD.

RESTRICTIONS

These library calls do not support multi-threaded applications.

NAME

dbopen – database access methods

SYNOPSIS

```
#include <sys/types.h>
#include <limits.h>
#include <db.h>
DB *dbopen(const char *file, int flags, int mode, DBTYPE type, const void *openinfo);
```

FEATURES

UFS

DESCRIPTION

The *dbopen* function is the library interface to database files. The file formats supported are btree, hashed and UNIX file oriented. The btree format is a representation of a sorted, balanced tree structure. The hashed format is an extensible, dynamic hashing scheme. The flat-file format is a byte stream file with fixed or variable length records. The formats and file format—specific information are described in detail in their respective manual pages *btree(3POSIX), hash(3POSIX)* and *recno(3POSIX).*

The *dbopen* function opens a `file` for reading and / or writing. Files not intended to be preserved on disk may be created by setting the file parameter to NULL.

The *flags* and *mode* arguments are as specified for the *open(2POSIX)* routine, however, only the O_CREAT, O_EXCL, O_EXLOCK, O_NONBLOCK, O_RDONLY, O_RDWR, O_SHLOCK and O_TRUNC flags are meaningful. (Note, opening a database file O_WRONLY is not possible.)

The `type` argument is of type DBTYPE (as defined in the <db.h> include file) and may be set to DB_BTREE, DB_HASH or DB_RECNO.

The *openinfo* argument is a pointer to an access—method—specific structure described in the access method's manual page. If *openinfo* is NULL, each access method will use defaults appropriate for the system and the access method.

The *dbopen* function returns a pointer to a DB structure on success, and NULL on error. The DB structure is defined in the <db.h> include file, and contains at least the following fields:

```
typedef struct {
    DBTYPE  type;
    int  (*close)(const DB* db);
    int  (*del)(const DB* db, const DBT* key, u_int flags);
    int  (*fd)(const DB* db);
    int  (*get)(const DB* db, DBT* key, DBT* data, u_int flags);
    int  (*put)(const DB* db, DBT* key, const DBT* data, u_int flags);
    int  (*sync)(const DB* db, u_int flags);
    int  (*seq)(const DB* db, DBT* key, DBT* data, u_int flags);
} DB;
```

These elements describe a database type and a set of functions performing various actions. These functions take a pointer to a structure as returned by *dbopen*, and sometimes one or more pointers to key / data structures and a flag value.

type The type of the underlying access method (and file format).

close A pointer to a routine to flush any cached information to disk, free any allocated resources, and close the underlying file(s). As key / data pairs may be cached in memory, failing to sync the file with a *close* or `sync` function may result in inconsistent or lost information. The *close* routines return -1 on error (setting *errno*) and 0 on success.

del A pointer to a routine to remove key / data pairs from the database.

The *flag* parameter may be set to the following value:

R_CURSOR	Delete the record referenced by the cursor. The cursor must have previously been initialized.

The *delete* routines return -1 on error (setting *errno*), 0 on success, and 1 if the specified *key* was not in the file.

fd A pointer to a routine which returns a file descriptor representative of the underlying database. A file descriptor referencing the same file will be returned to all processes which call *dbopen* with the same `file` name. This file descriptor may safely be used as an argument to the *fcntl*(2) and *flock*(2) locking functions. The file descriptor is not necessarily associated with any of the underlying files used by the access method. No file descriptor is available for in—memory databases. The *fd* routines return -1 on error (setting *errno*), and the file descriptor on success.

get A pointer to a routine which is the interface for keyed retrieval from the database. The address and length of the data associated with the specified *key* are returned in the structure referenced by *data*. The `get` routines return -1 on error (setting *errno*), 0 on success, and 1 if the *key* was not in the file.

put A pointer to a routine to store key / data pairs in the database.

The *flag* parameter may be set to one of the following values:

R_CURSOR	Replace the key / data pair referenced by the cursor. The cursor must have previously been initialized.
R_IAFTER	Append the data immediately after the data referenced by *key*, creating a new key / data pair. The record number of the appended key / data pair is returned in the *key* structure. (Applicable only to the DB_RECNO access method.)

R_IBEFORE	Insert the data immediately before the data referenced by *key*, creating a new key / data pair. The record number of the inserted key / data pair is returned in the *key* structure. (Applicable only to the DB_RECNO access method.)
R_NOOVERWRITE	Enter the new key / data pair only if the key did not previously exist.
R_SETCURSOR	Store the key / data pair, setting or initializing the position of the cursor to reference it. (Applicable only to the DB_BTREE and DB_RECNO access methods.)

The R_SETCURSOR option is only available for the DB_BTREE and DB_RECNO access methods (because it implies that the keys have an inherent order which does not change).

The R_IAFTER and R_IBEFORE functions are only available for the DB_RECNO access method (because they each imply that the access method is able to create new keys). This is only true if the keys are ordered and independent (record numbers, for example).

The default behavior of the *put* routines is to enter the new key / data pair, replacing any previously existing key.

The *put* routines return -1 on error (setting *errno*), 0 on success, and 1 if the R_NOOVERWRITE *flag* was set and the key already exists in the file.

seq	A pointer to a routine which is the interface for sequential retrieval from the database. The address and length of the key are returned in the structure referenced by *key*, and the address and length of the data are returned in the structure referenced by *data*.

Sequential key / data pair retrieval may begin at any time, and the position of the “cursor” is not affected by calls to the *del*, `get`, *put*, or `sync` routines. Modifications to the database during a sequential scan will be reflected in the scan; records inserted behind the cursor will not be returned, while records inserted in front of the cursor will be returned.

The flag value `must` be set to one of the following values:

R_CURSOR	The data associated with the specified key is returned. This differs from the `get` routines in that it sets or initializes the cursor to the location of the key as well. (Note that for the DB_BTREE

	access method, the key returned is not always an exact match for the key specified. The key returned is the smallest key greater than or equal to the key specified; this provides for partial key matches and range searches).
R_FIRST	The first key / data pair of the database is returned, and the cursor is set or initialized to reference it.
R_LAST	The last key / data pair of the database is returned, and the cursor is set or initialized to reference it. (Applicable only to the DB_BTREE and DB_RECNO access methods.)
R_NEXT	Retrieve the key / data pair immediately after the cursor. If the cursor is not yet set, this is the same as the R_FIRST flag.
R_PREV	Retrieve the key / data pair immediately before the cursor. If the cursor is not yet set, this is the same as the R_LAST flag. (Applicable only to the DB_BTREE and DB_RECNO access methods.)

The R_LAST and R_PREV options are only available for the DB_BTREE and DB_RECNO access methods (because they each imply that the keys have an inherent order which does not change).

The *seq* routines return -1 on error (setting *errno*), 0 on success and 1 if there are no key / data pairs less than or greater than the current or specified key. If the DB_RECNO access method is being used, and if the database file is a character special file, and no complete key / data pairs are currently available, the *seq* routines return 2.

sync A pointer to a routine to flush any cached information to disk. If the database is in memory only, the sync routine has no effect and will always succeed.

The flag value may be set to the following value:

R_RECNOSYNC	If the DB_RECNO access method is being used, this flag causes the sync routine to apply to the btree file which underlies the recno file, not the recno file itself. (See the *bfname* field of the *recno*(3) manual page for more information.)

The sync routines return -1 on error (setting *errno*) and 0 on success.

KEY/DATA PAIRS

Access to all file types is based on key / data pairs. Both keys and data are represented by the following data structure:

```
typedef struct {
   void*   data;
   size_t  size;
} DBT;
```

The elements of the DBT structure are defined as follows:

data	A pointer to a byte string.
size	The length of the byte string.

Key and data byte strings may reference strings of essentially unlimited length, although any two of them must fit into available memory at the same time. It should be noted that the access methods do not provide guarantees regarding byte string alignment.

ERRORS

The *dbopen* routine may fail and set *errno* to any of the errors specified for the library routines *open(2POSIX)* and *malloc(3STDC)* as well as the following:

[EFTYPE]	A file has been incorrectly formatted..
[EINVAL]	A parameter has been specified (hash function, pad byte) that is incompatible with the current file specification, or which is not meaningful for the function (for example, use of the cursor without prior initialization) , or there is a mismatch between the version number of the file and the software.

The *close* routines may fail and set *errno* to any of the errors specified for the library routines *close(2POSIX), read(2POSIX), write(2POSIX), free(3STDC),* or *fsync(2POSIX).*

The *del*, `get`, *put* and *seq* routines may fail and set *errno* to any of the errors specified for the library routines *read(2POSIX), write(2POSIX), free(3STDC)* or *malloc(3STDC).*

The *fd* routines will fail and set *errno* to ENOENT for in—memory databases.

The `sync` routines may fail and set *errno* to any of the errors specified for the library routine *fsync*(2).

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`btree`(3POSIX), `hash`(3POSIX), `mpool`(3POSIX), `recno`(3POSIX)

LIBTP: Portable, Modular Transactions for UNIX, Margo Seltzer, Michael Olson, USENIX proceedings, Winter 1992.

BUGS

The typedef DBT is a mnemonic for “data base thang”, and was used because no one could think of a reasonable name that wasn’t already in use.

The file descriptor interface is a temporary solution, and will be deleted in a future version of the interface.

None of the access methods provide any form of concurrent access, locking, or transactions.

RESTRICTIONS

These library calls do not support multi-threaded applications.

NAME

directory, opendir, readdir, telldir, seekdir, rewinddir, closedir – directory operations

SYNOPSIS

#include <sys/types.h>
#include <dirent.h>
DIR * **opendir**(const char * *filename*);

struct dirent * **readdir**(DIR * *dirp*);

long **telldir**(const DIR * *dirp*);

void **seekdir**(DIR * *dirp*, long *loc*);

void **rewinddir**(DIR * *dirp*);

int **closedir**(DIR * *dirp*);

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

The *opendir* function opens the directory named by *filename* , associates a directory stream with it and returns a pointer to be used to identify the directory stream in subsequent operations. The NULL pointer is returned if *filename* cannot be accessed, or if it cannot *malloc(3STDC)* enough memory to hold all of it.

The *readdir* function returns a pointer to the next directory entry. It returns NULL upon reaching the end of the directory or detecting an invalid *seekdir* operation.

The *telldir* function returns the current location associated with the named directory stream.

The *seekdir* function sets the position of the next *readdir* operation on the directory stream. The new position reverts to the one associated with the directory stream when the *telldir* operation was performed. Values returned by *telldir* are valid only for the lifetime of the DIR pointer, *dirp* , from which they are derived. If the directory is closed and then reopened, the *telldir* value may be invalidated due to undetected directory compaction. It is safe to use a previous *telldir* value immediately after a call to *opendir* and before any calls to *readdir* .

The *rewinddir* function resets the position of the named directory stream to the beginning of the directory.

The *closedir* function closes the named directory stream and frees the structure associated with the *dirp* pointer, returning 0 on success. On failure, –1 is returned and the global variable *errno* is set to indicate the error.

Sample code which searches a directory for the “name” entry is:

```
len = strlen(name);
dirp = opendir(".");
if (dirp) {
   while ((dp = readdir(dirp)) != NULL)
 if (dp->d_namlen == len && !strcmp(dp->d_name, name)) {
  (void) closedir(dirp);
  return FOUND;
         }
   (void) closedir(dirp);
   }
return NOT_FOUND;
```

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`open`(2POSIX) , `close`(2POSIX) , `read`(2POSIX) , `lseek`(2POSIX)

HISTORY

The *opendir*, *readdir*, *telldir*, *seekdir*, *rewinddir* and *closedir* functions appeared in 4.2 BSD.

RESTRICTIONS

These library calls do not support multi-threaded applications.

NAME

getnetent, getnetbyaddr, getnetbyname, setnetent, endnetent – get network entry

SYNOPSIS

#include <netdb.h>
struct netent * **getnetent**(void);

struct netent * **getnetbyname**(char * *name*);

struct netent * **getnetbyaddr**(long *net*, int *type*);

void **setnetent**(int *stayopen*);

void **endnetent**(void);

DESCRIPTION

The *getnetent* , *getnetbyname* , and *getnetbyaddr* functions each return a pointer to an object containing the broken-out fields of a line in the network data base */etc/networks* . The object has the following structure:

```
struct netent {
   char*    n_name;        /* official name of net */
   char**   n_aliases;     /* alias list */
   int      n_addrtype;    /* net number type */
   unsigned long   n_net;  /* net number */
};
```

The members of this structure are:

`n_name`	The official name of the network.
`n_aliases`	A zero terminated list of alternate names for the network.
`n_addrtype`	The type of the network number returned; currently only AF_INET.
`n_net`	The network number. Network numbers are returned in machine byte order.

The *getnetent* function reads the next line of the file, opening the file if necessary.

The *setnetent* function opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to *getnetbyname* or *getnetbyaddr* .

The *endnetent* function closes the file.

The *getnetbyname* and *getnetbyaddr* functions sequentially search from the beginning of the file until a matching net name or net address and type is found, or until EOF is encountered. Network numbers are supplied in host order.

FILES

/etc/networks

DIAGNOSTICS

A null pointer (0) is returned when EOF or an error is encountered.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `networks`(4CC)

BUGS The data space used by these functions is static; if the data will be required in the future, it should be copied before any subsequent calls to these functions overwrite it. Only Internet network numbers are currently understood.

NAME

getnetgrent, innetgr, setnetgrent, endnetgrent – netgroup database operations

SYNOPSIS

int **getnetgrent**(char ** *host*, char ** *user*, char ** *domain*);

int **innetgr**(const char * *netgroup*, const char * *host*, const char * *user*, const char * *domain*);

void **setnetgrent**(const char * *netgroup*);

void **endnetgrent**(void);

DESCRIPTION

These functions operate on the netgroup database file `/etc/netgroup` which is described in *netgroup(4CC)* . The database defines a set of netgroups, each made up of one or more triples:

(host, user, domain)

which define a combination of host, user and domain. Any of the three fields may be specified as “wildcards” which match any string.

The *getnetgrent* function sets the three pointer arguments to the strings of the next member of the current netgroup. If any of the string pointers are (`char` *)0 that field is considered a wildcard.

The *setnetgrent* and *endnetgrent* functions set the current netgroup and terminate the current netgroup, respectively. If *setnetgrent* is called with a different netgroup from the previous call, an *endnetgrent* is implied. The *setnetgrent* function also sets the offset to the first member of the netgroup.

The *innetgr* function searches for a match of all fields within the specified group. If any of the *host* or *domain* arguments are (`char` *)0 those fields will match any string value in the netgroup member.

RETURN VALUES

The *getnetgrent* function returns 0 for “no more netgroup members” and 1 otherwise. The *innetgr* function returns 1 for a successful match and 0 otherwise. The *setnetgrent* and *endnetgrent* functions have no return value.

FILES

`/etc/netgroup` netgroup database file

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`netgroup`(4CC)

COMPATIBILITY

The netgroup members have three string fields to maintain compatibility with other vendor implementations. However, the applicability of the *domain* string within BSD is unclear.

BUGS

The *getnetgrent* function returns pointers to dynamically allocated data areas that are freed when the *endnetgrent* function is called.

RESTRICTIONS

These library calls do not support multi-threaded applications.

NAME

getprotoent, getprotobynumber, getprotobyname, setprotoent, endprotoent – get protocol entry

SYNOPSIS

#include <netdb.h>
struct protoent * **getprotoent**(void);

struct protoent * **getprotobyname**(char * *name*);

structprotoent ***getprotobynumber**(int *proto*);

void **setprotoent**(int *stayopen*);

void **endprotoent**(void);

DESCRIPTION

The `getprotoent()`, `getprotobyname()`, and `getprotobynumber()` functions each return a pointer to an object containing the broken-out fields of a line in the network protocol data base, `/etc/protocols`. The object has the following structure:

```
struct protoent {
   char*    p_name;      /* official name of protocol */
   char**   p_aliases;   /* alias list */
   int      p_proto;     /* protocol number */
};
```

The members of this structure are:

`p_name` The official name of the protocol.

`p_aliases` A zero terminated list of alternate names for the protocol.

`p_proto` The protocol number.

The `getprotoent()` function reads the next line of the file, opening the file if necessary.

The `setprotoent()` function opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to `getprotobyname()` or `getprotobynumber()`.

The `endprotoent()` function closes the file.

The `getprotobyname()` and `getprotobynumber()` functions sequentially search from the beginning of the file until a matching protocol name or protocol number is found, or until EOF is encountered (see RESTRICTIONS).

RETURN VALUES

A `NULL` pointer (`0`) is returned when `EOF` or an error is encountered.

FILES

`/etc/protocols`

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`protocols`(4CC)

BUGS

These functions use a static data space; if the data is needed for future use, it should be copied before any subsequent calls overwrite it. Only the Internet protocols are currently understood.

RESTRICTIONS

The `getprotobynumber()` function is not yet implemented in ChorusOS 4.0.

NAME

getservent, getservbyname, getservbyport, setservent, endservent – get service entry

#include <netdb.h>
struct servent * **getservent**(void);

struct servent * **getservbyname**(const char * *name*, const char * *proto*);

struct servent * **getservbyport**(int *port*, const char * *proto*);

void **setservent**(int *stayopen*);

void **endservent**(void);

DESCRIPTION

The `getservent()` , `getservbyname()` , `getservbyport()` functions each return a pointer to an object with the following structure containing the broken-out fields of a line in the network services data base, `/etc/services` .

```
struct  servent {
        char    *s_name;        /* official name of service */
        char    **s_aliases;    /* alias list */
        int     s_port;         /* port service resides at */
        char    *s_proto;       /* protocol to use */
};
```

The members of this structure are:

s_name	The official name of the service.
s_aliases	A zero-terminated list of alternate names for the service.
s_port	The port number at which the service resides. Port numbers are returned in network byte order.
s_proto	The name of the protocol to use when contacting the service.

The `getservent()` function reads the next line of the file, opening the file if necessary.

The `setservent()` function opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to `getservbyname()` or `getservbyport()` .

The `endservent()` function closes the file.

The `getservbyname()` and `getservbyport()` functions sequentially search from the beginning of the file until a matching protocol name or port number is found, or until `EOF` is encountered. If a protocol name is also supplied (non-`NULL`), searches must also match the protocol.

FILES

`/etc/services`

DIAGNOSTICS

Null pointer (`0`) returned on `EOF` or error.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`getprotoent`(3POSIX) , `services`(4CC)

NAME

err, verr, verrx, warn, vwarn, warnx, vwarnx – formatted error messages

SYNOPSIS

#include <err.h>
void **err**(int *eval*, const char * *fmt*, ...);

void **verr**(int *eval*, const char * *fmt*, va_list *args*);

void **errx**(int *eval*, const char * *fmt*, ...);

void **verrx**(int *eval*, const char * *fmt*, va_list *args*);

void **warn**(const char * *fmt*, ...);

void **vwarn**(const char * *fmt*, va_list *args*);

void **warnx**(const char * *fmt*, ...);

void **vwarnx**(const char * *fmt*, va_list *args*);

DESCRIPTION

The *err* and *warn* family of functions display a formatted error message to the standard error output. If the *fmt* argument is not NULL , the formatted error message, a colon character, and a space are output. In the case of the *err* , *verr* , *warn* , and *vwarn* functions, the error message string affiliated with the current value of the global variable *errno* is output. In all cases, the output is followed by a newline character.

The *err* , *verr* , *errx* , and *verrx* functions do not return, but exit with the value of the argument *eval* .

EXAMPLES

Display the current errno information string and exit:

```
if ((p = malloc(size)) == NULL)
        err(1, NULL);
if ((fd = open(file_name, O_RDONLY, 0)) == -1)
        err(1, "%s", file_name);
```

Display an error message and exit:

```
if (tm.tm_hour < START_TIME)
        errx(1, "too early, wait until %s", start_time_string);
```

Warn of an error:

```
if ((fd = open(raw_device, O_RDONLY, 0)) == -1)
        warnx("%s: %s: trying the block device",
            raw_device, strerror(errno));
if ((fd = open(block_device, O_RDONLY, 0)) == -1)
        warn("%s", block_device);
```

HISTORY

The *err* and *warn* functions appeared in 4.4 BSD.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`strerror`(3STDC)

NAME

getcwd, getwd – get working directory pathname

SYNOPSIS

```
#include <stdio.h>
char * getcwd(char * buf, size_t size);

char * getwd(char * buf);
```

DESCRIPTION

The *getcwd* function copies the absolute pathname of the current working directory into the memory referenced by *buf* and returns a pointer to *buf* . The `size` argument is the size, in bytes, of the array referenced by *buf* .

If *buf* is `NULL` , space is allocated as necessary to store the pathname. This space may later be freed.

The function *getwd* is a compatibility routine which calls *getcwd* with its *buf* argument and a size of `MAXPATHLEN` (as defined in the include file `sys/param.h`).The *buf* argument should be at least `MAXPATHLEN` bytes in length.

These routines have traditionally been used by programs to save the name of a working directory for the purpose of returning to it. A much faster and less error-prone method of accomplishing this is to open the current directory and use the *fchdir(2POSIX)* function to return.

RETURN VALUES

Upon successful completion, a pointer to the pathname is returned. Otherwise a `NULL` pointer is returned and the global variable *errno* is set to indicate the error. In addition, *getwd* copies the error message associated with *errno* into the memory referenced by *buf* .

ERRORS

The following error messages are returned by *getcwd* :

[EACCES]	Read or search permission was denied for a component of the pathname.
[EINVAL]	The `size` argument is zero.
[ENOENT]	A component of the pathname no longer exists.
[ENOMEM]	There is insufficient memory available.
[ERANGE]	The `size` argument is greater than zero but smaller than the length of the pathname plus 1.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`chdir`(2POSIX) , `fchdir`(2POSIX) , `malloc`(3STDC) , `strerror`(3STDC)

STANDARDS

The *getcwd* function conforms to *ANSI C* . The ability to specify a `NULL` pointer and have *getcwd* allocate memory as necessary is an extension.

HISTORY

The *getwd* function appeared in 4.0 BSD

BUGS

The *getwd* function does not do sufficient error checking and is therefore not able to return very long, but valid, paths. It is provided for compatibility purposes.

RESTRICTIONS

These library calls do not support multi-threaded applications.

NAME

getdiskbyname – get generic disk description by its name

SYNOPSIS

```
#include <sys/types.h>
#include <sys/disklabel.h>
struct disklabel *getdiskbyname(const char *name);
```

FEATURES

UFS

DESCRIPTION

The *getdiskbyname* function takes a disk name (for example,. rm03) and returns a prototype disk label describing its geometry information and the standard disk partition tables. All information is obtained from the *disktab(4CC)* file.

RETURN VALUES

getdiskbyname returns a null pointer if the entry is not found in the current disktab file.

setdisktab returns 0 on success and -1 if name is a null pointer or points to an empty string.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`disklabel`(4CC), `disktab`(4CC), `disklabel`(1M)

HISTORY

The *getdiskbyname* function appeared in 4.3 BSD.

RESTRICTIONS

This library call does not support multi-threaded applications.

NAME

getmntinfo – get information about mounted file systems

SYNOPSIS

```
#include <sys/param.h>
#include <sys/ucred.h>
#include <sys/mount.h>
int getmntinfo(struct statfs **mntbufp, int flags);
```

DESCRIPTION

The *getmntinfo* function returns an array of *statfs* structures describing each file system currently mounted (see *stafs*(2POSIX))

The *getmntinfo* function passes its *flags* parameter transparently to *getfsstat(2POSIX).*

RETURN VALUES

On successful completion, *getmntinfo* returns a count of the number of elements in the array. The pointer to the array is stored in *mntbufp*.

If an error occurs, zero is returned and the external variable *errno* is set to indicate the error. Although the *mntbufp* pointer will be unmodified, any information previously returned by *getmntinfo* will be lost.

ERRORS

The *getmntinfo* function may fail and set errno to any of the errors specified for the *getfsstat(2POSIX)* or *malloc(3STDC)* library routines.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`getfsstat`(2POSIX), `statfs`(2POSIX), `mount`(2POSIX)

HISTORY

The *getmntinfo* function first appeared in 4.4 BSD.

BUGS

The *getmntinfo* function writes the array of structures to an internal static object and returns a pointer to that object. Subsequent calls to *getmntinfo* will modify that object.

The memory allocated by *getmntinfo* cannot be freed by the application.

RESTRICTIONS

This library call does not support multi-threaded applications.

NAME

getnetent, getnetbyaddr, getnetbyname, setnetent, endnetent – get network entry

SYNOPSIS

#include <netdb.h>
struct netent * **getnetent**(void);

struct netent * **getnetbyname**(char * *name*);

struct netent * **getnetbyaddr**(long *net*, int *type*);

void **setnetent**(int *stayopen*);

void **endnetent**(void);

DESCRIPTION

The *getnetent* , *getnetbyname* , and *getnetbyaddr* functions each return a pointer to an object containing the broken-out fields of a line in the network data base */etc/networks* . The object has the following structure:

```
struct netent {
   char*    n_name;        /* official name of net */
   char**   n_aliases;     /* alias list */
   int      n_addrtype;    /* net number type */
   unsigned long   n_net;  /* net number */
};
```

The members of this structure are:

`n_name` The official name of the network.

`n_aliases` A zero terminated list of alternate names for the network.

`n_addrtype` The type of the network number returned; currently only AF_INET.

`n_net` The network number. Network numbers are returned in machine byte order.

The *getnetent* function reads the next line of the file, opening the file if necessary.

The *setnetent* function opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to *getnetbyname* or *getnetbyaddr* .

The *endnetent* function closes the file.

The *getnetbyname* and *getnetbyaddr* functions sequentially search from the beginning of the file until a matching net name or net address and type is found, or until EOF is encountered. Network numbers are supplied in host order.

FILES

/etc/networks

DIAGNOSTICS

A null pointer (0) is returned when EOF or an error is encountered.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `networks`(4CC)

BUGS The data space used by these functions is static; if the data will be required in the future, it should be copied before any subsequent calls to these functions overwrite it. Only Internet network numbers are currently understood.

NAME

getnetent, getnetbyaddr, getnetbyname, setnetent, endnetent – get network entry

SYNOPSIS

#include <netdb.h>
struct netent * **getnetent**(void);

struct netent * **getnetbyname**(char * *name*);

struct netent * **getnetbyaddr**(long *net*, int *type*);

void **setnetent**(int *stayopen*);

void **endnetent**(void);

DESCRIPTION

The *getnetent* , *getnetbyname* , and *getnetbyaddr* functions each return a pointer to an object containing the broken-out fields of a line in the network data base */etc/networks* . The object has the following structure:

```
struct netent {
   char*    n_name;        /* official name of net */
   char**   n_aliases;     /* alias list */
   int      n_addrtype;    /* net number type */
   unsigned long   n_net;  /* net number */
};
```

The members of this structure are:

`n_name` The official name of the network.

`n_aliases` A zero terminated list of alternate names for the network.

`n_addrtype` The type of the network number returned; currently only AF_INET.

`n_net` The network number. Network numbers are returned in machine byte order.

The *getnetent* function reads the next line of the file, opening the file if necessary.

The *setnetent* function opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to *getnetbyname* or *getnetbyaddr* .

The *endnetent* function closes the file.

The *getnetbyname* and *getnetbyaddr* functions sequentially search from the beginning of the file until a matching net name or net address and type is found, or until EOF is encountered. Network numbers are supplied in host order.

FILES

/etc/networks

DIAGNOSTICS

A null pointer (0) is returned when EOF or an error is encountered.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `networks`(4CC)

BUGS The data space used by these functions is static; if the data will be required in the future, it should be copied before any subsequent calls to these functions overwrite it. Only Internet network numbers are currently understood.

NAME

getnetent, getnetbyaddr, getnetbyname, setnetent, endnetent – get network entry

SYNOPSIS

#include <netdb.h>
struct netent * **getnetent**(void);

struct netent * **getnetbyname**(char * *name*);

struct netent * **getnetbyaddr**(long *net*, int *type*);

void **setnetent**(int *stayopen*);

void **endnetent**(void);

DESCRIPTION

The *getnetent* , *getnetbyname* , and *getnetbyaddr* functions each return a pointer to an object containing the broken-out fields of a line in the network data base */etc/networks* . The object has the following structure:

```
struct netent {
   char*    n_name;        /* official name of net */
   char**   n_aliases;     /* alias list */
   int      n_addrtype;    /* net number type */
   unsigned long   n_net;  /* net number */
};
```

The members of this structure are:

`n_name` The official name of the network.

`n_aliases` A zero terminated list of alternate names for the network.

`n_addrtype` The type of the network number returned; currently only AF_INET.

`n_net` The network number. Network numbers are returned in machine byte order.

The *getnetent* function reads the next line of the file, opening the file if necessary.

The *setnetent* function opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to *getnetbyname* or *getnetbyaddr* .

The *endnetent* function closes the file.

The *getnetbyname* and *getnetbyaddr* functions sequentially search from the beginning of the file until a matching net name or net address and type is found, or until EOF is encountered. Network numbers are supplied in host order.

FILES

/etc/networks

DIAGNOSTICS

A null pointer (0) is returned when EOF or an error is encountered.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `networks`(4CC)

BUGS The data space used by these functions is static; if the data will be required in the future, it should be copied before any subsequent calls to these functions overwrite it. Only Internet network numbers are currently understood.

NAME

getnetgrent, innetgr, setnetgrent, endnetgrent – netgroup database operations

SYNOPSIS

int **getnetgrent**(char ** *host*, char ** *user*, char ** *domain*);

int **innetgr**(const char * *netgroup*, const char * *host*, const char * *user*, const char * *domain*);

void **setnetgrent**(const char * *netgroup*);

void **endnetgrent**(void);

DESCRIPTION

These functions operate on the netgroup database file `/etc/netgroup` which is described in *netgroup(4CC)* . The database defines a set of netgroups, each made up of one or more triples:

(host, user, domain)

which define a combination of host, user and domain. Any of the three fields may be specified as "wildcards" which match any string.

The *getnetgrent* function sets the three pointer arguments to the strings of the next member of the current netgroup. If any of the string pointers are (`char` *)0 that field is considered a wildcard.

The *setnetgrent* and *endnetgrent* functions set the current netgroup and terminate the current netgroup, respectively. If *setnetgrent* is called with a different netgroup from the previous call, an *endnetgrent* is implied. The *setnetgrent* function also sets the offset to the first member of the netgroup.

The *innetgr* function searches for a match of all fields within the specified group. If any of the *host* or *domain* arguments are (`char` *)0 those fields will match any string value in the netgroup member.

RETURN VALUES

The *getnetgrent* function returns 0 for "no more netgroup members" and 1 otherwise. The *innetgr* function returns 1 for a successful match and 0 otherwise. The *setnetgrent* and *endnetgrent* functions have no return value.

FILES

`/etc/netgroup` netgroup database file

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`netgroup`(4CC)

COMPATIBILITY

The netgroup members have three string fields to maintain compatibility with other vendor implementations. However, the applicability of the *domain* string within BSD is unclear.

BUGS

The *getnetgrent* function returns pointers to dynamically allocated data areas that are freed when the *endnetgrent* function is called.

RESTRICTIONS

These library calls do not support multi-threaded applications.

NAME

getprotoent, getprotobynumber, getprotobyname, setprotoent, endprotoent – get protocol entry

SYNOPSIS

#include <netdb.h>
struct protoent * **getprotoent**(void);

struct protoent * **getprotobyname**(char * *name*);

structprotoent ***getprotobynumber**(int *proto*);

void **setprotoent**(int *stayopen*);

void **endprotoent**(void);

DESCRIPTION

The `getprotoent()` , `getprotobyname()` , and `getprotobynumber()` functions each return a pointer to an object containing the broken-out fields of a line in the network protocol data base, `/etc/protocols` . The object has the following structure:

```
struct protoent {
   char*    p_name;      /* official name of protocol */
   char**   p_aliases;   /* alias list */
   int      p_proto;     /* protocol number */
};
```

The members of this structure are:

`p_name` The official name of the protocol.

`p_aliases` A zero terminated list of alternate names for the protocol.

`p_proto` The protocol number.

The `getprotoent()` function reads the next line of the file, opening the file if necessary.

The `setprotoent()` function opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to `getprotobyname()` or `getprotobynumber()` .

The `endprotoent()` function closes the file.

The `getprotobyname()` and `getprotobynumber()` functions sequentially search from the beginning of the file until a matching protocol name or protocol number is found, or until EOF is encountered (see RESTRICTIONS).

RETURN VALUES

A `NULL` pointer (`0`) is returned when `EOF` or an error is encountered.

FILES

`/etc/protocols`

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `protocols`(4CC)

BUGS These functions use a static data space; if the data is needed for future use, it should be copied before any subsequent calls overwrite it. Only the Internet protocols are currently understood.

RESTRICTIONS The `getprotobynumber()` function is not yet implemented in ChorusOS 4.0.

NAME

getprotoent, getprotobynumber, getprotobyname, setprotoent, endprotoent – get protocol entry

SYNOPSIS

#include <netdb.h>
struct protoent * **getprotoent**(void);

struct protoent * **getprotobyname**(char * *name*);

structprotoent ***getprotobynumber**(int *proto*);

void **setprotoent**(int *stayopen*);

void **endprotoent**(void);

DESCRIPTION

The `getprotoent()` , `getprotobyname()` , and `getprotobynumber()` functions each return a pointer to an object containing the broken-out fields of a line in the network protocol data base, `/etc/protocols` . The object has the following structure:

```
struct protoent {
   char*    p_name;      /* official name of protocol */
   char**   p_aliases;   /* alias list */
   int      p_proto;     /* protocol number */
};
```

The members of this structure are:

`p_name` The official name of the protocol.

`p_aliases` A zero terminated list of alternate names for the protocol.

`p_proto` The protocol number.

The `getprotoent()` function reads the next line of the file, opening the file if necessary.

The `setprotoent()` function opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to `getprotobyname()` or `getprotobynumber()` .

The `endprotoent()` function closes the file.

The `getprotobyname()` and `getprotobynumber()` functions sequentially search from the beginning of the file until a matching protocol name or protocol number is found, or until EOF is encountered (see RESTRICTIONS).

RETURN VALUES

A `NULL` pointer (`0`) is returned when `EOF` or an error is encountered.

FILES

`/etc/protocols`

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`protocols`(4CC)

BUGS

These functions use a static data space; if the data is needed for future use, it should be copied before any subsequent calls overwrite it. Only the Internet protocols are currently understood.

RESTRICTIONS

The `getprotobynumber()` function is not yet implemented in ChorusOS 4.0.

NAME

getprotoent, getprotobynumber, getprotobyname, setprotoent, endprotoent – get protocol entry

SYNOPSIS

#include <netdb.h>
struct protoent * **getprotoent**(void);

struct protoent * **getprotobyname**(char * *name*);

structprotoent ***getprotobynumber**(int *proto*);

void **setprotoent**(int *stayopen*);

void **endprotoent**(void);

DESCRIPTION

The `getprotoent()` , `getprotobyname()` , and `getprotobynumber()` functions each return a pointer to an object containing the broken-out fields of a line in the network protocol data base, `/etc/protocols` . The object has the following structure:

```
struct protoent {
   char*    p_name;      /* official name of protocol */
   char**   p_aliases;   /* alias list */
   int      p_proto;     /* protocol number */
};
```

The members of this structure are:

`p_name` The official name of the protocol.

`p_aliases` A zero terminated list of alternate names for the protocol.

`p_proto` The protocol number.

The `getprotoent()` function reads the next line of the file, opening the file if necessary.

The `setprotoent()` function opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to `getprotobyname()` or `getprotobynumber()` .

The `endprotoent()` function closes the file.

The `getprotobyname()` and `getprotobynumber()` functions sequentially search from the beginning of the file until a matching protocol name or protocol number is found, or until EOF is encountered (see RESTRICTIONS).

RETURN VALUES

A `NULL` pointer (`0`) is returned when `EOF` or an error is encountered.

FILES

`/etc/protocols`

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `protocols`(4CC)

BUGS These functions use a static data space; if the data is needed for future use, it should be copied before any subsequent calls overwrite it. Only the Internet protocols are currently understood.

RESTRICTIONS The `getprotobynumber()` function is not yet implemented in ChorusOS 4.0.

NAME

getservent, getservbyname, getservbyport, setservent, endservent – get service entry

#include <netdb.h>
struct servent * **getservent**(void);

struct servent * **getservbyname**(const char * *name*, const char * *proto*);

struct servent * **getservbyport**(int *port*, const char * *proto*);

void **setservent**(int *stayopen*);

void **endservent**(void);

DESCRIPTION

The `getservent()` , `getservbyname()` , `getservbyport()` functions each return a pointer to an object with the following structure containing the broken-out fields of a line in the network services data base, `/etc/services` .

```
struct  servent {
        char    *s_name;        /* official name of service */
        char    **s_aliases;    /* alias list */
        int     s_port;         /* port service resides at */
        char    *s_proto;       /* protocol to use */
};
```

The members of this structure are:

s_name	The official name of the service.
s_aliases	A zero-terminated list of alternate names for the service.
s_port	The port number at which the service resides. Port numbers are returned in network byte order.
s_proto	The name of the protocol to use when contacting the service.

The `getservent()` function reads the next line of the file, opening the file if necessary.

The `setservent()` function opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to `getservbyname()` or `getservbyport()` .

The `endservent()` function closes the file.

The `getservbyname()` and `getservbyport()` functions sequentially search from the beginning of the file until a matching protocol name or port number is found, or until `EOF` is encountered. If a protocol name is also supplied (non-`NULL`), searches must also match the protocol.

FILES

`/etc/services`

DIAGNOSTICS

Null pointer (`0`) returned on `EOF` or error.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`getprotoent`(3POSIX) , `services`(4CC)

NAME

getservent, getservbyname, getservbyport, setservent, endservent – get service entry

#include <netdb.h>
struct servent * **getservent**(void);

struct servent * **getservbyname**(const char * *name*, const char * *proto*);

struct servent * **getservbyport**(int *port*, const char * *proto*);

void **setservent**(int *stayopen*);

void **endservent**(void);

DESCRIPTION

The `getservent()` , `getservbyname()` , `getservbyport()` functions each return a pointer to an object with the following structure containing the broken-out fields of a line in the network services data base, `/etc/services` .

```
struct  servent {
        char    *s_name;        /* official name of service */
        char    **s_aliases;    /* alias list */
        int     s_port;         /* port service resides at */
        char    *s_proto;       /* protocol to use */
};
```

The members of this structure are:

s_name	The official name of the service.
s_aliases	A zero-terminated list of alternate names for the service.
s_port	The port number at which the service resides. Port numbers are returned in network byte order.
s_proto	The name of the protocol to use when contacting the service.

The `getservent()` function reads the next line of the file, opening the file if necessary.

The `setservent()` function opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to `getservbyname()` or `getservbyport()` .

The `endservent()` function closes the file.

The `getservbyname()` and `getservbyport()` functions sequentially search from the beginning of the file until a matching protocol name or port number is found, or until `EOF` is encountered. If a protocol name is also supplied (non-`NULL`), searches must also match the protocol.

FILES

`/etc/services`

DIAGNOSTICS

Null pointer (`0`) returned on `EOF` or error.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`getprotoent`(3POSIX) , `services`(4CC)

NAME

getservent, getservbyname, getservbyport, setservent, endservent – get service entry

SYNOPSIS

#include <netdb.h>
struct servent * **getservent**(void);

struct servent * **getservbyname**(const char * *name*, const char * *proto*);

struct servent * **getservbyport**(int *port*, const char * *proto*);

void **setservent**(int *stayopen*);

void **endservent**(void);

DESCRIPTION

The `getservent()` , `getservbyname()` , `getservbyport()` functions each return a pointer to an object with the following structure containing the broken-out fields of a line in the network services data base, `/etc/services` .

```
struct  servent {
        char    *s_name;        /* official name of service */
        char    **s_aliases;    /* alias list */
        int     s_port;         /* port service resides at */
        char    *s_proto;       /* protocol to use */
};
```

The members of this structure are:

s_name	The official name of the service.
s_aliases	A zero-terminated list of alternate names for the service.
s_port	The port number at which the service resides. Port numbers are returned in network byte order.
s_proto	The name of the protocol to use when contacting the service.

The `getservent()` function reads the next line of the file, opening the file if necessary.

The `setservent()` function opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to `getservbyname()` or `getservbyport()` .

The `endservent()` function closes the file.

The `getservbyname()` and `getservbyport()` functions sequentially search from the beginning of the file until a matching protocol name or port number is found, or until `EOF` is encountered. If a protocol name is also supplied (non-`NULL`), searches must also match the protocol.

FILES

`/etc/services`

DIAGNOSTICS

Null pointer (`0`) returned on `EOF` or error.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`getprotoent`(3POSIX) , `services`(4CC)

NAME

getcwd, getwd – get working directory pathname

SYNOPSIS

```
#include <stdio.h>
char * getcwd(char * buf, size_t size);

char * getwd(char * buf);
```

DESCRIPTION

The *getcwd* function copies the absolute pathname of the current working directory into the memory referenced by *buf* and returns a pointer to *buf* . The `size` argument is the size, in bytes, of the array referenced by *buf* .

If *buf* is `NULL` , space is allocated as necessary to store the pathname. This space may later be freed.

The function *getwd* is a compatibility routine which calls *getcwd* with its *buf* argument and a size of `MAXPATHLEN` (as defined in the include file `sys/param.h`).The *buf* argument should be at least `MAXPATHLEN` bytes in length.

These routines have traditionally been used by programs to save the name of a working directory for the purpose of returning to it. A much faster and less error-prone method of accomplishing this is to open the current directory and use the *fchdir(2POSIX)* function to return.

RETURN VALUES

Upon successful completion, a pointer to the pathname is returned. Otherwise a `NULL` pointer is returned and the global variable *errno* is set to indicate the error. In addition, *getwd* copies the error message associated with *errno* into the memory referenced by *buf* .

ERRORS

The following error messages are returned by *getcwd* :

[EACCES]	Read or search permission was denied for a component of the pathname.
[EINVAL]	The `size` argument is zero.
[ENOENT]	A component of the pathname no longer exists.
[ENOMEM]	There is insufficient memory available.
[ERANGE]	The `size` argument is greater than zero but smaller than the length of the pathname plus 1.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`chdir`(2POSIX) , `fchdir`(2POSIX) , `malloc`(3STDC) , `strerror`(3STDC)

STANDARDS

The *getcwd* function conforms to *ANSI C* . The ability to specify a `NULL` pointer and have *getcwd* allocate memory as necessary is an extension.

HISTORY

The *getwd* function appeared in 4.0 BSD

BUGS

The *getwd* function does not do sufficient error checking and is therefore not able to return very long, but valid, paths. It is provided for compatibility purposes.

RESTRICTIONS

These library calls do not support multi-threaded applications.

NAME

glob, globfree – generate pathnames matching a pattern

SYNOPSIS

#include <glob.h>

int **glob**(const char * *pattern*, int *flags*, const int (* *errfunc*) (const char *, int), glob_t * *pglob*);

void **globfree**(glob_t * *pglob*);

DESCRIPTION

The `glob` function is a pathname generator that implements the rules for file name pattern matching used by the shell.

The include file `glob.h` defines the structure type *glob_t* , which contains at least the following fields:

```
typedef struct {
    int     gl_pathc;       /* count of total paths so far */
    int     gl_matchc;      /* count of paths matching pattern */
    int     gl_offs;        /* reserved at beginning of gl_pathv */
    int     gl_flags;       /* returned flags */
    char    **gl_pathv;     /* list of paths matching pattern */
} glob_t;
```

The *pattern* argument is a pointer to a pathname pattern to be expanded. The `glob` argument matches all accessible pathnames against the pattern and creates a list of the pathnames that match. In order to have access to a pathname, `glob` requires search permission on every component of a path except the last, and read permission on each directory of any filename component of *pattern* that contains any of the special characters "*" , "?" or "[".

The `glob` argument stores the number of matched pathnames in the *gl_pathc* field, and a pointer to a list of pointers to pathnames in the *gl_pathv* field. The first pointer after the last pathname is `NULL` . If the pattern does not match any pathnames, the number of matched paths returned is set to zero.

It is the caller's responsibility to create the structure pointed to by *pglob* . The `glob` function allocates other space as needed, including the memory pointed to by *gl_pathv* .

The *flags* argument is used to modify the behavior of `glob` . The value of *flags* is the bitwise inclusive `OR` of any of the following values defined in `glob.h` :

GLOB_APPEND	Append pathnames generated to any from a previous call (or calls) to `glob` . The value of *gl_pathc* will be the total matches found by this call and the previous call(s). The pathnames are appended to, not merged with, the pathnames returned by the previous call(s). Between calls, the caller must not change the setting of the GLOB_DOOFFS flag, nor change the value

	of *gl_offs* when GLOB_DOOFFS is set, nor call *globfree* for *pglob* .
GLOB_DOOFFS	Use the *gl_offs* field. If this flag is set, *gl_offs* is used to specify how many `NULL` pointers to prepend to the beginning of the *gl_pathv* field. In other words, *gl_pathv* will point to *gl_offs* `NULL` pointers, followed by *gl_pathc* pathname pointers, followed by a `NULL` pointer.
GLOB_ERR	Causes `glob` to return when it encounters a directory that it cannot open or read. Ordinarily, `glob` continues to find matches.
GLOB_MARK	Each pathname that is a directory that matches *pattern* has a slash appended.
GLOB_NOCHECK	If *pattern* does not match any pathname, `glob` returns a list consisting of only *pattern* with the total number of pathnames set to 1, and the number of matched pathnames set to 0. If `GLOB_QUOTE` is set, its effect is present in the pattern returned.
GLOB_NOSORT	By default, the pathnames are sorted in ascending `ASCII` order; this flag prevents that sorting (speeding up `glob`) .

The following values may also be included in *flags* , however, they are non-standard extensions to POSIX 1003.2 .

GLOB_ALTDIRFUNC	The following additional fields in the pglob structure have been initialized with alternate functions for glob to use to open, read, and close directories and to get stat information on names found in those directories.

```
void*            (*gl_opendir)(const char* name);
struct dirent*   (*gl_readdir)(void*);
void             (*gl_closedir)(void*);
int              (*gl_lstat)(const char* name, struct stat* st);
int              (*gl_stat)(const char* name, struct stat* st);
```

GLOB_BRACE	Pre-process the pattern string to expand {pat,pat,.... The "{" pattern is left unexpanded for historical reasons.

GLOB_MAGCHAR	Set by the `glob` function if the pattern included globbing characters. See the description of the usage of the *gl_matchc* structure member for more details.
GLOB_NOMAGIC	Is the same as `GLOB_NOCHECK` but it only appends the *pattern* if it does not contain any of the special characters "*", "?" or "[".
GLOB_QUOTE	Use the backslash" \\" character for quoting: every occurrence of a backslash followed by a character in the pattern. This will be replaced by that character, avoiding any special interpretation of the character.
GLOB_TILDE	Expand patterns that start with a tilde ("~") to user name home directories.

If, during the search, a directory is encountered that cannot be opened or read and *errfunc* is non-NULL , `glob` calls *(*errfunc)(path, errno)* . This may be an unintuitive: pattern like */Makefile will try to *stat(2POSIX)* foo/Makefile even if foo is not a directory, resulting in a call to *errfunc* . The error routine can suppress this action by testing for `ENOENT` and `ENOTDIR` ; however, the `GLOB_ERR` flag will still cause an immediate return when this happens.

If *errfunc* returns non-zero, `glob` stops the scan and returns `GLOB_ABEND` after setting *gl_pathc* and *gl_pathv* to reflect any paths already matched. This also happens if an error is encountered and `GLOB_ERR` is set in *flags* , regardless of the return value of *errfunc* , if called. If `GLOB_ERR` is not set and either *errfunc* is `NULL` or *errfunc* returns zero, the error is ignored.

The *globfree* function frees any space associated with *pglob* from any previous call(s) to `glob` .

RETURN VALUES

On successful completion, `glob` returns zero. In addition the fields of *pglob* contain the values described below:

gl_pathc	contains the total number of pathnames matched so far. This includes other matches from previous invocations of `glob` if `GLOB_APPEND` was specified.
gl_matchc	contains the number of matched pathnames in the current invocation of `glob` .
gl_flags	contains a copy of the *flags* parameter with the `GLOB_MAGCHAR` bit set if *pattern* contained any

	of the special characters "*", "?" or "[". If not, it is cleared.
gl_pathv	contains a pointer to a `NULL` terminated list of matched pathnames. However, if *gl_pathc* is zero, the contents of *gl_pathv* are undefined.

If `glob` terminates due to an error, it sets errno and returns one of the following non-zero constants, which are defined in the include file `glob.h` :

GLOB_NOSPACE	An attempt to allocate memory failed.
GLOB_ABEND	The scan was stopped because an error was encountered and either `GLOB_ERR` was set or *(*errfunc)()* returned a non-zero value.

The arguments *pglob–>gl_pathc* and *pglob–>gl_pathv* are still set as specified above.

STANDARDS

The `glob` function is POSIX 1003.2 compatible with the exception that the flags `GLOB_ALTDIRFUNC` , `GLOB_BRACE` , `GLOB_MAGCHAR` , `GLOB_NOMAGIC` , `GLOB_QUOTE` and `GLOB_TILDE` and the fields *gl_matchc* and *gl_flags* should not be used by applications which require strict *POSIX* conformance.

HISTORY

The `glob` and *globfree* functions first appeared in 4.4BSD.

BUGS

Patterns longer than `MAXPATHLEN` may cause unchecked errors.

The `glob` argument may fail and set errno for any of the errors specified for the library routines stat(2POSIX), closedir(3POSIX), opendir(3POSIX), readdir(3POSIX), malloc(3STDC), and free(3STDC).

RESTRICTIONS FOR ChorusOS

These library calls do not support multi-threaded applications.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

glob, globfree – generate pathnames matching a pattern

SYNOPSIS

#include <glob.h>

int **glob**(const char * *pattern*, int *flags*, const int (* *errfunc*) (const char *, int), glob_t * *pglob*);

void **globfree**(glob_t * *pglob*);

DESCRIPTION

The glob function is a pathname generator that implements the rules for file name pattern matching used by the shell.

The include file glob.h defines the structure type *glob_t* , which contains at least the following fields:

```
typedef struct {
    int     gl_pathc;       /* count of total paths so far */
    int     gl_matchc;      /* count of paths matching pattern */
    int     gl_offs;        /* reserved at beginning of gl_pathv */
    int     gl_flags;       /* returned flags */
    char    **gl_pathv;     /* list of paths matching pattern */
} glob_t;
```

The *pattern* argument is a pointer to a pathname pattern to be expanded. The glob argument matches all accessible pathnames against the pattern and creates a list of the pathnames that match. In order to have access to a pathname, glob requires search permission on every component of a path except the last, and read permission on each directory of any filename component of *pattern* that contains any of the special characters "*" , "?" or "[".

The glob argument stores the number of matched pathnames in the *gl_pathc* field, and a pointer to a list of pointers to pathnames in the *gl_pathv* field. The first pointer after the last pathname is NULL . If the pattern does not match any pathnames, the number of matched paths returned is set to zero.

It is the caller’s responsibility to create the structure pointed to by *pglob* . The glob function allocates other space as needed, including the memory pointed to by *gl_pathv* .

The *flags* argument is used to modify the behavior of glob . The value of *flags* is the bitwise inclusive OR of any of the following values defined in glob.h :

GLOB_APPEND
: Append pathnames generated to any from a previous call (or calls) to glob . The value of *gl_pathc* will be the total matches found by this call and the previous call(s). The pathnames are appended to, not merged with, the pathnames returned by the previous call(s). Between calls, the caller must not change the setting of the GLOB_DOOFFS flag, nor change the value

	of *gl_offs* when GLOB_DOOFFS is set, nor call *globfree* for *pglob* .
GLOB_DOOFFS	Use the *gl_offs* field. If this flag is set, *gl_offs* is used to specify how many `NULL` pointers to prepend to the beginning of the *gl_pathv* field. In other words, *gl_pathv* will point to *gl_offs* `NULL` pointers, followed by *gl_pathc* pathname pointers, followed by a `NULL` pointer.
GLOB_ERR	Causes `glob` to return when it encounters a directory that it cannot open or read. Ordinarily, `glob` continues to find matches.
GLOB_MARK	Each pathname that is a directory that matches *pattern* has a slash appended.
GLOB_NOCHECK	If *pattern* does not match any pathname, `glob` returns a list consisting of only *pattern* with the total number of pathnames set to 1, and the number of matched pathnames set to 0. If `GLOB_QUOTE` is set, its effect is present in the pattern returned.
GLOB_NOSORT	By default, the pathnames are sorted in ascending `ASCII` order; this flag prevents that sorting (speeding up `glob`) .

The following values may also be included in *flags* , however, they are non-standard extensions to POSIX 1003.2 .

GLOB_ALTDIRFUNC	The following additional fields in the pglob structure have been initialized with alternate functions for glob to use to open, read, and close directories and to get stat information on names found in those directories.

```
void*             (*gl_opendir)(const char* name);
struct dirent*    (*gl_readdir)(void*);
void              (*gl_closedir)(void*);
int               (*gl_lstat)(const char* name, struct stat* st);
int               (*gl_stat)(const char* name, struct stat* st);
```

GLOB_BRACE	Pre-process the pattern string to expand {pat,pat,.... The “{” pattern is left unexpanded for historical reasons.

GLOB_MAGCHAR	Set by the `glob` function if the pattern included globbing characters. See the description of the usage of the *gl_matchc* structure member for more details.
GLOB_NOMAGIC	Is the same as `GLOB_NOCHECK` but it only appends the *pattern* if it does not contain any of the special characters "*", "?" or "[".
GLOB_QUOTE	Use the backslash" \\" character for quoting: every occurrence of a backslash followed by a character in the pattern. This will be replaced by that character, avoiding any special interpretation of the character.
GLOB_TILDE	Expand patterns that start with a tilde ("~") to user name home directories.

If, during the search, a directory is encountered that cannot be opened or read and *errfunc* is non-NULL , `glob` calls *(*errfunc)(path, errno)* . This may be an unintuitive: pattern like */Makefile will try to *stat(2POSIX)* foo/Makefile even if foo is not a directory, resulting in a call to *errfunc* . The error routine can suppress this action by testing for `ENOENT` and `ENOTDIR` ; however, the `GLOB_ERR` flag will still cause an immediate return when this happens.

If *errfunc* returns non-zero, `glob` stops the scan and returns `GLOB_ABEND` after setting *gl_pathc* and *gl_pathv* to reflect any paths already matched. This also happens if an error is encountered and `GLOB_ERR` is set in *flags* , regardless of the return value of *errfunc* , if called. If `GLOB_ERR` is not set and either *errfunc* is `NULL` or *errfunc* returns zero, the error is ignored.

The *globfree* function frees any space associated with *pglob* from any previous call(s) to `glob` .

RETURN VALUES

On successful completion, `glob` returns zero. In addition the fields of *pglob* contain the values described below:

gl_pathc	contains the total number of pathnames matched so far. This includes other matches from previous invocations of `glob` if `GLOB_APPEND` was specified.
gl_matchc	contains the number of matched pathnames in the current invocation of `glob` .
gl_flags	contains a copy of the *flags* parameter with the `GLOB_MAGCHAR` bit set if *pattern* contained any

	of the special characters “*”, “?” or “[”. If not, it is cleared.
gl_pathv	contains a pointer to a `NULL` terminated list of matched pathnames. However, if *gl_pathc* is zero, the contents of *gl_pathv* are undefined.

If `glob` terminates due to an error, it sets errno and returns one of the following non-zero constants, which are defined in the include file `glob.h` :

GLOB_NOSPACE	An attempt to allocate memory failed.
GLOB_ABEND	The scan was stopped because an error was encountered and either `GLOB_ERR` was set or *(*errfunc)()* returned a non-zero value.

The arguments *pglob–>gl_pathc* and *pglob–>gl_pathv* are still set as specified above.

STANDARDS

The `glob` function is POSIX 1003.2 compatible with the exception that the flags `GLOB_ALTDIRFUNC` , `GLOB_BRACE` , `GLOB_MAGCHAR` , `GLOB_NOMAGIC` , `GLOB_QUOTE` and `GLOB_TILDE` and the fields *gl_matchc* and *gl_flags* should not be used by applications which require strict *POSIX* conformance.

HISTORY

The `glob` and *globfree* functions first appeared in 4.4BSD.

BUGS

Patterns longer than `MAXPATHLEN` may cause unchecked errors.

The `glob` argument may fail and set errno for any of the errors specified for the library routines stat(2POSIX), closedir(3POSIX), opendir(3POSIX), readdir(3POSIX), malloc(3STDC), and free(3STDC).

RESTRICTIONS FOR ChorusOS

These library calls do not support multi-threaded applications.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

hash – hash database access method

SYNOPSIS

```
#include <sys/types.h>
#include <db.h>
```

FEATURES

UFS

DESCRIPTION

The *dbopen* routine is the library interface to database files. One of the file formats supported is hash files. The general description of the database access methods is in *dbopen(3POSIX)*, this manual page only describes the hash—specific information.

The hash data structure is an extensible, dynamic hashing scheme.

The access method—specific data structure provided to *dbopen* is defined in the <db.h> include file as follows:

```
typedef struct {
u_int   bsize;
u_int   ffactor;
u_int   nelem;
u_int   cachesize;
u_int32_t  (*hash)(const void*, size_t);
int   lorder;
} HASHINFO;
```

The elements of this structure are as follows:

bsize	*bsize* defines the hash table bucket size, and is, by default, 256 bytes. It may be preferable to increase the page size for disk-resident tables and tables with large data items.
ffactor	*ffactor* indicates a desired density within the hash table. It is an approximation of the number of keys allowed to accumulate in any one bucket, determining when the hash table grows or shrinks. The default value is 8.
nelem	*nelem* is an estimate of the final size of the hash table. If not set or set too low, hash tables will expand gracefully as keys are entered, although a slight performance degradation may be noticed. The default value is 1.
cachesize	A suggested maximum size, in bytes, of the memory cache. This value is `only` advisory, and the access method will allocate more memory rather than fail.

hash	`hash` is a user—defined hash function. As no hash function performs equally well on all possible data, the user may find that the built-in hash function works poorly on a particular data set. User—specified hash functions must take two arguments (a pointer to a byte string and a length) and return a 32-bit quantity to be used as the hash value.
lorder	The byte order for integers in the stored database meta data. The number should represent the order as an integer; for example, big endian order would be the number 4,321. If `lorder` is 0 (no order is specified) the current host order is used. If the file already exists, the value specified is ignored and the value specified when the tree was created is used.

If the file already exists (and the O_TRUNC flag is not specified), the values specified for the parameters bsize, ffactor, lorder and nelem are ignored and the values specified when the tree was created are used.

If a hash function is specified, *hash_open* will attempt to determine whether the hash function specified is the same as the one with which the database was created, and will fail if it is not.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`btree`(3POSIX), `dbopen`(3POSIX), `mpool`(3POSIX), `recno`(3POSIX)

Dynamic Hash Tables, Per-Ake Larson, Communications of the ACM, April 1988.

A New Hash Package for UNIX, Margo Seltzer, USENIX Proceedings, Winter 1991.

BUGS

Only big and little endian byte order is supported.

RESTRICTIONS

These library calls do not support multi-threaded applications.

NAME

getnetgrent, innetgr, setnetgrent, endnetgrent – netgroup database operations

SYNOPSIS

int **getnetgrent**(char ** *host*, char ** *user*, char ** *domain*);

int **innetgr**(const char * *netgroup*, const char * *host*, const char * *user*, const char * *domain*);

void **setnetgrent**(const char * *netgroup*);

void **endnetgrent**(void);

DESCRIPTION

These functions operate on the netgroup database file `/etc/netgroup` which is described in *netgroup(4CC)* . The database defines a set of netgroups, each made up of one or more triples:

(host, user, domain)

which define a combination of host, user and domain. Any of the three fields may be specified as “wildcards” which match any string.

The *getnetgrent* function sets the three pointer arguments to the strings of the next member of the current netgroup. If any of the string pointers are (`char` *)0 that field is considered a wildcard.

The *setnetgrent* and *endnetgrent* functions set the current netgroup and terminate the current netgroup, respectively. If *setnetgrent* is called with a different netgroup from the previous call, an *endnetgrent* is implied. The *setnetgrent* function also sets the offset to the first member of the netgroup.

The *innetgr* function searches for a match of all fields within the specified group. If any of the *host* or *domain* arguments are (`char` *)0 those fields will match any string value in the netgroup member.

RETURN VALUES

The *getnetgrent* function returns 0 for “no more netgroup members” and 1 otherwise. The *innetgr* function returns 1 for a successful match and 0 otherwise. The *setnetgrent* and *endnetgrent* functions have no return value.

FILES

`/etc/netgroup` netgroup database file

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`netgroup`(4CC)

COMPATIBILITY

The netgroup members have three string fields to maintain compatibility with other vendor implementations. However, the applicability of the *domain* string within BSD is unclear.

BUGS

The *getnetgrent* function returns pointers to dynamically allocated data areas that are freed when the *endnetgrent* function is called.

RESTRICTIONS

These library calls do not support multi-threaded applications.

NAME

link_addr, link_ntoa – elementary address specification routines for link level access

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <net/if_dl.h>
void link_addr(const char * addr, struct sockaddr_dl * sdl);

char *link_ntoa(const struct sockaddr_dl * sdl);
```

DESCRIPTION

The *link_addr* routine interprets character strings representing link-level addresses, returning binary information suitable for use in system calls.
The *link_ntoa* routine takes a link-level address and returns an ASCII string representing some of the information present, including the link level address itself, and the interface name or number, if present. This facility is experimental and is still subject to change.

For *link_addr* , the string *addr* may contain an optional network interface identifier of the form `"name unit-number"` , suitable for the first argument to *ifconfig(1M)* , followed in all cases by a colon and an interface address in the form of groups of hexadecimal digits separated by dots. Each group represents a byte of address; address bytes are filled left to right from low order bytes up to high order bytes.

Thus *le0:8.0.9.13.d.30* represents an ethernet address to be transmitted on the first Lance ethernet interface.

RETURN VALUES

The *link_ntoa* function always returns a null terminated string. The *link_addr* function has no return value.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`ios`(7P)

See RESTRICTIONS

BUGS

The returned values for *link_ntoa* reside in a static memory area.

The function *link_addr* should diagnose improperly formed input, and there should be an unambiguous way to recognize this.

If the *sdl_len* field of the link socket address *sdl* is 0, *link_ntoa* will not insert a colon before the interface address bytes. If this translated address is given to *link_addr* without inserting an initial colon, the latter will not interpret it correctly.

RESTRICTIONS

On top of ChorusOS the `ISO` family protocol is not yet supported, *ISO(7P)* is therefore not available. These functions have been ported in order to use network debugging utilities such as `netstat` and `route` with ChorusOS.

NAME

link_addr, link_ntoa – elementary address specification routines for link level access

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <net/if_dl.h>
void link_addr(const char * addr, struct sockaddr_dl * sdl);

char *link_ntoa(const struct sockaddr_dl * sdl);
```

DESCRIPTION

The *link_addr* routine interprets character strings representing link-level addresses, returning binary information suitable for use in system calls.
The *link_ntoa* routine takes a link-level address and returns an ASCII string representing some of the information present, including the link level address itself, and the interface name or number, if present. This facility is experimental and is still subject to change.

For *link_addr* , the string *addr* may contain an optional network interface identifier of the form `"name unit-number"` , suitable for the first argument to *ifconfig(1M)* , followed in all cases by a colon and an interface address in the form of groups of hexadecimal digits separated by dots. Each group represents a byte of address; address bytes are filled left to right from low order bytes up to high order bytes.

Thus *le0:8.0.9.13.d.30* represents an ethernet address to be transmitted on the first Lance ethernet interface.

RETURN VALUES

The *link_ntoa* function always returns a null terminated string. The *link_addr* function has no return value.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`ios`(7P)

See RESTRICTIONS

BUGS

The returned values for *link_ntoa* reside in a static memory area.

The function *link_addr* should diagnose improperly formed input, and there should be an unambiguous way to recognize this.

If the *sdl_len* field of the link socket address *sdl* is 0, *link_ntoa* will not insert a colon before the interface address bytes. If this translated address is given to *link_addr* without inserting an initial colon, the latter will not interpret it correctly.

RESTRICTIONS

On top of ChorusOS the `ISO` family protocol is not yet supported, *ISO(7P)* is therefore not available. These functions have been ported in order to use network debugging utilities such as `netstat` and `route` with ChorusOS.

NAME

mpool – shared memory buffer pool

SYNOPSIS

#include <db.h>
#include <mpool.h>
MPOOL ***mpool_open**(DBT **key*, int *fd*, pgno_t *pagesize*, pgno_t *maxcache*);

void **mpool_filter**(MPOOL **mp*, void (**pgin*)(void *, pgno_t, void *), void (**pgout*)(void *,pgno_t,void *), void **pgcookie*);

void ***mpool_new**(MPOOL **mp*, pgno_t **pgnoaddr*);

void ***mpool_get**(MPOOL **mp*, pgno_t *pgno*, u_int *flags*);

int **mpool_put**(MPOOL **mp*, void **pgaddr*, u_int *flags*);

int **mpool_sync**(MPOOL **mp*);

int **mpool_close**(MPOOL **mp*);

FEATURES

UFS

DESCRIPTION

The *mpool* function is the library interface intended to provide page—oriented buffer management of files. The buffers may be shared between processes.

The *mpool_open* function initializes a memory pool. The *key* argument is the byte string used to negotiate between multiple processes requiring to share buffers. If the file buffers are mapped in shared memory, all processes using the same key will share the buffers. If *key* is NULL, the buffers are mapped into private memory. The *fd* argument is a file descriptor for the underlying file, which must be seekable. If *key* is non-NULL and matches a file already being mapped, the *fd* argument is ignored.

The `pagesize` argument is the size, in bytes, of the pages into which the file is broken up. The *maxcache* argument is the maximum number of pages from the underlying file to cache at any one time. This value is not relative to the number of processes which share a file's buffers, but will be the largest value specified by any of the processes sharing the file.

The *mpool_filter* function is intended to enable transparent input and output processing of the pages. If the *pgin* function is specified, it is called each time a buffer is read into the memory pool from the backing file. If the *pgout* function is specified, it is called each time a buffer is written into the backing file. Both functions are called using the *pgcookie* pointer, the page number and a pointer to the page being read or written to.

The *mpool_new* function takes a MPOOL pointer and an address as arguments. If a new page can be allocated, a pointer to the page is returned and the page number is stored in the *pgnoaddr* address. Otherwise, a NULL value is returned and errno is set to indicqte the error condition..

The *mpool_get* function takes a MPOOL pointer and a page number as arguments. If the page exists, a pointer to the page is returned. Otherwise, a NULL value is returned and errno is set to indicate the error condition. The flags parameter is not currently used.

The *mpool_put* function un-pins the page referenced by *pgaddr*. The *pgaddr* must be an address previously returned by *mpool_get* or *mpool_new*. The flag value is specified by *or* ing any of the following values:

MPOOL_DIRTY	The page has been modified and needs to be written to the backing file.

The *mpool_put* function returns 0 on success and -1 if an error occurs.

The *mpool_sync* function writes all modified pages associated with the MPOOL pointer to the backing file. *mpool_sync* returns 0 on success and -1 if an error occurs.

The *mpool_close* function frees up any allocated memory associated with the memory pool cookie. Modified pages are `not` written to the backing file. The *mpool_close* returns 0 on success and -1 if an error occurs.

ERRORS

The *mpool_open* function may fail and set *errno* to any of the errors specified for the library routine *malloc*(3).

The *mpool_get* function may fail and set *errno* to the following error condition:

[EINVAL]	The requested record does not exist.

The *mpool_new* and *mpool_get* functions may fail and set *errno* to any of the errors specified for the library routines *read(2POSIX), write(2POSIX),* and *malloc(3STDC).*

The *mpool_sync* function may fail and set *errno* to any of the errors specified for the library routine *write(2POSIX).*

The *mpool_close* function may fail and set *errno* to any of the errors specified for the library routine *free(3STDC).*

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`read`(2POSIX), `write`(2POSIX), `free`(3STDC), `malloc`(3STDC), `dbopen`(3POSIX), `btree`(3POSIX), `hash`(3POSIX), `recno`(3POSIX)

RESTRICTIONS

These library calls do not support multi-threaded applications.

NAME

nanosleep – delay the current thread with high resolution

SYNOPSIS

```
#include <time.h>
int nanosleep(const struct timespec *rqtp, struct timespec *rmtp);
```

DESCRIPTION

The *nanosleep* function causes the current thread to be suspended from execution for a period specified in the *rqtp* argument. The suspension time may be longer than requested because the argument value is rounded up to an integer multiple of the resolution of the system realtime clock (CLOCK_REALTIME) or due to scheduling of other activities in the system.

If the current thread is aborted (see *threadAbort*(2K)) while suspended, *nanosleep* will return immediately with *errno* set to EINTR. In this case, if the *rmtp* argument is not NULL, the time remaining before *nanosleep* would have terminated is normally stored at the location referenced by *rmtp*. (NOTE: thread abort is not directly supported by the CHORUS / POSIX Micro Realtime Profile (see *mrtp*(3POSIX))..

RETURN VALUE

Upon normal completion after the requested time has elapsed, *nanosleep* returns zero. Otherwise, if *nanosleep* returned prematurely due to an abort or if any other error is detected, a value of –1 is returned and *errno* is set to indicate the error condition.

ERRORS

[EFAULT]	A pointer argument contains an address outside the current actor's address space.
[EINTR]	*nanosleep* was interrupted by an abort.
[EINVAL]	The *rqtp* argument is NULL, or the time specification referenced by *rqtp* contains an invalid value.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ns_addr, ns_ntoa – Xerox NS address conversion routines

SYNOPSIS

```
#include <sys/types.h>
#include <netns/ns.h>
struct ns_addr ns_addr(char * cp);

char *ns_ntoa(struct ns_addr ns);
```

DESCRIPTION

The *ns_addr* routine interprets character strings representing XNS addresses, returning binary information suitable for use in system calls. The *ns_ntoa* routine takes XNS addresses and returns ASCII strings representing the address in a notation in common use in the Xerox Development Environment:

<network number>.<host number>.<port number>

Trailing zero fields are suppressed, and each number is printed in hexadecimal, in a format suitable for input to *ns_addr* . Any fields lacking super-decimal digits will have a trailing *H* appended.

There is no universal standard for representing XNS addresses. An effort has been made to ensure that *ns_addr* is compatible with most formats in common use. It will first separate an address into 1 to 3 fields using a single delimiter chosen from dot *(".")* , colon *(":")* or pound-sign *("#")* . Each field is then examined for byte separators (colon or dot). If there are byte separators, each subfield separated is taken to be a small hexadecimal number, and the entirety is taken as a network-byte-ordered quantity to be zero extended in the high-network-order bytes. Next, the field is inspected for hyphens, in which case the field is assumed to be a number in decimal notation with hyphens separating the millenia. Next, the field is assumed to be a number: It is interpreted as hexadecimal if there is a leading *0x* (as in C), a trailing *H* (as in Mesa), or there are any super-decimal digits present. It is interpreted as octal if there is a leading *0* and there are no super-octal digits. Otherwise, it is converted as a decimal number.

RETURN VALUES

None.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`hosts`(4CC) , `networks`(4CC)

BUGS

The string returned by *ns_ntoa* resides in a static memory area. The function *ns_addr* should diagnose improperly formed input, and there should be an unambiguous way to recognize this.

NAME

ns_addr, ns_ntoa – Xerox NS address conversion routines

SYNOPSIS

```
#include <sys/types.h>
#include <netns/ns.h>
struct ns_addr ns_addr(char * cp);

char *ns_ntoa(struct ns_addr ns);
```

DESCRIPTION

The *ns_addr* routine interprets character strings representing XNS addresses, returning binary information suitable for use in system calls. The *ns_ntoa* routine takes XNS addresses and returns ASCII strings representing the address in a notation in common use in the Xerox Development Environment:

<network number>.<host number>.<port number>

Trailing zero fields are suppressed, and each number is printed in hexadecimal, in a format suitable for input to *ns_addr* . Any fields lacking super-decimal digits will have a trailing *H* appended.

There is no universal standard for representing XNS addresses. An effort has been made to ensure that *ns_addr* is compatible with most formats in common use. It will first separate an address into 1 to 3 fields using a single delimiter chosen from dot *(".")* , colon *(":")* or pound-sign *("#")* . Each field is then examined for byte separators (colon or dot). If there are byte separators, each subfield separated is taken to be a small hexadecimal number, and the entirety is taken as a network-byte-ordered quantity to be zero extended in the high-network-order bytes. Next, the field is inspected for hyphens, in which case the field is assumed to be a number in decimal notation with hyphens separating the millenia. Next, the field is assumed to be a number: It is interpreted as hexadecimal if there is a leading *0x* (as in C), a trailing *H* (as in Mesa), or there are any super-decimal digits present. It is interpreted as octal if there is a leading *0* and there are no super-octal digits. Otherwise, it is converted as a decimal number.

RETURN VALUES

None.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`hosts`(4CC) , `networks`(4CC)

BUGS

The string returned by *ns_ntoa* resides in a static memory area. The function *ns_addr* should diagnose improperly formed input, and there should be an unambiguous way to recognize this.

NAME

directory, opendir, readdir, telldir, seekdir, rewinddir, closedir – directory operations

SYNOPSIS

#include <sys/types.h>
#include <dirent.h>
DIR * **opendir**(const char * *filename*);

struct dirent * **readdir**(DIR * *dirp*);

long **telldir**(const DIR * *dirp*);

void **seekdir**(DIR * *dirp*, long *loc*);

void **rewinddir**(DIR * *dirp*);

int **closedir**(DIR * *dirp*);

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

The *opendir* function opens the directory named by *filename* , associates a directory stream with it and returns a pointer to be used to identify the directory stream in subsequent operations. The `NULL` pointer is returned if *filename* cannot be accessed, or if it cannot *malloc(3STDC)* enough memory to hold all of it.

The *readdir* function returns a pointer to the next directory entry. It returns `NULL` upon reaching the end of the directory or detecting an invalid *seekdir* operation.

The *telldir* function returns the current location associated with the named directory stream.

The *seekdir* function sets the position of the next *readdir* operation on the directory stream. The new position reverts to the one associated with the directory stream when the *telldir* operation was performed. Values returned by *telldir* are valid only for the lifetime of the `DIR` pointer, *dirp* , from which they are derived. If the directory is closed and then reopened, the *telldir* value may be invalidated due to undetected directory compaction. It is safe to use a previous *telldir* value immediately after a call to *opendir* and before any calls to *readdir* .

The *rewinddir* function resets the position of the named directory stream to the beginning of the directory.

The *closedir* function closes the named directory stream and frees the structure associated with the *dirp* pointer, returning 0 on success. On failure, –1 is returned and the global variable *errno* is set to indicate the error.

Sample code which searches a directory for the “name” entry is:

```
len = strlen(name);
dirp = opendir(".");
if (dirp) {
   while ((dp = readdir(dirp)) != NULL)
 if (dp->d_namlen == len && !strcmp(dp->d_name, name)) {
  (void) closedir(dirp);
  return FOUND;
         }
   (void) closedir(dirp);
   }
return NOT_FOUND;
```

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`open`(2POSIX) , `close`(2POSIX) , `read`(2POSIX) , `lseek`(2POSIX)

HISTORY

The *opendir, readdir, telldir, seekdir, rewinddir* and *closedir* functions appeared in 4.2 BSD.

RESTRICTIONS

These library calls do not support multi-threaded applications.

NAME

pthread_attr_init, pthread_attr_destroy, pthread_attr_setstacksize, pthread_attr_getstacksize, pthread_attr_setstackaddr, pthread_attr_getstackaddr, pthread_attr_setdetachstate, pthread_attr_getdetachstate – Initialize a thread attribute object; Destroy a thread attribute object; Set the stacksize attribute; Get the stacksize attribute; Set the stackaddr attribute; Get the stackaddr attribute; Set the detachstate attribute; Get the detachstate attribute

SYNOPSIS

#include <pthread.h>

int **pthread_attr_init**(pthread_attr_t * *attr*);

int **pthread_attr_destroy**(pthread_attr_t * *attr*);

int **pthread_attr_setstacksize**(pthread_attr_t * *attr*, size_t *stacksize*);

int **pthread_attr_getstacksize**(const pthread_attr_t * *attr*, size_t * *stacksize*);

int **pthread_attr_setstackaddr**(pthread_attr_t * *attr*, void * *stackaddr*);

int **pthread_attr_getstackaddr**(const pthread_attr_t * *attr*, void ** *stackaddr*);

int **pthread_attr_setdetachstate**(pthread_attr_t * *attr*, int *detachstate*);

int **pthread_attr_getdetachstate**(const pthread_attr_t * *attr*, int * *detachstate*);

DESCRIPTION

The *pthread_attr_init* function initializes the thread attribute object referenced by *attr* with the default values for all the individual thread attributes. The resulting attribute object may be modified by setting individual attribute values. When subsequently used by *pthread_create* , it defines the attributes of the newly created thread. A single attribute object can be used in multiple simultaneous calls to *pthread_create* . Modification of an attribute object has no effect on threads already created using that object.

The *pthread_attr_destroy* function is used to delete a thread attribute object.

The complete list of thread creation attributes follows, with defaults indicated.

stacksize	PTHREAD_STACK_MIN
stackaddr	*stack dynamically allocated*
detachstate	PTHREAD_CREATE_JOINABLE
contentionscope	PTHREAD_SCOPE_SYSTEM
inheritsched	PTHREAD_INHERIT_SCHED
schedpolicy	*default schedpolicy (see mrtp(*POSIX*))*
schedparam	*default schedparam (see mrtp(*POSIX*))*

The latter four attributes, which pertain to scheduling, are described in *pthread_attr_setscope* (3POSIX).

The *stacksize* attribute defines the minimum stack size (in bytes). The *pthread_attr_setstacksize* and *pthread_attr_getstacksize* functions respectively set and get the value of the *stacksize* attribute in the thread attribute object referenced by *attr* .

The *stackaddr* attribute specifies the location in memory to be used for the newly created thread's stack. If no *stackaddr* value is provided explicitly, the stack for the new thread will be allocated dynamically by the system. The *pthread_attr_setstackaddr* and *pthread_attr_getstackaddr* functions respectively set and get the value of the *stackaddr* attribute in the *attr* object.

The *detachstate* attribute controls the behavior of a new thread when using the *pthread_join* function. If the *detachstate* attribute is set to PTHREAD_CREATE_JOINABLE (the default), the identifier of the thread may be used as the target of a *pthread_join* . If *detachstate* is set to PTHREAD_CREATE_DETACHED, all resources associated with the thread are freed immediately on exit, and the thread identifier may not be used in a *pthread_join* (see *pthread_exit* (3POSIX), *pthread_join* (3POSIX)). The *pthread_attr_setdetachstate* and *pthread_attr_getdetachstate* functions respectively set and get the *detachstate* attribute in the *attr* object.

RESTRICTIONS

The *pthread_attr_setstackaddr* and *pthread_attr_getstackaddr* functions are available only in user mode. In supervisor actors the *stackaddr* attribute is not defined, and the stack for a new thread is always provided by the system.

RETURN VALUE

Upon successful completion, all calls listed above return zero. Otherwise, an error code is returned. (NOTE: These calls do not set *errno* .)

ERRORS

[EINVAL]	The *stacksize* argument to *pthread_attr_setstacksize* is less than PTHREAD_STACK_MIN. The *detachstate* argument to *pthread_attr_setdetachstate* contains a value other than PTHREAD_CREATE_JOINABLE or PTHREAD_CREATE_DETACHED.
[ENOSYS]	*pthread_attr_setstackaddr* and *pthread_attr_getstackaddr* are not supported for supervisor mode actors.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`pthread_create`(3POSIX)

NAME

pthread_attr_init, pthread_attr_destroy, pthread_attr_setstacksize, pthread_attr_getstacksize, pthread_attr_setstackaddr, pthread_attr_getstackaddr, pthread_attr_setdetachstate, pthread_attr_getdetachstate – Initialize a thread attribute object; Destroy a thread attribute object; Set the stacksize attribute; Get the stacksize attribute; Set the stackaddr attribute; Get the stackaddr attribute; Set the detachstate attribute; Get the detachstate attribute

SYNOPSIS

#include <pthread.h>
int **pthread_attr_init**(pthread_attr_t * *attr*);

int **pthread_attr_destroy**(pthread_attr_t * *attr*);

int **pthread_attr_setstacksize**(pthread_attr_t * *attr*, size_t *stacksize*);

int **pthread_attr_getstacksize**(const pthread_attr_t * *attr*, size_t * *stacksize*);

int **pthread_attr_setstackaddr**(pthread_attr_t * *attr*, void * *stackaddr*);

int **pthread_attr_getstackaddr**(const pthread_attr_t * *attr*, void ** *stackaddr*);

int **pthread_attr_setdetachstate**(pthread_attr_t * *attr*, int *detachstate*);

int **pthread_attr_getdetachstate**(const pthread_attr_t * *attr*, int * *detachstate*);

DESCRIPTION

The *pthread_attr_init* function initializes the thread attribute object referenced by *attr* with the default values for all the individual thread attributes. The resulting attribute object may be modified by setting individual attribute values. When subsequently used by *pthread_create* , it defines the attributes of the newly created thread. A single attribute object can be used in multiple simultaneous calls to *pthread_create* . Modification of an attribute object has no effect on threads already created using that object.

The *pthread_attr_destroy* function is used to delete a thread attribute object.

The complete list of thread creation attributes follows, with defaults indicated.

stacksize	PTHREAD_STACK_MIN
stackaddr	*stack dynamically allocated*
detachstate	PTHREAD_CREATE_JOINABLE
contentionscope	PTHREAD_SCOPE_SYSTEM
inheritsched	PTHREAD_INHERIT_SCHED
schedpolicy	*default schedpolicy (see mrtp(*POSIX*))*
schedparam	*default schedparam (see mrtp(*POSIX*))*

The latter four attributes, which pertain to scheduling, are described in *pthread_attr_setscope* (3POSIX).

The *stacksize* attribute defines the minimum stack size (in bytes). The *pthread_attr_setstacksize* and *pthread_attr_getstacksize* functions respectively set and get the value of the *stacksize* attribute in the thread attribute object referenced by *attr* .

The *stackaddr* attribute specifies the location in memory to be used for the newly created thread's stack. If no *stackaddr* value is provided explicitly, the stack for the new thread will be allocated dynamically by the system. The *pthread_attr_setstackaddr* and *pthread_attr_getstackaddr* functions respectively set and get the value of the *stackaddr* attribute in the *attr* object.

The *detachstate* attribute controls the behavior of a new thread when using the *pthread_join* function. If the *detachstate* attribute is set to PTHREAD_CREATE_JOINABLE (the default), the identifier of the thread may be used as the target of a *pthread_join* . If *detachstate* is set to PTHREAD_CREATE_DETACHED, all resources associated with the thread are freed immediately on exit, and the thread identifier may not be used in a *pthread_join* (see *pthread_exit* (3POSIX), *pthread_join* (3POSIX)). The *pthread_attr_setdetachstate* and *pthread_attr_getdetachstate* functions respectively set and get the *detachstate* attribute in the *attr* object.

RESTRICTIONS

The *pthread_attr_setstackaddr* and *pthread_attr_getstackaddr* functions are available only in user mode. In supervisor actors the *stackaddr* attribute is not defined, and the stack for a new thread is always provided by the system.

RETURN VALUE

Upon successful completion, all calls listed above return zero. Otherwise, an error code is returned. (NOTE: These calls do not set *errno* .)

ERRORS

[EINVAL]	The *stacksize* argument to *pthread_attr_setstacksize* is less than PTHREAD_STACK_MIN. The *detachstate* argument to *pthread_attr_setdetachstate* contains a value other than PTHREAD_CREATE_JOINABLE or PTHREAD_CREATE_DETACHED.
[ENOSYS]	*pthread_attr_setstackaddr* and *pthread_attr_getstackaddr* are not supported for supervisor mode actors.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`pthread_create`(3POSIX)

NAME

pthread_attr_setscope, pthread_attr_getscope, pthread_attr_setinheritsched, pthread_attr_getinheritsched, pthread_attr_setschedpolicy, pthread_attr_getschedpolicy, pthread_attr_setschedparam, pthread_attr_getschedparam – Set the contention scope attribute; Get the contention scope attribute; Set the scheduling inheritance attribute; Get the scheduling inheritance attribute; Set the scheduling policy attribute; Get the scheduling policy attribute; Set the scheduling parameter attribute; Get the scheduling parameter attribute

SYNOPSIS

#include <pthread.h>
int **pthread_attr_setscope**(pthread_attr_t * *attr*, int *contentionscope*);

int **pthread_attr_getscope**(const pthread_attr_t * *attr*, int * *contentionscope*);

int **pthread_attr_setinheritsched**(pthread_attr_t * *attr*, int *inheritsched*);

int **pthread_attr_getinheritsched**(const pthread_attr_t * *attr*, int * *inheritsched*);

int **pthread_attr_setschedpolicy**(pthread_attr_t * *attr*, int *policy*);

int **pthread_attr_getschedpolicy**(const pthread_attr_t * *attr*, int * *policy*);

int **pthread_attr_setschedparam**(pthread_attr_t * *attr*, const struct sched_param * *param*);

int **pthread_attr_getschedparam**(const pthread_attr_t * *attr*, struct sched_param * *param*);

DESCRIPTION

Thread creation attributes and their defaults are summarized in *pthread_attr_init* (3POSIX).

The *contentionscope* attribute specifies the scope of thread scheduling decisions relative to a pthread. In the CHORUS / POSIX Micro Realtime Profile, the only *contentionscope* value supported is PTHREAD_SCOPE_SYSTEM, which means that the thread competes directly with all other threads on the site for processor resources. The *pthread_attr_setscope* and *pthread_attr_getscope* functions respectively set and get the *contentionscope* attribute in the thread creation attribute object designated by *attr* .

The *inheritsched* attribute controls the initialization of scheduling parameters in the newly created thread:

PTHREAD_INHERIT_SCHED Specifies that the scheduling policy and associated attributes (for exanmple, priority) are to be inherited from the creating thread, and the corresponding values in the attribute object are to be ignored. This is the default.

PTHREAD_EXPLICIT_SCHED Specifies that the scheduling policy and associated attributes in the new thread are to be

set to the corresponding values from the attribute object.

The *pthread_attr_setinheritsched* and *pthread_attr_getinheritsched* functions respectively set and get the *inheritsched* attribute in the *attr* object.

The *schedpolicy* attribute specifies the thread scheduling policy. Supported values include SCHED_RR, SCHED_FIFO, and SCHED_OTHER which is equivalent to SCHED_RR (see *mrtp* (3POSIX) for information on scheduling). The *pthread_attr_setschedpolicy* and *pthread_attr_getschedpolicy* functions respectively set and get the *schedpolicy* attribute in the *attr* object.

Each POSIX scheduling policy defines a parameter structure *sched_param* containing parameters specific to the policy. For both SCHED_RR and SCHED_FIFO the sole parameter is thread priority, and the structure is defined as follows:

```
struct sched_param {
    int       sched_priority;
};
```

The *pthread_attr_setschedparam* and *pthread_attr_getschedparam* functions respectively set and get the scheduling parameter structure in the *attr* object.

RETURN VALUE

Upon successful completion, all calls listed above return zero. Otherwise an error code is returned. (NOTE: These calls do not set *errno* .)

ERRORS

[EINVAL]	*pthread_attr_setinheritsched* was invoked with an invalid *inherit* argument. *pthread_attr_setschedparam* was invoked with an invalid value for the *sched_priority* member of the *param* argument.
[ENOTSUP]	*pthread_attr_setscope* was invoked with a *contentionscope* argument other than PTHREAD_SCOPE_SYSTEM. *pthread_attr_setschedpolicy* was invoked with an unsupported value for the *policy* argument.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`pthread_attr_init`(3POSIX) , `pthread_create`(3POSIX)

NAME

pthread_attr_setscope, pthread_attr_getscope, pthread_attr_setinheritsched, pthread_attr_getinheritsched, pthread_attr_setschedpolicy, pthread_attr_getschedpolicy, pthread_attr_setschedparam, pthread_attr_getschedparam – Set the contention scope attribute; Get the contention scope attribute; Set the scheduling inheritance attribute; Get the scheduling inheritance attribute; Set the scheduling policy attribute; Get the scheduling policy attribute; Set the scheduling parameter attribute; Get the scheduling parameter attribute

SYNOPSIS

#include <pthread.h>
int **pthread_attr_setscope**(pthread_attr_t * *attr*, int *contentionscope*);

int **pthread_attr_getscope**(const pthread_attr_t * *attr*, int * *contentionscope*);

int **pthread_attr_setinheritsched**(pthread_attr_t * *attr*, int *inheritsched*);

int **pthread_attr_getinheritsched**(const pthread_attr_t * *attr*, int * *inheritsched*);

int **pthread_attr_setschedpolicy**(pthread_attr_t * *attr*, int *policy*);

int **pthread_attr_getschedpolicy**(const pthread_attr_t * *attr*, int * *policy*);

int **pthread_attr_setschedparam**(pthread_attr_t * *attr*, const struct sched_param * *param*);

int **pthread_attr_getschedparam**(const pthread_attr_t * *attr*, struct sched_param * *param*);

DESCRIPTION

Thread creation attributes and their defaults are summarized in *pthread_attr_init* (3POSIX).

The *contentionscope* attribute specifies the scope of thread scheduling decisions relative to a pthread. In the CHORUS / POSIX Micro Realtime Profile, the only *contentionscope* value supported is PTHREAD_SCOPE_SYSTEM, which means that the thread competes directly with all other threads on the site for processor resources. The *pthread_attr_setscope* and *pthread_attr_getscope* functions respectively set and get the *contentionscope* attribute in the thread creation attribute object designated by *attr* .

The *inheritsched* attribute controls the initialization of scheduling parameters in the newly created thread:

PTHREAD_INHERIT_SCHED	Specifies that the scheduling policy and associated attributes (for exanmple, priority) are to be inherited from the creating thread, and the corresponding values in the attribute object are to be ignored. This is the default.
PTHREAD_EXPLICIT_SCHED	Specifies that the scheduling policy and associated attributes in the new thread are to be

set to the corresponding values from the attribute object.

The *pthread_attr_setinheritsched* and *pthread_attr_getinheritsched* functions respectively set and get the *inheritsched* attribute in the *attr* object.

The *schedpolicy* attribute specifies the thread scheduling policy. Supported values include SCHED_RR, SCHED_FIFO, and SCHED_OTHER which is equivalent to SCHED_RR (see *mrtp* (3POSIX) for information on scheduling). The *pthread_attr_setschedpolicy* and *pthread_attr_getschedpolicy* functions respectively set and get the *schedpolicy* attribute in the *attr* object.

Each POSIX scheduling policy defines a parameter structure *sched_param* containing parameters specific to the policy. For both SCHED_RR and SCHED_FIFO the sole parameter is thread priority, and the structure is defined as follows:

```
struct sched_param {
    int      sched_priority;
};
```

The *pthread_attr_setschedparam* and *pthread_attr_getschedparam* functions respectively set and get the scheduling parameter structure in the *attr* object.

RETURN VALUE

Upon successful completion, all calls listed above return zero. Otherwise an error code is returned. (NOTE: These calls do not set *errno* .)

ERRORS

[EINVAL]	*pthread_attr_setinheritsched* was invoked with an invalid *inherit* argument. *pthread_attr_setschedparam* was invoked with an invalid value for the *sched_priority* member of the *param* argument.
[ENOTSUP]	*pthread_attr_setscope* was invoked with a *contentionscope* argument other than PTHREAD_SCOPE_SYSTEM. *pthread_attr_setschedpolicy* was invoked with an unsupported value for the *policy* argument.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`pthread_attr_init`(3POSIX) , `pthread_create`(3POSIX)

NAME

pthread_attr_setscope, pthread_attr_getscope, pthread_attr_setinheritsched, pthread_attr_getinheritsched, pthread_attr_setschedpolicy, pthread_attr_getschedpolicy, pthread_attr_setschedparam, pthread_attr_getschedparam – Set the contention scope attribute; Get the contention scope attribute; Set the scheduling inheritance attribute; Get the scheduling inheritance attribute; Set the scheduling policy attribute; Get the scheduling policy attribute; Set the scheduling parameter attribute; Get the scheduling parameter attribute

SYNOPSIS

#include <pthread.h>
int **pthread_attr_setscope**(pthread_attr_t * *attr*, int *contentionscope*);

int **pthread_attr_getscope**(const pthread_attr_t * *attr*, int * *contentionscope*);

int **pthread_attr_setinheritsched**(pthread_attr_t * *attr*, int *inheritsched*);

int **pthread_attr_getinheritsched**(const pthread_attr_t * *attr*, int * *inheritsched*);

int **pthread_attr_setschedpolicy**(pthread_attr_t * *attr*, int *policy*);

int **pthread_attr_getschedpolicy**(const pthread_attr_t * *attr*, int * *policy*);

int **pthread_attr_setschedparam**(pthread_attr_t * *attr*, const struct sched_param * *param*);

int **pthread_attr_getschedparam**(const pthread_attr_t * *attr*, struct sched_param * *param*);

DESCRIPTION

Thread creation attributes and their defaults are summarized in *pthread_attr_init* (3POSIX).

The *contentionscope* attribute specifies the scope of thread scheduling decisions relative to a pthread. In the CHORUS/POSIX Micro Realtime Profile, the only *contentionscope* value supported is PTHREAD_SCOPE_SYSTEM, which means that the thread competes directly with all other threads on the site for processor resources. The *pthread_attr_setscope* and *pthread_attr_getscope* functions respectively set and get the *contentionscope* attribute in the thread creation attribute object designated by *attr* .

The *inheritsched* attribute controls the initialization of scheduling parameters in the newly created thread:

PTHREAD_INHERIT_SCHED	Specifies that the scheduling policy and associated attributes (for exanmple, priority) are to be inherited from the creating thread, and the corresponding values in the attribute object are to be ignored. This is the default.
PTHREAD_EXPLICIT_SCHED	Specifies that the scheduling policy and associated attributes in the new thread are to be

set to the corresponding values from the attribute object.

The *pthread_attr_setinheritsched* and *pthread_attr_getinheritsched* functions respectively set and get the *inheritsched* attribute in the *attr* object.

The *schedpolicy* attribute specifies the thread scheduling policy. Supported values include SCHED_RR, SCHED_FIFO, and SCHED_OTHER which is equivalent to SCHED_RR (see *mrtp* (3POSIX) for information on scheduling). The *pthread_attr_setschedpolicy* and *pthread_attr_getschedpolicy* functions respectively set and get the *schedpolicy* attribute in the *attr* object.

Each POSIX scheduling policy defines a parameter structure *sched_param* containing parameters specific to the policy. For both SCHED_RR and SCHED_FIFO the sole parameter is thread priority, and the structure is defined as follows:

```
struct sched_param {
    int      sched_priority;
};
```

The *pthread_attr_setschedparam* and *pthread_attr_getschedparam* functions respectively set and get the scheduling parameter structure in the *attr* object.

RETURN VALUE

Upon successful completion, all calls listed above return zero. Otherwise an error code is returned. (NOTE: These calls do not set *errno* .)

ERRORS

[EINVAL]	*pthread_attr_setinheritsched* was invoked with an invalid *inherit* argument. *pthread_attr_setschedparam* was invoked with an invalid value for the *sched_priority* member of the *param* argument.
[ENOTSUP]	*pthread_attr_setscope* was invoked with a *contentionscope* argument other than PTHREAD_SCOPE_SYSTEM. *pthread_attr_setschedpolicy* was invoked with an unsupported value for the *policy* argument.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`pthread_attr_init`(3POSIX) , `pthread_create`(3POSIX)

NAME

pthread_attr_setscope, pthread_attr_getscope, pthread_attr_setinheritsched, pthread_attr_getinheritsched, pthread_attr_setschedpolicy, pthread_attr_getschedpolicy, pthread_attr_setschedparam, pthread_attr_getschedparam – Set the contention scope attribute; Get the contention scope attribute; Set the scheduling inheritance attribute; Get the scheduling inheritance attribute; Set the scheduling policy attribute; Get the scheduling policy attribute; Set the scheduling parameter attribute; Get the scheduling parameter attribute

SYNOPSIS

#include <pthread.h>

int **pthread_attr_setscope**(pthread_attr_t * *attr*, int *contentionscope*);

int **pthread_attr_getscope**(const pthread_attr_t * *attr*, int * *contentionscope*);

int **pthread_attr_setinheritsched**(pthread_attr_t * *attr*, int *inheritsched*);

int **pthread_attr_getinheritsched**(const pthread_attr_t * *attr*, int * *inheritsched*);

int **pthread_attr_setschedpolicy**(pthread_attr_t * *attr*, int *policy*);

int **pthread_attr_getschedpolicy**(const pthread_attr_t * *attr*, int * *policy*);

int **pthread_attr_setschedparam**(pthread_attr_t * *attr*, const struct sched_param * *param*);

int **pthread_attr_getschedparam**(const pthread_attr_t * *attr*, struct sched_param * *param*);

DESCRIPTION

Thread creation attributes and their defaults are summarized in *pthread_attr_init* (3POSIX).

The *contentionscope* attribute specifies the scope of thread scheduling decisions relative to a pthread. In the CHORUS / POSIX Micro Realtime Profile, the only *contentionscope* value supported is PTHREAD_SCOPE_SYSTEM, which means that the thread competes directly with all other threads on the site for processor resources. The *pthread_attr_setscope* and *pthread_attr_getscope* functions respectively set and get the *contentionscope* attribute in the thread creation attribute object designated by *attr* .

The *inheritsched* attribute controls the initialization of scheduling parameters in the newly created thread:

PTHREAD_INHERIT_SCHED Specifies that the scheduling policy and associated attributes (for exanmple, priority) are to be inherited from the creating thread, and the corresponding values in the attribute object are to be ignored. This is the default.

PTHREAD_EXPLICIT_SCHED Specifies that the scheduling policy and associated attributes in the new thread are to be

set to the corresponding values from the attribute object.

The *pthread_attr_setinheritsched* and *pthread_attr_getinheritsched* functions respectively set and get the *inheritsched* attribute in the *attr* object.

The *schedpolicy* attribute specifies the thread scheduling policy. Supported values include SCHED_RR, SCHED_FIFO, and SCHED_OTHER which is equivalent to SCHED_RR (see *mrtp* (3POSIX) for information on scheduling). The *pthread_attr_setschedpolicy* and *pthread_attr_getschedpolicy* functions respectively set and get the *schedpolicy* attribute in the *attr* object.

Each POSIX scheduling policy defines a parameter structure *sched_param* containing parameters specific to the policy. For both SCHED_RR and SCHED_FIFO the sole parameter is thread priority, and the structure is defined as follows:

```
struct sched_param {
    int      sched_priority;
};
```

The *pthread_attr_setschedparam* and *pthread_attr_getschedparam* functions respectively set and get the scheduling parameter structure in the *attr* object.

RETURN VALUE

Upon successful completion, all calls listed above return zero. Otherwise an error code is returned. (NOTE: These calls do not set *errno* .)

ERRORS

[EINVAL]	*pthread_attr_setinheritsched* was invoked with an invalid *inherit* argument. *pthread_attr_setschedparam* was invoked with an invalid value for the *sched_priority* member of the *param* argument.
[ENOTSUP]	*pthread_attr_setscope* was invoked with a *contentionscope* argument other than PTHREAD_SCOPE_SYSTEM. *pthread_attr_setschedpolicy* was invoked with an unsupported value for the *policy* argument.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`pthread_attr_init`(3POSIX) , `pthread_create`(3POSIX)

NAME

pthread_attr_init, pthread_attr_destroy, pthread_attr_setstacksize, pthread_attr_getstacksize, pthread_attr_setstackaddr, pthread_attr_getstackaddr, pthread_attr_setdetachstate, pthread_attr_getdetachstate – Initialize a thread attribute object; Destroy a thread attribute object; Set the stacksize attribute; Get the stacksize attribute; Set the stackaddr attribute; Get the stackaddr attribute; Set the detachstate attribute; Get the detachstate attribute

SYNOPSIS

#include <pthread.h>
int **pthread_attr_init**(pthread_attr_t * *attr*);

int **pthread_attr_destroy**(pthread_attr_t * *attr*);

int **pthread_attr_setstacksize**(pthread_attr_t * *attr*, size_t *stacksize*);

int **pthread_attr_getstacksize**(const pthread_attr_t * *attr*, size_t * *stacksize*);

int **pthread_attr_setstackaddr**(pthread_attr_t * *attr*, void * *stackaddr*);

int **pthread_attr_getstackaddr**(const pthread_attr_t * *attr*, void ** *stackaddr*);

int **pthread_attr_setdetachstate**(pthread_attr_t * *attr*, int *detachstate*);

int **pthread_attr_getdetachstate**(const pthread_attr_t * *attr*, int * *detachstate*);

DESCRIPTION

The *pthread_attr_init* function initializes the thread attribute object referenced by *attr* with the default values for all the individual thread attributes. The resulting attribute object may be modified by setting individual attribute values. When subsequently used by *pthread_create* , it defines the attributes of the newly created thread. A single attribute object can be used in multiple simultaneous calls to *pthread_create* . Modification of an attribute object has no effect on threads already created using that object.

The *pthread_attr_destroy* function is used to delete a thread attribute object.

The complete list of thread creation attributes follows, with defaults indicated.

`stacksize`	`PTHREAD_STACK_MIN`
`stackaddr`	*stack dynamically allocated*
`detachstate`	`PTHREAD_CREATE_JOINABLE`
`contentionscope`	`PTHREAD_SCOPE_SYSTEM`
`inheritsched`	`PTHREAD_INHERIT_SCHED`
`schedpolicy`	*default schedpolicy (see mrtp(*POSIX*))*
`schedparam`	*default schedparam (see mrtp(*POSIX*))*

The latter four attributes, which pertain to scheduling, are described in *pthread_attr_setscope* (3POSIX).

The *stacksize* attribute defines the minimum stack size (in bytes). The *pthread_attr_setstacksize* and *pthread_attr_getstacksize* functions respectively set and get the value of the *stacksize* attribute in the thread attribute object referenced by *attr* .

The *stackaddr* attribute specifies the location in memory to be used for the newly created thread's stack. If no *stackaddr* value is provided explicitly, the stack for the new thread will be allocated dynamically by the system. The *pthread_attr_setstackaddr* and *pthread_attr_getstackaddr* functions respectively set and get the value of the *stackaddr* attribute in the *attr* object.

The *detachstate* attribute controls the behavior of a new thread when using the *pthread_join* function. If the *detachstate* attribute is set to PTHREAD_CREATE_JOINABLE (the default), the identifier of the thread may be used as the target of a *pthread_join* . If *detachstate* is set to PTHREAD_CREATE_DETACHED, all resources associated with the thread are freed immediately on exit, and the thread identifier may not be used in a *pthread_join* (see *pthread_exit* (3POSIX), *pthread_join* (3POSIX)). The *pthread_attr_setdetachstate* and *pthread_attr_getdetachstate* functions respectively set and get the *detachstate* attribute in the *attr* object.

RESTRICTIONS

The *pthread_attr_setstackaddr* and *pthread_attr_getstackaddr* functions are available only in user mode. In supervisor actors the *stackaddr* attribute is not defined, and the stack for a new thread is always provided by the system.

RETURN VALUE

Upon successful completion, all calls listed above return zero. Otherwise, an error code is returned. (NOTE: These calls do not set *errno* .)

ERRORS

[EINVAL]	The *stacksize* argument to *pthread_attr_setstacksize* is less than PTHREAD_STACK_MIN. The *detachstate* argument to *pthread_attr_setdetachstate* contains a value other than PTHREAD_CREATE_JOINABLE or PTHREAD_CREATE_DETACHED.
[ENOSYS]	*pthread_attr_setstackaddr* and *pthread_attr_getstackaddr* are not supported for supervisor mode actors.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`pthread_create`(3POSIX)

NAME

pthread_attr_init, pthread_attr_destroy, pthread_attr_setstacksize, pthread_attr_getstacksize, pthread_attr_setstackaddr, pthread_attr_getstackaddr, pthread_attr_setdetachstate, pthread_attr_getdetachstate – Initialize a thread attribute object; Destroy a thread attribute object; Set the stacksize attribute; Get the stacksize attribute; Set the stackaddr attribute; Get the stackaddr attribute; Set the detachstate attribute; Get the detachstate attribute

SYNOPSIS

#include <pthread.h>
int **pthread_attr_init**(pthread_attr_t * *attr*);

int **pthread_attr_destroy**(pthread_attr_t * *attr*);

int **pthread_attr_setstacksize**(pthread_attr_t * *attr*, size_t *stacksize*);

int **pthread_attr_getstacksize**(const pthread_attr_t * *attr*, size_t * *stacksize*);

int **pthread_attr_setstackaddr**(pthread_attr_t * *attr*, void * *stackaddr*);

int **pthread_attr_getstackaddr**(const pthread_attr_t * *attr*, void ** *stackaddr*);

int **pthread_attr_setdetachstate**(pthread_attr_t * *attr*, int *detachstate*);

int **pthread_attr_getdetachstate**(const pthread_attr_t * *attr*, int * *detachstate*);

DESCRIPTION

The *pthread_attr_init* function initializes the thread attribute object referenced by *attr* with the default values for all the individual thread attributes. The resulting attribute object may be modified by setting individual attribute values. When subsequently used by *pthread_create* , it defines the attributes of the newly created thread. A single attribute object can be used in multiple simultaneous calls to *pthread_create* . Modification of an attribute object has no effect on threads already created using that object.

The *pthread_attr_destroy* function is used to delete a thread attribute object.

The complete list of thread creation attributes follows, with defaults indicated.

`stacksize`	`PTHREAD_STACK_MIN`
`stackaddr`	*stack dynamically allocated*
`detachstate`	`PTHREAD_CREATE_JOINABLE`
`contentionscope`	`PTHREAD_SCOPE_SYSTEM`
`inheritsched`	`PTHREAD_INHERIT_SCHED`
`schedpolicy`	*default schedpolicy (see mrtp(*`POSIX`*))*
`schedparam`	*default schedparam (see mrtp(*`POSIX`*))*

The latter four attributes, which pertain to scheduling, are described in *pthread_attr_setscope* (3POSIX).

The *stacksize* attribute defines the minimum stack size (in bytes). The *pthread_attr_setstacksize* and *pthread_attr_getstacksize* functions respectively set and get the value of the *stacksize* attribute in the thread attribute object referenced by *attr* .

The *stackaddr* attribute specifies the location in memory to be used for the newly created thread's stack. If no *stackaddr* value is provided explicitly, the stack for the new thread will be allocated dynamically by the system. The *pthread_attr_setstackaddr* and *pthread_attr_getstackaddr* functions respectively set and get the value of the *stackaddr* attribute in the *attr* object.

The *detachstate* attribute controls the behavior of a new thread when using the *pthread_join* function. If the *detachstate* attribute is set to PTHREAD_CREATE_JOINABLE (the default), the identifier of the thread may be used as the target of a *pthread_join* . If *detachstate* is set to PTHREAD_CREATE_DETACHED, all resources associated with the thread are freed immediately on exit, and the thread identifier may not be used in a *pthread_join* (see *pthread_exit* (3POSIX), *pthread_join* (3POSIX)). The *pthread_attr_setdetachstate* and *pthread_attr_getdetachstate* functions respectively set and get the *detachstate* attribute in the *attr* object.

RESTRICTIONS

The *pthread_attr_setstackaddr* and *pthread_attr_getstackaddr* functions are available only in user mode. In supervisor actors the *stackaddr* attribute is not defined, and the stack for a new thread is always provided by the system.

RETURN VALUE

Upon successful completion, all calls listed above return zero. Otherwise, an error code is returned. (NOTE: These calls do not set *errno* .)

ERRORS

[EINVAL]	The *stacksize* argument to *pthread_attr_setstacksize* is less than PTHREAD_STACK_MIN. The *detachstate* argument to *pthread_attr_setdetachstate* contains a value other than PTHREAD_CREATE_JOINABLE or PTHREAD_CREATE_DETACHED.
[ENOSYS]	*pthread_attr_setstackaddr* and *pthread_attr_getstackaddr* are not supported for supervisor mode actors.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`pthread_create`(3POSIX)

NAME

pthread_attr_init, pthread_attr_destroy, pthread_attr_setstacksize, pthread_attr_getstacksize, pthread_attr_setstackaddr, pthread_attr_getstackaddr, pthread_attr_setdetachstate, pthread_attr_getdetachstate – Initialize a thread attribute object; Destroy a thread attribute object; Set the stacksize attribute; Get the stacksize attribute; Set the stackaddr attribute; Get the stackaddr attribute; Set the detachstate attribute; Get the detachstate attribute

SYNOPSIS

#include <pthread.h>
int **pthread_attr_init**(pthread_attr_t * *attr*);

int **pthread_attr_destroy**(pthread_attr_t * *attr*);

int **pthread_attr_setstacksize**(pthread_attr_t * *attr*, size_t *stacksize*);

int **pthread_attr_getstacksize**(const pthread_attr_t * *attr*, size_t * *stacksize*);

int **pthread_attr_setstackaddr**(pthread_attr_t * *attr*, void * *stackaddr*);

int **pthread_attr_getstackaddr**(const pthread_attr_t * *attr*, void ** *stackaddr*);

int **pthread_attr_setdetachstate**(pthread_attr_t * *attr*, int *detachstate*);

int **pthread_attr_getdetachstate**(const pthread_attr_t * *attr*, int * *detachstate*);

DESCRIPTION

The *pthread_attr_init* function initializes the thread attribute object referenced by *attr* with the default values for all the individual thread attributes. The resulting attribute object may be modified by setting individual attribute values. When subsequently used by *pthread_create* , it defines the attributes of the newly created thread. A single attribute object can be used in multiple simultaneous calls to *pthread_create* . Modification of an attribute object has no effect on threads already created using that object.

The *pthread_attr_destroy* function is used to delete a thread attribute object.

The complete list of thread creation attributes follows, with defaults indicated.

stacksize	PTHREAD_STACK_MIN
stackaddr	*stack dynamically allocated*
detachstate	PTHREAD_CREATE_JOINABLE
contentionscope	PTHREAD_SCOPE_SYSTEM
inheritsched	PTHREAD_INHERIT_SCHED
schedpolicy	*default schedpolicy (see mrtp(*POSIX*))*
schedparam	*default schedparam (see mrtp(*POSIX*))*

The latter four attributes, which pertain to scheduling, are described in *pthread_attr_setscope* (3POSIX).

The *stacksize* attribute defines the minimum stack size (in bytes). The *pthread_attr_setstacksize* and *pthread_attr_getstacksize* functions respectively set and get the value of the *stacksize* attribute in the thread attribute object referenced by *attr* .

The *stackaddr* attribute specifies the location in memory to be used for the newly created thread's stack. If no *stackaddr* value is provided explicitly, the stack for the new thread will be allocated dynamically by the system. The *pthread_attr_setstackaddr* and *pthread_attr_getstackaddr* functions respectively set and get the value of the *stackaddr* attribute in the *attr* object.

The *detachstate* attribute controls the behavior of a new thread when using the *pthread_join* function. If the *detachstate* attribute is set to PTHREAD_CREATE_JOINABLE (the default), the identifier of the thread may be used as the target of a *pthread_join* . If *detachstate* is set to PTHREAD_CREATE_DETACHED, all resources associated with the thread are freed immediately on exit, and the thread identifier may not be used in a *pthread_join* (see *pthread_exit* (3POSIX), *pthread_join* (3POSIX)). The *pthread_attr_setdetachstate* and *pthread_attr_getdetachstate* functions respectively set and get the *detachstate* attribute in the *attr* object.

RESTRICTIONS

The *pthread_attr_setstackaddr* and *pthread_attr_getstackaddr* functions are available only in user mode. In supervisor actors the *stackaddr* attribute is not defined, and the stack for a new thread is always provided by the system.

RETURN VALUE

Upon successful completion, all calls listed above return zero. Otherwise, an error code is returned. (NOTE: These calls do not set *errno* .)

ERRORS

[EINVAL]	The *stacksize* argument to *pthread_attr_setstacksize* is less than PTHREAD_STACK_MIN. The *detachstate* argument to *pthread_attr_setdetachstate* contains a value other than PTHREAD_CREATE_JOINABLE or PTHREAD_CREATE_DETACHED.
[ENOSYS]	*pthread_attr_setstackaddr* and *pthread_attr_getstackaddr* are not supported for supervisor mode actors.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`pthread_create`(3POSIX)

NAME

pthread_attr_init, pthread_attr_destroy, pthread_attr_setstacksize, pthread_attr_getstacksize, pthread_attr_setstackaddr, pthread_attr_getstackaddr, pthread_attr_setdetachstate, pthread_attr_getdetachstate – Initialize a thread attribute object; Destroy a thread attribute object; Set the stacksize attribute; Get the stacksize attribute; Set the stackaddr attribute; Get the stackaddr attribute; Set the detachstate attribute; Get the detachstate attribute

SYNOPSIS

#include <pthread.h>
int **pthread_attr_init**(pthread_attr_t * *attr*);

int **pthread_attr_destroy**(pthread_attr_t * *attr*);

int **pthread_attr_setstacksize**(pthread_attr_t * *attr*, size_t *stacksize*);

int **pthread_attr_getstacksize**(const pthread_attr_t * *attr*, size_t * *stacksize*);

int **pthread_attr_setstackaddr**(pthread_attr_t * *attr*, void * *stackaddr*);

int **pthread_attr_getstackaddr**(const pthread_attr_t * *attr*, void ** *stackaddr*);

int **pthread_attr_setdetachstate**(pthread_attr_t * *attr*, int *detachstate*);

int **pthread_attr_getdetachstate**(const pthread_attr_t * *attr*, int * *detachstate*);

DESCRIPTION

The *pthread_attr_init* function initializes the thread attribute object referenced by *attr* with the default values for all the individual thread attributes. The resulting attribute object may be modified by setting individual attribute values. When subsequently used by *pthread_create* , it defines the attributes of the newly created thread. A single attribute object can be used in multiple simultaneous calls to *pthread_create* . Modification of an attribute object has no effect on threads already created using that object.

The *pthread_attr_destroy* function is used to delete a thread attribute object.

The complete list of thread creation attributes follows, with defaults indicated.

stacksize	PTHREAD_STACK_MIN
stackaddr	*stack dynamically allocated*
detachstate	PTHREAD_CREATE_JOINABLE
contentionscope	PTHREAD_SCOPE_SYSTEM
inheritsched	PTHREAD_INHERIT_SCHED
schedpolicy	*default schedpolicy (see mrtp(*POSIX*))*
schedparam	*default schedparam (see mrtp(*POSIX*))*

The latter four attributes, which pertain to scheduling, are described in *pthread_attr_setscope* (3POSIX).

The *stacksize* attribute defines the minimum stack size (in bytes). The *pthread_attr_setstacksize* and *pthread_attr_getstacksize* functions respectively set and get the value of the *stacksize* attribute in the thread attribute object referenced by *attr* .

The *stackaddr* attribute specifies the location in memory to be used for the newly created thread's stack. If no *stackaddr* value is provided explicitly, the stack for the new thread will be allocated dynamically by the system. The *pthread_attr_setstackaddr* and *pthread_attr_getstackaddr* functions respectively set and get the value of the *stackaddr* attribute in the *attr* object.

The *detachstate* attribute controls the behavior of a new thread when using the *pthread_join* function. If the *detachstate* attribute is set to PTHREAD_CREATE_JOINABLE (the default), the identifier of the thread may be used as the target of a *pthread_join* . If *detachstate* is set to PTHREAD_CREATE_DETACHED, all resources associated with the thread are freed immediately on exit, and the thread identifier may not be used in a *pthread_join* (see *pthread_exit* (3POSIX), *pthread_join* (3POSIX)). The *pthread_attr_setdetachstate* and *pthread_attr_getdetachstate* functions respectively set and get the *detachstate* attribute in the *attr* object.

RESTRICTIONS

The *pthread_attr_setstackaddr* and *pthread_attr_getstackaddr* functions are available only in user mode. In supervisor actors the *stackaddr* attribute is not defined, and the stack for a new thread is always provided by the system.

RETURN VALUE

Upon successful completion, all calls listed above return zero. Otherwise, an error code is returned. (NOTE: These calls do not set *errno* .)

ERRORS

[EINVAL]	The *stacksize* argument to *pthread_attr_setstacksize* is less than PTHREAD_STACK_MIN. The *detachstate* argument to *pthread_attr_setdetachstate* contains a value other than PTHREAD_CREATE_JOINABLE or PTHREAD_CREATE_DETACHED.
[ENOSYS]	*pthread_attr_setstackaddr* and *pthread_attr_getstackaddr* are not supported for supervisor mode actors.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`pthread_create`(3POSIX)

NAME

pthread_attr_setscope, pthread_attr_getscope, pthread_attr_setinheritsched, pthread_attr_getinheritsched, pthread_attr_setschedpolicy, pthread_attr_getschedpolicy, pthread_attr_setschedparam, pthread_attr_getschedparam – Set the contention scope attribute; Get the contention scope attribute; Set the scheduling inheritance attribute; Get the scheduling inheritance attribute; Set the scheduling policy attribute; Get the scheduling policy attribute; Set the scheduling parameter attribute; Get the scheduling parameter attribute

SYNOPSIS

#include <pthread.h>

int **pthread_attr_setscope**(pthread_attr_t * *attr*, int *contentionscope*);

int **pthread_attr_getscope**(const pthread_attr_t * *attr*, int * *contentionscope*);

int **pthread_attr_setinheritsched**(pthread_attr_t * *attr*, int *inheritsched*);

int **pthread_attr_getinheritsched**(const pthread_attr_t * *attr*, int * *inheritsched*);

int **pthread_attr_setschedpolicy**(pthread_attr_t * *attr*, int *policy*);

int **pthread_attr_getschedpolicy**(const pthread_attr_t * *attr*, int * *policy*);

int **pthread_attr_setschedparam**(pthread_attr_t * *attr*, const struct sched_param * *param*);

int **pthread_attr_getschedparam**(const pthread_attr_t * *attr*, struct sched_param * *param*);

DESCRIPTION

Thread creation attributes and their defaults are summarized in *pthread_attr_init* (3POSIX).

The *contentionscope* attribute specifies the scope of thread scheduling decisions relative to a pthread. In the CHORUS / POSIX Micro Realtime Profile, the only *contentionscope* value supported is PTHREAD_SCOPE_SYSTEM, which means that the thread competes directly with all other threads on the site for processor resources. The *pthread_attr_setscope* and *pthread_attr_getscope* functions respectively set and get the *contentionscope* attribute in the thread creation attribute object designated by *attr* .

The *inheritsched* attribute controls the initialization of scheduling parameters in the newly created thread:

PTHREAD_INHERIT_SCHED Specifies that the scheduling policy and associated attributes (for exanmple, priority) are to be inherited from the creating thread, and the corresponding values in the attribute object are to be ignored. This is the default.

PTHREAD_EXPLICIT_SCHED Specifies that the scheduling policy and associated attributes in the new thread are to be

set to the corresponding values from the attribute object.

The *pthread_attr_setinheritsched* and *pthread_attr_getinheritsched* functions respectively set and get the *inheritsched* attribute in the *attr* object.

The *schedpolicy* attribute specifies the thread scheduling policy. Supported values include SCHED_RR, SCHED_FIFO, and SCHED_OTHER which is equivalent to SCHED_RR (see *mrtp* (3POSIX) for information on scheduling). The *pthread_attr_setschedpolicy* and *pthread_attr_getschedpolicy* functions respectively set and get the *schedpolicy* attribute in the *attr* object.

Each POSIX scheduling policy defines a parameter structure *sched_param* containing parameters specific to the policy. For both SCHED_RR and SCHED_FIFO the sole parameter is thread priority, and the structure is defined as follows:

```
struct sched_param {
    int       sched_priority;
};
```

The *pthread_attr_setschedparam* and *pthread_attr_getschedparam* functions respectively set and get the scheduling parameter structure in the *attr* object.

RETURN VALUE

Upon successful completion, all calls listed above return zero. Otherwise an error code is returned. (NOTE: These calls do not set *errno* .)

ERRORS

[EINVAL]	*pthread_attr_setinheritsched* was invoked with an invalid *inherit* argument. *pthread_attr_setschedparam* was invoked with an invalid value for the *sched_priority* member of the *param* argument.
[ENOTSUP]	*pthread_attr_setscope* was invoked with a *contentionscope* argument other than PTHREAD_SCOPE_SYSTEM. *pthread_attr_setschedpolicy* was invoked with an unsupported value for the *policy* argument.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`pthread_attr_init`(3POSIX) , `pthread_create`(3POSIX)

NAME

pthread_attr_setscope, pthread_attr_getscope, pthread_attr_setinheritsched, pthread_attr_getinheritsched, pthread_attr_setschedpolicy, pthread_attr_getschedpolicy, pthread_attr_setschedparam, pthread_attr_getschedparam – Set the contention scope attribute; Get the contention scope attribute; Set the scheduling inheritance attribute; Get the scheduling inheritance attribute; Set the scheduling policy attribute; Get the scheduling policy attribute; Set the scheduling parameter attribute; Get the scheduling parameter attribute

SYNOPSIS

#include <pthread.h>
int **pthread_attr_setscope**(pthread_attr_t * *attr*, int *contentionscope*);

int **pthread_attr_getscope**(const pthread_attr_t * *attr*, int * *contentionscope*);

int **pthread_attr_setinheritsched**(pthread_attr_t * *attr*, int *inheritsched*);

int **pthread_attr_getinheritsched**(const pthread_attr_t * *attr*, int * *inheritsched*);

int **pthread_attr_setschedpolicy**(pthread_attr_t * *attr*, int *policy*);

int **pthread_attr_getschedpolicy**(const pthread_attr_t * *attr*, int * *policy*);

int **pthread_attr_setschedparam**(pthread_attr_t * *attr*, const struct sched_param * *param*);

int **pthread_attr_getschedparam**(const pthread_attr_t * *attr*, struct sched_param * *param*);

DESCRIPTION

Thread creation attributes and their defaults are summarized in *pthread_attr_init* (3POSIX).

The *contentionscope* attribute specifies the scope of thread scheduling decisions relative to a pthread. In the CHORUS / POSIX Micro Realtime Profile, the only *contentionscope* value supported is PTHREAD_SCOPE_SYSTEM, which means that the thread competes directly with all other threads on the site for processor resources. The *pthread_attr_setscope* and *pthread_attr_getscope* functions respectively set and get the *contentionscope* attribute in the thread creation attribute object designated by *attr* .

The *inheritsched* attribute controls the initialization of scheduling parameters in the newly created thread:

PTHREAD_INHERIT_SCHED	Specifies that the scheduling policy and associated attributes (for exanmple, priority) are to be inherited from the creating thread, and the corresponding values in the attribute object are to be ignored. This is the default.
PTHREAD_EXPLICIT_SCHED	Specifies that the scheduling policy and associated attributes in the new thread are to be

set to the corresponding values from the attribute object.

The *pthread_attr_setinheritsched* and *pthread_attr_getinheritsched* functions respectively set and get the *inheritsched* attribute in the *attr* object.

The *schedpolicy* attribute specifies the thread scheduling policy. Supported values include SCHED_RR, SCHED_FIFO, and SCHED_OTHER which is equivalent to SCHED_RR (see *mrtp* (3POSIX) for information on scheduling). The *pthread_attr_setschedpolicy* and *pthread_attr_getschedpolicy* functions respectively set and get the *schedpolicy* attribute in the *attr* object.

Each POSIX scheduling policy defines a parameter structure *sched_param* containing parameters specific to the policy. For both SCHED_RR and SCHED_FIFO the sole parameter is thread priority, and the structure is defined as follows:

```
struct sched_param {
    int      sched_priority;
};
```

The *pthread_attr_setschedparam* and *pthread_attr_getschedparam* functions respectively set and get the scheduling parameter structure in the *attr* object.

RETURN VALUE

Upon successful completion, all calls listed above return zero. Otherwise an error code is returned. (NOTE: These calls do not set *errno* .)

ERRORS

[EINVAL]	*pthread_attr_setinheritsched* was invoked with an invalid *inherit* argument. *pthread_attr_setschedparam* was invoked with an invalid value for the *sched_priority* member of the *param* argument.
[ENOTSUP]	*pthread_attr_setscope* was invoked with a *contentionscope* argument other than PTHREAD_SCOPE_SYSTEM. *pthread_attr_setschedpolicy* was invoked with an unsupported value for the *policy* argument.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`pthread_attr_init`(3POSIX) , `pthread_create`(3POSIX)

NAME

pthread_attr_setscope, pthread_attr_getscope, pthread_attr_setinheritsched, pthread_attr_getinheritsched, pthread_attr_setschedpolicy, pthread_attr_getschedpolicy, pthread_attr_setschedparam, pthread_attr_getschedparam – Set the contention scope attribute; Get the contention scope attribute; Set the scheduling inheritance attribute; Get the scheduling inheritance attribute; Set the scheduling policy attribute; Get the scheduling policy attribute; Set the scheduling parameter attribute; Get the scheduling parameter attribute

SYNOPSIS

#include <pthread.h>
int **pthread_attr_setscope**(pthread_attr_t * *attr*, int *contentionscope*);

int **pthread_attr_getscope**(const pthread_attr_t * *attr*, int * *contentionscope*);

int **pthread_attr_setinheritsched**(pthread_attr_t * *attr*, int *inheritsched*);

int **pthread_attr_getinheritsched**(const pthread_attr_t * *attr*, int * *inheritsched*);

int **pthread_attr_setschedpolicy**(pthread_attr_t * *attr*, int *policy*);

int **pthread_attr_getschedpolicy**(const pthread_attr_t * *attr*, int * *policy*);

int **pthread_attr_setschedparam**(pthread_attr_t * *attr*, const struct sched_param * *param*);

int **pthread_attr_getschedparam**(const pthread_attr_t * *attr*, struct sched_param * *param*);

DESCRIPTION

Thread creation attributes and their defaults are summarized in *pthread_attr_init* (3POSIX).

The *contentionscope* attribute specifies the scope of thread scheduling decisions relative to a pthread. In the CHORUS / POSIX Micro Realtime Profile, the only *contentionscope* value supported is PTHREAD_SCOPE_SYSTEM, which means that the thread competes directly with all other threads on the site for processor resources. The *pthread_attr_setscope* and *pthread_attr_getscope* functions respectively set and get the *contentionscope* attribute in the thread creation attribute object designated by *attr* .

The *inheritsched* attribute controls the initialization of scheduling parameters in the newly created thread:

PTHREAD_INHERIT_SCHED Specifies that the scheduling policy and associated attributes (for exanmple, priority) are to be inherited from the creating thread, and the corresponding values in the attribute object are to be ignored. This is the default.

PTHREAD_EXPLICIT_SCHED Specifies that the scheduling policy and associated attributes in the new thread are to be

set to the corresponding values from the attribute object.

The *pthread_attr_setinheritsched* and *pthread_attr_getinheritsched* functions respectively set and get the *inheritsched* attribute in the *attr* object.

The *schedpolicy* attribute specifies the thread scheduling policy. Supported values include SCHED_RR, SCHED_FIFO, and SCHED_OTHER which is equivalent to SCHED_RR (see *mrtp* (3POSIX) for information on scheduling). The *pthread_attr_setschedpolicy* and *pthread_attr_getschedpolicy* functions respectively set and get the *schedpolicy* attribute in the *attr* object.

Each POSIX scheduling policy defines a parameter structure *sched_param* containing parameters specific to the policy. For both SCHED_RR and SCHED_FIFO the sole parameter is thread priority, and the structure is defined as follows:

```
struct sched_param {
    int       sched_priority;
};
```

The *pthread_attr_setschedparam* and *pthread_attr_getschedparam* functions respectively set and get the scheduling parameter structure in the *attr* object.

RETURN VALUE

Upon successful completion, all calls listed above return zero. Otherwise an error code is returned. (NOTE: These calls do not set *errno* .)

ERRORS

[EINVAL]	*pthread_attr_setinheritsched* was invoked with an invalid *inherit* argument. *pthread_attr_setschedparam* was invoked with an invalid value for the *sched_priority* member of the *param* argument.
[ENOTSUP]	*pthread_attr_setscope* was invoked with a *contentionscope* argument other than PTHREAD_SCOPE_SYSTEM. *pthread_attr_setschedpolicy* was invoked with an unsupported value for the *policy* argument.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`pthread_attr_init`(3POSIX) , `pthread_create`(3POSIX)

NAME

pthread_attr_setscope, pthread_attr_getscope, pthread_attr_setinheritsched, pthread_attr_getinheritsched, pthread_attr_setschedpolicy, pthread_attr_getschedpolicy, pthread_attr_setschedparam, pthread_attr_getschedparam – Set the contention scope attribute; Get the contention scope attribute; Set the scheduling inheritance attribute; Get the scheduling inheritance attribute; Set the scheduling policy attribute; Get the scheduling policy attribute; Set the scheduling parameter attribute; Get the scheduling parameter attribute

SYNOPSIS

#include <pthread.h>
int **pthread_attr_setscope**(pthread_attr_t * *attr*, int *contentionscope*);

int **pthread_attr_getscope**(const pthread_attr_t * *attr*, int * *contentionscope*);

int **pthread_attr_setinheritsched**(pthread_attr_t * *attr*, int *inheritsched*);

int **pthread_attr_getinheritsched**(const pthread_attr_t * *attr*, int * *inheritsched*);

int **pthread_attr_setschedpolicy**(pthread_attr_t * *attr*, int *policy*);

int **pthread_attr_getschedpolicy**(const pthread_attr_t * *attr*, int * *policy*);

int **pthread_attr_setschedparam**(pthread_attr_t * *attr*, const struct sched_param * *param*);

int **pthread_attr_getschedparam**(const pthread_attr_t * *attr*, struct sched_param * *param*);

DESCRIPTION

Thread creation attributes and their defaults are summarized in *pthread_attr_init* (3POSIX).

The *contentionscope* attribute specifies the scope of thread scheduling decisions relative to a pthread. In the CHORUS / POSIX Micro Realtime Profile, the only *contentionscope* value supported is PTHREAD_SCOPE_SYSTEM, which means that the thread competes directly with all other threads on the site for processor resources. The *pthread_attr_setscope* and *pthread_attr_getscope* functions respectively set and get the *contentionscope* attribute in the thread creation attribute object designated by *attr* .

The *inheritsched* attribute controls the initialization of scheduling parameters in the newly created thread:

PTHREAD_INHERIT_SCHED	Specifies that the scheduling policy and associated attributes (for exanmple, priority) are to be inherited from the creating thread, and the corresponding values in the attribute object are to be ignored. This is the default.
PTHREAD_EXPLICIT_SCHED	Specifies that the scheduling policy and associated attributes in the new thread are to be

set to the corresponding values from the attribute object.

The *pthread_attr_setinheritsched* and *pthread_attr_getinheritsched* functions respectively set and get the *inheritsched* attribute in the *attr* object.

The *schedpolicy* attribute specifies the thread scheduling policy. Supported values include SCHED_RR, SCHED_FIFO, and SCHED_OTHER which is equivalent to SCHED_RR (see *mrtp* (3POSIX) for information on scheduling). The *pthread_attr_setschedpolicy* and *pthread_attr_getschedpolicy* functions respectively set and get the *schedpolicy* attribute in the *attr* object.

Each POSIX scheduling policy defines a parameter structure *sched_param* containing parameters specific to the policy. For both SCHED_RR and SCHED_FIFO the sole parameter is thread priority, and the structure is defined as follows:

```
struct sched_param {
    int        sched_priority;
};
```

The *pthread_attr_setschedparam* and *pthread_attr_getschedparam* functions respectively set and get the scheduling parameter structure in the *attr* object.

RETURN VALUE

Upon successful completion, all calls listed above return zero. Otherwise an error code is returned. (NOTE: These calls do not set *errno* .)

ERRORS

[EINVAL]	*pthread_attr_setinheritsched* was invoked with an invalid *inherit* argument. *pthread_attr_setschedparam* was invoked with an invalid value for the *sched_priority* member of the *param* argument.
[ENOTSUP]	*pthread_attr_setscope* was invoked with a *contentionscope* argument other than PTHREAD_SCOPE_SYSTEM. *pthread_attr_setschedpolicy* was invoked with an unsupported value for the *policy* argument.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`pthread_attr_init`(3POSIX) , `pthread_create`(3POSIX)

NAME

pthread_attr_init, pthread_attr_destroy, pthread_attr_setstacksize, pthread_attr_getstacksize, pthread_attr_setstackaddr, pthread_attr_getstackaddr, pthread_attr_setdetachstate, pthread_attr_getdetachstate – Initialize a thread attribute object; Destroy a thread attribute object; Set the stacksize attribute; Get the stacksize attribute; Set the stackaddr attribute; Get the stackaddr attribute; Set the detachstate attribute; Get the detachstate attribute

SYNOPSIS

#include <pthread.h>
int **pthread_attr_init**(pthread_attr_t * *attr*);

int **pthread_attr_destroy**(pthread_attr_t * *attr*);

int **pthread_attr_setstacksize**(pthread_attr_t * *attr*, size_t *stacksize*);

int **pthread_attr_getstacksize**(const pthread_attr_t * *attr*, size_t * *stacksize*);

int **pthread_attr_setstackaddr**(pthread_attr_t * *attr*, void * *stackaddr*);

int **pthread_attr_getstackaddr**(const pthread_attr_t * *attr*, void ** *stackaddr*);

int **pthread_attr_setdetachstate**(pthread_attr_t * *attr*, int *detachstate*);

int **pthread_attr_getdetachstate**(const pthread_attr_t * *attr*, int * *detachstate*);

DESCRIPTION

The *pthread_attr_init* function initializes the thread attribute object referenced by *attr* with the default values for all the individual thread attributes. The resulting attribute object may be modified by setting individual attribute values. When subsequently used by *pthread_create* , it defines the attributes of the newly created thread. A single attribute object can be used in multiple simultaneous calls to *pthread_create* . Modification of an attribute object has no effect on threads already created using that object.

The *pthread_attr_destroy* function is used to delete a thread attribute object.

The complete list of thread creation attributes follows, with defaults indicated.

`stacksize`	`PTHREAD_STACK_MIN`
`stackaddr`	*stack dynamically allocated*
`detachstate`	`PTHREAD_CREATE_JOINABLE`
`contentionscope`	`PTHREAD_SCOPE_SYSTEM`
`inheritsched`	`PTHREAD_INHERIT_SCHED`
`schedpolicy`	*default schedpolicy (see mrtp(*`POSIX`*))*
`schedparam`	*default schedparam (see mrtp(*`POSIX`*))*

The latter four attributes, which pertain to scheduling, are described in *pthread_attr_setscope* (3POSIX).

The *stacksize* attribute defines the minimum stack size (in bytes). The *pthread_attr_setstacksize* and *pthread_attr_getstacksize* functions respectively set and get the value of the *stacksize* attribute in the thread attribute object referenced by *attr* .

The *stackaddr* attribute specifies the location in memory to be used for the newly created thread's stack. If no *stackaddr* value is provided explicitly, the stack for the new thread will be allocated dynamically by the system. The *pthread_attr_setstackaddr* and *pthread_attr_getstackaddr* functions respectively set and get the value of the *stackaddr* attribute in the *attr* object.

The *detachstate* attribute controls the behavior of a new thread when using the *pthread_join* function. If the *detachstate* attribute is set to PTHREAD_CREATE_JOINABLE (the default), the identifier of the thread may be used as the target of a *pthread_join* . If *detachstate* is set to PTHREAD_CREATE_DETACHED, all resources associated with the thread are freed immediately on exit, and the thread identifier may not be used in a *pthread_join* (see *pthread_exit* (3POSIX), *pthread_join* (3POSIX)). The *pthread_attr_setdetachstate* and *pthread_attr_getdetachstate* functions respectively set and get the *detachstate* attribute in the *attr* object.

RESTRICTIONS

The *pthread_attr_setstackaddr* and *pthread_attr_getstackaddr* functions are available only in user mode. In supervisor actors the *stackaddr* attribute is not defined, and the stack for a new thread is always provided by the system.

RETURN VALUE

Upon successful completion, all calls listed above return zero. Otherwise, an error code is returned. (NOTE: These calls do not set *errno* .)

ERRORS

[EINVAL]	The *stacksize* argument to *pthread_attr_setstacksize* is less than PTHREAD_STACK_MIN. The *detachstate* argument to *pthread_attr_setdetachstate* contains a value other than PTHREAD_CREATE_JOINABLE or PTHREAD_CREATE_DETACHED.
[ENOSYS]	*pthread_attr_setstackaddr* and *pthread_attr_getstackaddr* are not supported for supervisor mode actors.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`pthread_create`(3POSIX)

NAME

pthread_attr_init, pthread_attr_destroy, pthread_attr_setstacksize, pthread_attr_getstacksize, pthread_attr_setstackaddr, pthread_attr_getstackaddr, pthread_attr_setdetachstate, pthread_attr_getdetachstate – Initialize a thread attribute object; Destroy a thread attribute object; Set the stacksize attribute; Get the stacksize attribute; Set the stackaddr attribute; Get the stackaddr attribute; Set the detachstate attribute; Get the detachstate attribute

SYNOPSIS

#include <pthread.h>
int **pthread_attr_init**(pthread_attr_t * *attr*);

int **pthread_attr_destroy**(pthread_attr_t * *attr*);

int **pthread_attr_setstacksize**(pthread_attr_t * *attr*, size_t *stacksize*);

int **pthread_attr_getstacksize**(const pthread_attr_t * *attr*, size_t * *stacksize*);

int **pthread_attr_setstackaddr**(pthread_attr_t * *attr*, void * *stackaddr*);

int **pthread_attr_getstackaddr**(const pthread_attr_t * *attr*, void ** *stackaddr*);

int **pthread_attr_setdetachstate**(pthread_attr_t * *attr*, int *detachstate*);

int **pthread_attr_getdetachstate**(const pthread_attr_t * *attr*, int * *detachstate*);

DESCRIPTION

The *pthread_attr_init* function initializes the thread attribute object referenced by *attr* with the default values for all the individual thread attributes. The resulting attribute object may be modified by setting individual attribute values. When subsequently used by *pthread_create* , it defines the attributes of the newly created thread. A single attribute object can be used in multiple simultaneous calls to *pthread_create* . Modification of an attribute object has no effect on threads already created using that object.

The *pthread_attr_destroy* function is used to delete a thread attribute object.

The complete list of thread creation attributes follows, with defaults indicated.

stacksize	PTHREAD_STACK_MIN
stackaddr	*stack dynamically allocated*
detachstate	PTHREAD_CREATE_JOINABLE
contentionscope	PTHREAD_SCOPE_SYSTEM
inheritsched	PTHREAD_INHERIT_SCHED
schedpolicy	*default schedpolicy (see mrtp(*POSIX*))*
schedparam	*default schedparam (see mrtp(*POSIX*))*

The latter four attributes, which pertain to scheduling, are described in *pthread_attr_setscope* (3POSIX).

The *stacksize* attribute defines the minimum stack size (in bytes). The *pthread_attr_setstacksize* and *pthread_attr_getstacksize* functions respectively set and get the value of the *stacksize* attribute in the thread attribute object referenced by *attr* .

The *stackaddr* attribute specifies the location in memory to be used for the newly created thread's stack. If no *stackaddr* value is provided explicitly, the stack for the new thread will be allocated dynamically by the system. The *pthread_attr_setstackaddr* and *pthread_attr_getstackaddr* functions respectively set and get the value of the *stackaddr* attribute in the *attr* object.

The *detachstate* attribute controls the behavior of a new thread when using the *pthread_join* function. If the *detachstate* attribute is set to PTHREAD_CREATE_JOINABLE (the default), the identifier of the thread may be used as the target of a *pthread_join* . If *detachstate* is set to PTHREAD_CREATE_DETACHED, all resources associated with the thread are freed immediately on exit, and the thread identifier may not be used in a *pthread_join* (see *pthread_exit* (3POSIX), *pthread_join* (3POSIX)). The *pthread_attr_setdetachstate* and *pthread_attr_getdetachstate* functions respectively set and get the *detachstate* attribute in the *attr* object.

RESTRICTIONS

The *pthread_attr_setstackaddr* and *pthread_attr_getstackaddr* functions are available only in user mode. In supervisor actors the *stackaddr* attribute is not defined, and the stack for a new thread is always provided by the system.

RETURN VALUE

Upon successful completion, all calls listed above return zero. Otherwise, an error code is returned. (NOTE: These calls do not set *errno* .)

ERRORS

[EINVAL]	The *stacksize* argument to *pthread_attr_setstacksize* is less than PTHREAD_STACK_MIN. The *detachstate* argument to *pthread_attr_setdetachstate* contains a value other than PTHREAD_CREATE_JOINABLE or PTHREAD_CREATE_DETACHED.
[ENOSYS]	*pthread_attr_setstackaddr* and *pthread_attr_getstackaddr* are not supported for supervisor mode actors.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`pthread_create`(3POSIX)

NAME

pthread_condattr_init, pthread_condattr_destroy – initialize or destroy a condition variable attribute object

SYNOPSIS

```
#include <pthread.h>
int pthread_condattr_init(pthread_condattr_t * attr);

int pthread_condattr_destroy(pthread_condattr_t * attr);
```

DESCRIPTION

Warning: condition variable attributes are not currently supported.

The *pthread_condattr_init* function initializes the condition variable attribute object referenced by *attr* with the default values for all condition variable attributes. When subsequently used by *pthread_cond_init* , it specifies attributes for the condition variable being initialized. Modification of an attribute object has no effect on condition variables already initialized from that object.

The *pthread_condattr_destroy* function deletes a condition variable attribute object.

RETURN VALUE

The *pthread_condattr_init* and *pthread_condattr_destroy* functions always return zero.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`pthread_cond_init`(3POSIX)

NAME

pthread_condattr_init, pthread_condattr_destroy – initialize or destroy a condition variable attribute object

SYNOPSIS

#include <pthread.h>
int **pthread_condattr_init**(pthread_condattr_t * *attr*);

int **pthread_condattr_destroy**(pthread_condattr_t * *attr*);

DESCRIPTION

Warning: condition variable attributes are not currently supported.

The *pthread_condattr_init* function initializes the condition variable attribute object referenced by *attr* with the default values for all condition variable attributes. When subsequently used by *pthread_cond_init* , it specifies attributes for the condition variable being initialized. Modification of an attribute object has no effect on condition variables already initialized from that object.

The *pthread_condattr_destroy* function deletes a condition variable attribute object.

RETURN VALUE

The *pthread_condattr_init* and *pthread_condattr_destroy* functions always return zero.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`pthread_cond_init`(3POSIX)

NAME

pthread_cond_init, pthread_cond_destroy, pthread_cond_signal, pthread_cond_broadcast, pthread_cond_wait, pthread_cond_timedwait – initialize and use a condition variable

SYNOPSIS

#include <pthread.h>
#include <time.h>
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int **`pthread_cond_init`**(pthread_cond_t * *cond*, const pthread_condattr_t * *attr*);

int **`pthread_cond_destroy`**(pthread_cond_t * *cond*);

int **`pthread_cond_signal`**(pthread_cond_t * *cond*);

int **`pthread_cond_broadcast`**(pthread_cond_t * *cond*);

int **`pthread_cond_wait`**(pthread_cond_t * *cond*, pthread_mutex_t * *mutex*);

int **`pthread_cond_timedwait`**(pthread_cond_t * *cond*, pthread_mutex_t * *mutex*, const struct timespec * *abstime*);

DESCRIPTION

The *pthread_cond_init* function initializes the condition variable referenced by *cond* with attributes obtained from *attr* . If *attr* is NULL, the default condition variable attributes are used.

A condition variable that is statically allocated may be initialized with the initializer macro PTHREAD_COND_INITIALIZER. The effect is equivalent to dynamic initialization using *pthread_cond_init* with *attr* equal to NULL, except that no error checks are performed.

The *pthread_cond_destroy* function deletes the condition variable designated by *cond* , if there are no threads currently blocked on it. Otherwise, *pthread_cond_destroy* returns an error.

The *pthread_cond_signal* function awakens one or more threads blocked on the condition variable *cond* . If there are multiple threads blocked on *cond* , they will be awakened in the order they blocked. The *pthread_cond_broadcast* function awakens all threads blocked on the condition variable *cond* .

Neither *pthread_cond_signal* nor *pthread_cond_broadcast* has any effect on the condition variable if there are no threads currently blocked on it.

The *pthread_cond_wait* and *pthread_cond_timedwait* functions cause a wait on the condition variable *cond* . They must be called with the mutex referenced by *mutex* locked by the calling thread. These functions atomically release *mutex* and block the calling thread. It is impossible for any other thread to acquire the mutex and invoke *pthread_cond_signal* or *pthread_cond_broadcast* so that those operations take effect between the release of the mutex and the block of the thread calling *pthread_cond_wait* or *pthread_cond_timedwait* .

Upon successful return from *pthread_cond_wait* or *pthread_cond_timedwait* , the mutex will have been re-acquired by the calling thread.

The *pthread_cond_timedwait* function is the same as *pthread_cond_wait* except that an error is returned in the following cases:

- if the absolute time specified by *abstime* expires (that is, system time as measured by the realtime clock equals or exceeds *abstime*) before the condition variable *cond* is signaled or broadcast
- if the time specified by *abstime* has already expired at the time of the call

If a timeout occurs, *pthread_cond_timedwait* will nonetheless release and reacquire the mutex *mutex* .

A condition variable wait is subject to spurious wakeups. Within the user program, a boolean predicate is usually associated with a condition variable. The predicate involves shared variables and is tested with the mutex locked. If it is found to be false, the thread blocks using a condition wait. As the return from *pthread_cond_wait* or *pthread_cond_timedwait* does not confirm anything about the value of the predicate, it is crucial to re-evaluate the predicate and be prepared to wait again if it is still false.

In effect, *pthread_cond_wait* and *pthread_cond_timedwait* establish a dynamic binding between a condition variable and a particular mutex. The effect of using more than one mutex for concurrent condition waits on the same condition variable is undefined.

RETURN VALUE

Upon successful completion, all functions listed above return zero. Otherwise an error code is returned. (NOTE: These calls do not set *errno* .)

ERRORS

Except for ETIMEDOUT, all the error checks indicated are performed at the beginning of processing for the function, and error returns are made before any change is made to the state of any condition variable or mutex argument.

[EINVAL]	The *cond* argument does not refer to a valid condition variable. The *mutex* argument does not refer to a valid mutex (*pthread_cond_wait* and *pthread_cond_timedwait* only). The *abstime* argument refers to an unusable time value (*pthread_cond_timedwait* only).
[EBUSY]	The condition variable *cond* cannot be deleted because one or more threads are currently blocked on it (*pthread_cond_destroy* only).
[ETIMEDOUT]	The time specified by *abstime* has passed (*pthread_cond_timedwait* only).

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

pthread_cond_init, pthread_cond_destroy, pthread_cond_signal, pthread_cond_broadcast, pthread_cond_wait, pthread_cond_timedwait – initialize and use a condition variable

SYNOPSIS

#include <pthread.h>
#include <time.h>
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int **pthread_cond_init**(pthread_cond_t * *cond*, const pthread_condattr_t * *attr*);

int **pthread_cond_destroy**(pthread_cond_t * *cond*);

int **pthread_cond_signal**(pthread_cond_t * *cond*);

int **pthread_cond_broadcast**(pthread_cond_t * *cond*);

int **pthread_cond_wait**(pthread_cond_t * *cond*, pthread_mutex_t * *mutex*);

int **pthread_cond_timedwait**(pthread_cond_t * *cond*, pthread_mutex_t * *mutex*, const struct timespec * *abstime*);

DESCRIPTION

The *pthread_cond_init* function initializes the condition variable referenced by *cond* with attributes obtained from *attr* . If *attr* is NULL, the default condition variable attributes are used.

A condition variable that is statically allocated may be initialized with the initializer macro PTHREAD_COND_INITIALIZER. The effect is equivalent to dynamic initialization using *pthread_cond_init* with *attr* equal to NULL, except that no error checks are performed.

The *pthread_cond_destroy* function deletes the condition variable designated by *cond* , if there are no threads currently blocked on it. Otherwise, *pthread_cond_destroy* returns an error.

The *pthread_cond_signal* function awakens one or more threads blocked on the condition variable *cond* . If there are multiple threads blocked on *cond* , they will be awakened in the order they blocked. The *pthread_cond_broadcast* function awakens all threads blocked on the condition variable *cond* .

Neither *pthread_cond_signal* nor *pthread_cond_broadcast* has any effect on the condition variable if there are no threads currently blocked on it.

The *pthread_cond_wait* and *pthread_cond_timedwait* functions cause a wait on the condition variable *cond* . They must be called with the mutex referenced by *mutex* locked by the calling thread. These functions atomically release *mutex* and block the calling thread. It is impossible for any other thread to acquire the mutex and invoke *pthread_cond_signal* or *pthread_cond_broadcast* so that those operations take effect between the release of the mutex and the block of the thread calling *pthread_cond_wait* or *pthread_cond_timedwait* .

Upon successful return from *pthread_cond_wait* or *pthread_cond_timedwait* , the mutex will have been re-acquired by the calling thread.

The *pthread_cond_timedwait* function is the same as *pthread_cond_wait* except that an error is returned in the following cases:

- if the absolute time specified by *abstime* expires (that is, system time as measured by the realtime clock equals or exceeds *abstime*) before the condition variable *cond* is signaled or broadcast
- if the time specified by *abstime* has already expired at the time of the call

If a timeout occurs, *pthread_cond_timedwait* will nonetheless release and reacquire the mutex *mutex* .

A condition variable wait is subject to spurious wakeups. Within the user program, a boolean predicate is usually associated with a condition variable. The predicate involves shared variables and is tested with the mutex locked. If it is found to be false, the thread blocks using a condition wait. As the return from *pthread_cond_wait* or *pthread_cond_timedwait* does not confirm anything about the value of the predicate, it is crucial to re-evaluate the predicate and be prepared to wait again if it is still false.

In effect, *pthread_cond_wait* and *pthread_cond_timedwait* establish a dynamic binding between a condition variable and a particular mutex. The effect of using more than one mutex for concurrent condition waits on the same condition variable is undefined.

RETURN VALUE

Upon successful completion, all functions listed above return zero. Otherwise an error code is returned. (NOTE: These calls do not set *errno* .)

ERRORS

Except for ETIMEDOUT, all the error checks indicated are performed at the beginning of processing for the function, and error returns are made before any change is made to the state of any condition variable or mutex argument.

[EINVAL]	The *cond* argument does not refer to a valid condition variable. The *mutex* argument does not refer to a valid mutex (*pthread_cond_wait* and *pthread_cond_timedwait* only). The *abstime* argument refers to an unusable time value (*pthread_cond_timedwait* only).
[EBUSY]	The condition variable *cond* cannot be deleted because one or more threads are currently blocked on it (*pthread_cond_destroy* only).
[ETIMEDOUT]	The time specified by *abstime* has passed (*pthread_cond_timedwait* only).

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

pthread_cond_init, pthread_cond_destroy, pthread_cond_signal, pthread_cond_broadcast, pthread_cond_wait, pthread_cond_timedwait – initialize and use a condition variable

SYNOPSIS

#include <pthread.h>
#include <time.h>
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int **pthread_cond_init**(pthread_cond_t * *cond*, const pthread_condattr_t * *attr*);

int **pthread_cond_destroy**(pthread_cond_t * *cond*);

int **pthread_cond_signal**(pthread_cond_t * *cond*);

int **pthread_cond_broadcast**(pthread_cond_t * *cond*);

int **pthread_cond_wait**(pthread_cond_t * *cond*, pthread_mutex_t * *mutex*);

int **pthread_cond_timedwait**(pthread_cond_t * *cond*, pthread_mutex_t * *mutex*, const struct timespec * *abstime*);

DESCRIPTION

The *pthread_cond_init* function initializes the condition variable referenced by *cond* with attributes obtained from *attr* . If *attr* is NULL, the default condition variable attributes are used.

A condition variable that is statically allocated may be initialized with the initializer macro PTHREAD_COND_INITIALIZER. The effect is equivalent to dynamic initialization using *pthread_cond_init* with *attr* equal to NULL, except that no error checks are performed.

The *pthread_cond_destroy* function deletes the condition variable designated by *cond* , if there are no threads currently blocked on it. Otherwise, *pthread_cond_destroy* returns an error.

The *pthread_cond_signal* function awakens one or more threads blocked on the condition variable *cond* . If there are multiple threads blocked on *cond* , they will be awakened in the order they blocked. The *pthread_cond_broadcast* function awakens all threads blocked on the condition variable *cond* .

Neither *pthread_cond_signal* nor *pthread_cond_broadcast* has any effect on the condition variable if there are no threads currently blocked on it.

The *pthread_cond_wait* and *pthread_cond_timedwait* functions cause a wait on the condition variable *cond* . They must be called with the mutex referenced by *mutex* locked by the calling thread. These functions atomically release *mutex* and block the calling thread. It is impossible for any other thread to acquire the mutex and invoke *pthread_cond_signal* or *pthread_cond_broadcast* so that those operations take effect between the release of the mutex and the block of the thread calling *pthread_cond_wait* or *pthread_cond_timedwait* .

Upon successful return from *pthread_cond_wait* or *pthread_cond_timedwait* , the mutex will have been re-acquired by the calling thread.

The *pthread_cond_timedwait* function is the same as *pthread_cond_wait* except that an error is returned in the following cases:

- if the absolute time specified by *abstime* expires (that is, system time as measured by the realtime clock equals or exceeds *abstime*) before the condition variable *cond* is signaled or broadcast
- if the time specified by *abstime* has already expired at the time of the call

If a timeout occurs, *pthread_cond_timedwait* will nonetheless release and reacquire the mutex *mutex* .

A condition variable wait is subject to spurious wakeups. Within the user program, a boolean predicate is usually associated with a condition variable. The predicate involves shared variables and is tested with the mutex locked. If it is found to be false, the thread blocks using a condition wait. As the return from *pthread_cond_wait* or *pthread_cond_timedwait* does not confirm anything about the value of the predicate, it is crucial to re-evaluate the predicate and be prepared to wait again if it is still false.

In effect, *pthread_cond_wait* and *pthread_cond_timedwait* establish a dynamic binding between a condition variable and a particular mutex. The effect of using more than one mutex for concurrent condition waits on the same condition variable is undefined.

RETURN VALUE

Upon successful completion, all functions listed above return zero. Otherwise an error code is returned. (NOTE: These calls do not set *errno* .)

ERRORS

Except for ETIMEDOUT, all the error checks indicated are performed at the beginning of processing for the function, and error returns are made before any change is made to the state of any condition variable or mutex argument.

[EINVAL]	The *cond* argument does not refer to a valid condition variable. The *mutex* argument does not refer to a valid mutex (*pthread_cond_wait* and *pthread_cond_timedwait* only). The *abstime* argument refers to an unusable time value (*pthread_cond_timedwait* only).
[EBUSY]	The condition variable *cond* cannot be deleted because one or more threads are currently blocked on it (*pthread_cond_destroy* only).
[ETIMEDOUT]	The time specified by *abstime* has passed (*pthread_cond_timedwait* only).

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

pthread_cond_init, pthread_cond_destroy, pthread_cond_signal, pthread_cond_broadcast, pthread_cond_wait, pthread_cond_timedwait – initialize and use a condition variable

SYNOPSIS

#include <pthread.h>
#include <time.h>
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int **pthread_cond_init**(pthread_cond_t * *cond*, const pthread_condattr_t * *attr*);

int **pthread_cond_destroy**(pthread_cond_t * *cond*);

int **pthread_cond_signal**(pthread_cond_t * *cond*);

int **pthread_cond_broadcast**(pthread_cond_t * *cond*);

int **pthread_cond_wait**(pthread_cond_t * *cond*, pthread_mutex_t * *mutex*);

int **pthread_cond_timedwait**(pthread_cond_t * *cond*, pthread_mutex_t * *mutex*, const struct timespec * *abstime*);

DESCRIPTION

The *pthread_cond_init* function initializes the condition variable referenced by *cond* with attributes obtained from *attr* . If *attr* is NULL, the default condition variable attributes are used.

A condition variable that is statically allocated may be initialized with the initializer macro PTHREAD_COND_INITIALIZER. The effect is equivalent to dynamic initialization using *pthread_cond_init* with *attr* equal to NULL, except that no error checks are performed.

The *pthread_cond_destroy* function deletes the condition variable designated by *cond* , if there are no threads currently blocked on it. Otherwise, *pthread_cond_destroy* returns an error.

The *pthread_cond_signal* function awakens one or more threads blocked on the condition variable *cond* . If there are multiple threads blocked on *cond* , they will be awakened in the order they blocked. The *pthread_cond_broadcast* function awakens all threads blocked on the condition variable *cond* .

Neither *pthread_cond_signal* nor *pthread_cond_broadcast* has any effect on the condition variable if there are no threads currently blocked on it.

The *pthread_cond_wait* and *pthread_cond_timedwait* functions cause a wait on the condition variable *cond* . They must be called with the mutex referenced by *mutex* locked by the calling thread. These functions atomically release *mutex* and block the calling thread. It is impossible for any other thread to acquire the mutex and invoke *pthread_cond_signal* or *pthread_cond_broadcast* so that those operations take effect between the release of the mutex and the block of the thread calling *pthread_cond_wait* or *pthread_cond_timedwait* .

Upon successful return from *pthread_cond_wait* or *pthread_cond_timedwait* , the mutex will have been re-acquired by the calling thread.

The *pthread_cond_timedwait* function is the same as *pthread_cond_wait* except that an error is returned in the following cases:

- if the absolute time specified by *abstime* expires (that is, system time as measured by the realtime clock equals or exceeds *abstime*) before the condition variable *cond* is signaled or broadcast
- if the time specified by *abstime* has already expired at the time of the call

If a timeout occurs, *pthread_cond_timedwait* will nonetheless release and reacquire the mutex *mutex* .

A condition variable wait is subject to spurious wakeups. Within the user program, a boolean predicate is usually associated with a condition variable. The predicate involves shared variables and is tested with the mutex locked. If it is found to be false, the thread blocks using a condition wait. As the return from *pthread_cond_wait* or *pthread_cond_timedwait* does not confirm anything about the value of the predicate, it is crucial to re-evaluate the predicate and be prepared to wait again if it is still false.

In effect, *pthread_cond_wait* and *pthread_cond_timedwait* establish a dynamic binding between a condition variable and a particular mutex. The effect of using more than one mutex for concurrent condition waits on the same condition variable is undefined.

RETURN VALUE

Upon successful completion, all functions listed above return zero. Otherwise an error code is returned. (NOTE: These calls do not set *errno* .)

ERRORS

Except for ETIMEDOUT, all the error checks indicated are performed at the beginning of processing for the function, and error returns are made before any change is made to the state of any condition variable or mutex argument.

[EINVAL]	The *cond* argument does not refer to a valid condition variable. The *mutex* argument does not refer to a valid mutex (*pthread_cond_wait* and *pthread_cond_timedwait* only). The *abstime* argument refers to an unusable time value (*pthread_cond_timedwait* only).
[EBUSY]	The condition variable *cond* cannot be deleted because one or more threads are currently blocked on it (*pthread_cond_destroy* only).
[ETIMEDOUT]	The time specified by *abstime* has passed (*pthread_cond_timedwait* only).

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

pthread_cond_init, pthread_cond_destroy, pthread_cond_signal, pthread_cond_broadcast, pthread_cond_wait, pthread_cond_timedwait – initialize and use a condition variable

SYNOPSIS

#include <pthread.h>
#include <time.h>
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int **pthread_cond_init**(pthread_cond_t * *cond*, const pthread_condattr_t * *attr*);

int **pthread_cond_destroy**(pthread_cond_t * *cond*);

int **pthread_cond_signal**(pthread_cond_t * *cond*);

int **pthread_cond_broadcast**(pthread_cond_t * *cond*);

int **pthread_cond_wait**(pthread_cond_t * *cond*, pthread_mutex_t * *mutex*);

int **pthread_cond_timedwait**(pthread_cond_t * *cond*, pthread_mutex_t * *mutex*, const struct timespec * *abstime*);

DESCRIPTION

The *pthread_cond_init* function initializes the condition variable referenced by *cond* with attributes obtained from *attr* . If *attr* is NULL, the default condition variable attributes are used.

A condition variable that is statically allocated may be initialized with the initializer macro PTHREAD_COND_INITIALIZER. The effect is equivalent to dynamic initialization using *pthread_cond_init* with *attr* equal to NULL, except that no error checks are performed.

The *pthread_cond_destroy* function deletes the condition variable designated by *cond* , if there are no threads currently blocked on it. Otherwise, *pthread_cond_destroy* returns an error.

The *pthread_cond_signal* function awakens one or more threads blocked on the condition variable *cond* . If there are multiple threads blocked on *cond* , they will be awakened in the order they blocked. The *pthread_cond_broadcast* function awakens all threads blocked on the condition variable *cond* .

Neither *pthread_cond_signal* nor *pthread_cond_broadcast* has any effect on the condition variable if there are no threads currently blocked on it.

The *pthread_cond_wait* and *pthread_cond_timedwait* functions cause a wait on the condition variable *cond* . They must be called with the mutex referenced by *mutex* locked by the calling thread. These functions atomically release *mutex* and block the calling thread. It is impossible for any other thread to acquire the mutex and invoke *pthread_cond_signal* or *pthread_cond_broadcast* so that those operations take effect between the release of the mutex and the block of the thread calling *pthread_cond_wait* or *pthread_cond_timedwait* .

Upon successful return from *pthread_cond_wait* or *pthread_cond_timedwait* , the mutex will have been re-acquired by the calling thread.

The *pthread_cond_timedwait* function is the same as *pthread_cond_wait* except that an error is returned in the following cases:

- if the absolute time specified by *abstime* expires (that is, system time as measured by the realtime clock equals or exceeds *abstime*) before the condition variable *cond* is signaled or broadcast
- if the time specified by *abstime* has already expired at the time of the call

If a timeout occurs, *pthread_cond_timedwait* will nonetheless release and reacquire the mutex *mutex* .

A condition variable wait is subject to spurious wakeups. Within the user program, a boolean predicate is usually associated with a condition variable. The predicate involves shared variables and is tested with the mutex locked. If it is found to be false, the thread blocks using a condition wait. As the return from *pthread_cond_wait* or *pthread_cond_timedwait* does not confirm anything about the value of the predicate, it is crucial to re-evaluate the predicate and be prepared to wait again if it is still false.

In effect, *pthread_cond_wait* and *pthread_cond_timedwait* establish a dynamic binding between a condition variable and a particular mutex. The effect of using more than one mutex for concurrent condition waits on the same condition variable is undefined.

RETURN VALUE

Upon successful completion, all functions listed above return zero. Otherwise an error code is returned. (NOTE: These calls do not set *errno* .)

ERRORS

Except for ETIMEDOUT, all the error checks indicated are performed at the beginning of processing for the function, and error returns are made before any change is made to the state of any condition variable or mutex argument.

[EINVAL]	The *cond* argument does not refer to a valid condition variable. The *mutex* argument does not refer to a valid mutex (*pthread_cond_wait* and *pthread_cond_timedwait* only). The *abstime* argument refers to an unusable time value (*pthread_cond_timedwait* only).
[EBUSY]	The condition variable *cond* cannot be deleted because one or more threads are currently blocked on it (*pthread_cond_destroy* only).
[ETIMEDOUT]	The time specified by *abstime* has passed (*pthread_cond_timedwait* only).

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

pthread_cond_init, pthread_cond_destroy, pthread_cond_signal, pthread_cond_broadcast, pthread_cond_wait, pthread_cond_timedwait – initialize and use a condition variable

SYNOPSIS

#include <pthread.h>
#include <time.h>
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int **pthread_cond_init**(pthread_cond_t * *cond*, const pthread_condattr_t * *attr*);

int **pthread_cond_destroy**(pthread_cond_t * *cond*);

int **pthread_cond_signal**(pthread_cond_t * *cond*);

int **pthread_cond_broadcast**(pthread_cond_t * *cond*);

int **pthread_cond_wait**(pthread_cond_t * *cond*, pthread_mutex_t * *mutex*);

int **pthread_cond_timedwait**(pthread_cond_t * *cond*, pthread_mutex_t * *mutex*, const struct timespec * *abstime*);

DESCRIPTION

The *pthread_cond_init* function initializes the condition variable referenced by *cond* with attributes obtained from *attr* . If *attr* is NULL, the default condition variable attributes are used.

A condition variable that is statically allocated may be initialized with the initializer macro PTHREAD_COND_INITIALIZER. The effect is equivalent to dynamic initialization using *pthread_cond_init* with *attr* equal to NULL, except that no error checks are performed.

The *pthread_cond_destroy* function deletes the condition variable designated by *cond* , if there are no threads currently blocked on it. Otherwise, *pthread_cond_destroy* returns an error.

The *pthread_cond_signal* function awakens one or more threads blocked on the condition variable *cond* . If there are multiple threads blocked on *cond* , they will be awakened in the order they blocked. The *pthread_cond_broadcast* function awakens all threads blocked on the condition variable *cond* .

Neither *pthread_cond_signal* nor *pthread_cond_broadcast* has any effect on the condition variable if there are no threads currently blocked on it.

The *pthread_cond_wait* and *pthread_cond_timedwait* functions cause a wait on the condition variable *cond* . They must be called with the mutex referenced by *mutex* locked by the calling thread. These functions atomically release *mutex* and block the calling thread. It is impossible for any other thread to acquire the mutex and invoke *pthread_cond_signal* or *pthread_cond_broadcast* so that those operations take effect between the release of the mutex and the block of the thread calling *pthread_cond_wait* or *pthread_cond_timedwait* .

Upon successful return from *pthread_cond_wait* or *pthread_cond_timedwait* , the mutex will have been re-acquired by the calling thread.

The *pthread_cond_timedwait* function is the same as *pthread_cond_wait* except that an error is returned in the following cases:

- if the absolute time specified by *abstime* expires (that is, system time as measured by the realtime clock equals or exceeds *abstime*) before the condition variable *cond* is signaled or broadcast
- if the time specified by *abstime* has already expired at the time of the call

If a timeout occurs, *pthread_cond_timedwait* will nonetheless release and reacquire the mutex *mutex* .

A condition variable wait is subject to spurious wakeups. Within the user program, a boolean predicate is usually associated with a condition variable. The predicate involves shared variables and is tested with the mutex locked. If it is found to be false, the thread blocks using a condition wait. As the return from *pthread_cond_wait* or *pthread_cond_timedwait* does not confirm anything about the value of the predicate, it is crucial to re-evaluate the predicate and be prepared to wait again if it is still false.

In effect, *pthread_cond_wait* and *pthread_cond_timedwait* establish a dynamic binding between a condition variable and a particular mutex. The effect of using more than one mutex for concurrent condition waits on the same condition variable is undefined.

RETURN VALUE

Upon successful completion, all functions listed above return zero. Otherwise an error code is returned. (NOTE: These calls do not set *errno* .)

ERRORS

Except for ETIMEDOUT, all the error checks indicated are performed at the beginning of processing for the function, and error returns are made before any change is made to the state of any condition variable or mutex argument.

[EINVAL]	The *cond* argument does not refer to a valid condition variable. The *mutex* argument does not refer to a valid mutex (*pthread_cond_wait* and *pthread_cond_timedwait* only). The *abstime* argument refers to an unusable time value (*pthread_cond_timedwait* only).
[EBUSY]	The condition variable *cond* cannot be deleted because one or more threads are currently blocked on it (*pthread_cond_destroy* only).
[ETIMEDOUT]	The time specified by *abstime* has passed (*pthread_cond_timedwait* only).

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

pthread_create – create a thread

SYNOPSIS

#include <pthread.h>
int **pthread_create**(pthread_t **thread*, const pthread_attr_t **attr*, void *(**start_routine*)(void *), void **arg*);

DESCRIPTION

The *pthread_create* function creates a new thread, with attributes specified by *attr*, within the current actor. If *attr* is NULL, the default attributes are used. If the attributes specified by *attr* are modified later, the thread's attributes are not affected. Upon successful completion, *pthread_create* stores the identifier of the created thread in the location referenced by *thread.*

On creation, the new thread begins executing the function *start_routine* with *arg* as its sole argument. If that function returns, the effect is the same as if the thread had called *pthread_exit* using the return value of *start_routine* as the exit status.

If *pthread_create* fails, no new thread is created, and the contents of the location referenced by *thread* are undefined.

RETURN VALUE

Upon successful completion, *pthread_create* returns a value of zero. Otherwise, an error code is returned. (NOTE: *errno* is not set.)

ERRORS

[EINVAL]	One or more attribute values specified by *attr* are invalid.
[EAGAIN]	A new thread could not be created because PTHREAD_THREADS_MAX would have been exceeded. Insufficient memory or system resources are available to create a new thread.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`pthread_exit`(3POSIX), `pthread_join`(3POSIX)

NAME

pthread_equal – compare thread identifiers

SYNOPSIS

#include <pthread.h>
int **pthread_equal**(pthread_t *t1*, pthread_t *t2*);

DESCRIPTION

The *pthread_equal* function compares the thread identifiers *t1* and *t2.* This function is implemented as a macro.

RETURN VALUE

The *pthread_equal*function returns a non-zero value if *t1* and *t2* are equal, otherwise zero is returned.

If either *t1* or *t2* is not a valid thread identifier, the behavior is undefined.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`pthread_create`(3POSIX), `pthread_self`(3POSIX)

NAME

pthread_exit – terminate the calling thread

SYNOPSIS

#include <pthread.h>
void **pthread_exit**(void **status*);

DESCRIPTION

The *pthread_exit* function terminates the calling thread. If the thread was created with a *detachstate* value of PTHREAD_CREATE_JOINABLE, the value *status* is made available to any successful *pthread_join* with the terminating thread. System resources associated with the exiting thread are not freed until the *pthread_join* completes. If the thread was created with *detachstate* of PTHREAD_CREATE_DETACHED, the *status* argument is ignored. In this case system resources associated with the exiting thread are freed by *pthread_exit*.

If the thread has any thread-specific data, the corresponding destructor functions are invoked in an unspecified order before the thread terminates. Thread termination does not release any application-visible actor resources, such as mutexes, condition variables, or timers.

An implicit call to *pthread_exit* is made when any thread that was created with *pthread_create* returns from its initial *start_routine* function. The return value from the thread's initial function serves as the exit status.

Any thread, including the initial thread in the actor, may use *pthread_exit* to terminate. Note that actor termination is never triggered automatically by the termination of any thread, including the initial thread, or of all threads. Program termination may be accomplished with an explicit call to `exit`.

If *pthread_exit* is called recursively, for example from a per-thread data key destructor, results are undefined.

RESTRICTIONS

The current implementation does not support thread cancellation or cancellation cleanup handlers.

RETURN VALUE

The *pthread_exit* function does not return to its caller.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`pthread_create`(3POSIX), `pthread_join`(3POSIX), `pthread_key_create`(3POSIX), `exit`(3STDC)

NAME

pthread_setschedparam, pthread_getschedparam – set or get current scheduling policy and parameters of a thread

SYNOPSIS

#include <pthread.h>
#include <sched.h>
int **pthread_setschedparam**(pthread_t *thread*, int *policy*, const struct sched_param * *param*);

int **pthread_getschedparam**(pthread_t *thread*, int * *policy*, struct sched_param * *param*);

DESCRIPTION

The *pthread_setschedparam* and *pthread_getschedparam* functions respectively modify and retrieve the dynamic scheduling policy and scheduling parameters of individual threads. The only settable parameter for the SCHED_RR and SCHED_FIFO policies (and thus the only member of *struct sched_param*) is thread priority (see *pthread_attr_setscope* (3POSIX)).

The *pthread_setschedparam* function sets the scheduling policy and scheduling parameters for the thread designated by *thread* to the policy and parameters provided in *policy* and *param* , respectively. The value of *policy* may be either SCHED_RR, SCHED_FIFO, or SCHED_OTHER which is equivalent to SCHED_RR. Thread priority is stored in the *sched_priority* member of *param* .

The *pthread_getschedparam* function retrieves the scheduling policy and scheduling parameters for the thread designated by *thread* , and stores those values in *policy* and *param* , respectively. The policy and priority returned by *pthread_getschedparam* are the values specified by the most recent *pthread_setschedparam* or *pthread_create* call affecting the target thread.

RETURN VALUE

Upon successful completion, *pthread_setschedparam* and *pthread_getschedparam* return zero. Otherwise an error code is returned. (NOTE: These calls do not set *errno* .)

ERRORS

[EINVAL]	The *thread* argument is not a valid thread identifier. Either the *policy* or the *param* argument contains an invalid value (*pthread_setschedparam* only).
[ESRCH]	No active thread corresponding to the identifier *thread* was found.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

pthread_setspecific, pthread_getspecific – set or get the thread-specific value associated with a key

SYNOPSIS

```
#include <pthread.h>
int pthread_setspecific(pthread_key_t key, const void * value);

void *pthread_getspecific(pthread_key_t key);
```

DESCRIPTION

The *pthread_setspecific* function associates 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 *pthread_getspecific* function returns the value currently bound to the specified *key* in the calling thread. This function is implemented as a macro. The effect of calling *pthread_getspecific* with a *key* value not obtained from *pthread_key_create* or after *key* has been deleted with *pthread_key_delete* is undefined.

Both *pthread_setspecific* and *pthread_getspecific* may be called, either explicitly or implicitly, from a thread-specific data destructor function. Calling *pthread_setspecific* from a destructor could result in lost storage.

RETURN VALUE

The *pthread_getspecific* function returns the thread-specific data value associated with the given *key* . If no thread-specific data value is associated with *key* in the calling thread, the value NULL is returned.

Upon successful completion, *pthread_setspecific* returns zero. Otherwise, an error code is returned. (NOTE: *errno* is not set.)

ERRORS

The following error codes apply only to *pthread_setspecific* .

[EINVAL]	The key value is invalid.
[ENOMEM]	There is nsufficient memory to associate the value with the key.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`pthread_key_create`(3POSIX)

NAME

pthread_join – wait for thread termination

SYNOPSIS

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **status);
```

DESCRIPTION

The *pthread_join* function suspends execution of the calling thread, until the target *thread* terminates (unless *thread* has already terminated). On return from a successful *pthread_join* with a non-NULL *status* argument, the value passed to *pthread_exit* by the terminating thread is made available in the location referenced by *status*. When *pthread_join* returns, the target *thread* has been terminated and its associated resources freed.

The *pthread_join* function may be used on any target thread created with the *detachstate* attribute set to PTHREAD_CREATE_JOINABLE (the default). If the *thread* argument designates a thread created in the PTHREAD_CREATE_DETACHED state, *pthread_join* returns immediately with an error. A joinable thread may only be subject to one outstanding *pthread_join*. A subsequent call to *pthread_join* while the first is still pending will return an error. However, if a thread blocked in *pthread_join* is aborted (see *threadAbort*(2K)), the target thread remains joinable by a subsequent *pthread_join*. (NOTE: thread abort is not directly supported by the CHORUS / POSIX Micro Realtime Profile (see *mrtp*(3POSIX)).

A joinable thread which has exited but remains unjoined counts against the PTHREAD_THREADS_MAX total, as system resources remain allocated on its behalf.

RETURN VALUE

Upon successful completion, *pthread_join* returns a value of zero. Otherwise, an error code is returned. (NOTE: *errno* is not set.)

ERRORS

[EINVAL]	The *thread* argument is not a valid thread identifier. The *thread* argument identifies a thread created in the PTHREAD_CREATE_DETACHED mode. Another *pthread_join* call is already pending for the thread identified by *thread*.
[ESRCH]	No thread corresponding to the identifier *thread* was found.
[EINTR]	*pthread_join* was interrupted by a thread abort.
[EDEADLK]	The *thread* argument identifies the calling thread.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

pthread_key_create, pthread_key_delete – create or delete a thread-specific data key

SYNOPSIS

```
#include <pthread.h>
int pthread_key_create(pthread_key_t * key, void (* destructor ) (void *));

int pthread_key_delete(pthread_key_t key);
```

DESCRIPTION

The *pthread_key_create* function dynamically creates a unique thread-specific data key visible to all threads in the current actor, and stores it at the location referenced by *key* . Key values returned by *pthread_key_create* are opaque indices used to locate thread-specific data. Although the same key value will 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 is associated with the new key in all active threads. Upon thread creation, the value NULL is associated with all defined keys in the new thread.

An optional destructor function may be associated with each key value. At thread exit, if any key value has a non-NULL destructor pointer, and the thread has a non-NULL value associated with that key, the function *destructor* is called with the value currently associated to it as its sole argument. After the destructor function returns, the value associated with the key in the current thread is reset to NULL. The order of destructor calls is unspecified if there is more than one destructor function for a thread when it exits.

If a destructor creates thread-specific data (by invoking *pthread_setspecific*), the destructor call can be repeated as many as PTHREAD_DESTRUCTOR_ITERATIONS times, to process all the thread-specific data. After PTHREAD_DESTRUCTOR_ITERATIONS iterations, destructor processing will terminate for the thread even if more non-NULL thread-specific data remains.

The *pthread_key_delete* function deletes a thread-specific data key. Any attempt to use *key* in any thread following the call to *pthread_key_delete* results in undefined behavior. No destructor functions are invoked by *pthread_key_delete* . The application must free storage and perform any cleanup operations needed for data structures related to the deleted key or to associated thread-specific data. The cleanup can be performed before or after invoking *pthread_key_delete* .

At actor destruction, all remaining keys associated with the actor are silently deleted. No destructor functions are invoked.

RETURN VALUE

Upon successful completion, *pthread_key_create* and *pthread_key_delete* return zero. Otherwise, an error code is returned. (NOTE: These functions do not set *errno* .)

ERRORS

[EINVAL]	The *key* value is invalid (*pthread_key_delete* only).

[EAGAIN]	The system-imposed limit on the total number of keys per actor (PTHREADS_KEYS_MAX) has been exceeded (*pthread_key_create* only).
[ENOMEM]	There is insufficient memory to create the key (*pthread_key_create* only).

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`pthread_setspecific`(3POSIX) , `pthread_exit`(3POSIX)

NAME

pthread_key_create, pthread_key_delete – create or delete a thread-specific data key

SYNOPSIS

```
#include <pthread.h>
int pthread_key_create(pthread_key_t * key, void (* destructor ) (void *));

int pthread_key_delete(pthread_key_t key);
```

DESCRIPTION

The *pthread_key_create* function dynamically creates a unique thread-specific data key visible to all threads in the current actor, and stores it at the location referenced by *key* . Key values returned by *pthread_key_create* are opaque indices used to locate thread-specific data. Although the same key value will 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 is associated with the new key in all active threads. Upon thread creation, the value NULL is associated with all defined keys in the new thread.

An optional destructor function may be associated with each key value. At thread exit, if any key value has a non-NULL destructor pointer, and the thread has a non-NULL value associated with that key, the function *destructor* is called with the value currently associated to it as its sole argument. After the destructor function returns, the value associated with the key in the current thread is reset to NULL. The order of destructor calls is unspecified if there is more than one destructor function for a thread when it exits.

If a destructor creates thread-specific data (by invoking *pthread_setspecific*), the destructor call can be repeated as many as PTHREAD_DESTRUCTOR_ITERATIONS times, to process all the thread-specific data. After PTHREAD_DESTRUCTOR_ITERATIONS iterations, destructor processing will terminate for the thread even if more non-NULL thread-specific data remains.

The *pthread_key_delete* function deletes a thread-specific data key. Any attempt to use *key* in any thread following the call to *pthread_key_delete* results in undefined behavior. No destructor functions are invoked by *pthread_key_delete* . The application must free storage and perform any cleanup operations needed for data structures related to the deleted key or to associated thread-specific data. The cleanup can be performed before or after invoking *pthread_key_delete* .

At actor destruction, all remaining keys associated with the actor are silently deleted. No destructor functions are invoked.

RETURN VALUE

Upon successful completion, *pthread_key_create* and *pthread_key_delete* return zero. Otherwise, an error code is returned. (NOTE: These functions do not set *errno* .)

ERRORS

[EINVAL]	The *key* value is invalid (*pthread_key_delete* only).

[EAGAIN]	The system-imposed limit on the total number of keys per actor (PTHREADS_KEYS_MAX) has been exceeded (*pthread_key_create* only).
[ENOMEM]	There is insufficient memory to create the key (*pthread_key_create* only).

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`pthread_setspecific`(3POSIX) , `pthread_exit`(3POSIX)

NAME

pthread_kill – send a deletion signal to a thread

SYNOPSIS

```
#include <pthread.h>
#include <signal.h>
int pthread_kill(pthread_t thread, int sig);
```

DESCRIPTION

The *pthread_kill* function sends an asynchronous signal to the thread specified.

The signal type *sig* must be equal to SIGTHREADKILL, or zero. The SIGTHREADKILL signal unconditionally deletes the target thread. It may not be caught, ignored, or blocked. No other signal types are supported. General POSIX signal features are not supported by the CHORUS / POSIX Micro Realtime Profile.

If the target thread was created with PTHREAD_CREATE_JOINABLE, it is suspended at its current point of execution. A pending or subsequent *pthread_join* for this thread will return the exit status value PTHREAD_KILLED. If the target thread is in the PTHREAD_CREATE_DETACHED state, it is deleted immediately by the *pthread_kill* call.

The *sig* parameter may be zero, in which case error checking is performed but no signal is actually sent.

RETURN VALUE

Upon successful completion, *pthread_kill* returns a value of zero. Otherwise, no signal is sent, and an error code is returned. (NOTE: *errno* is not set.)

ERRORS

[EINVAL]	The *thread* argument is not a valid thread identifier. The *sig* argument is neither zero nor SIGTHREADKILL.
[ESRCH]	No thread corresponding to the identifier *thread* was found.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`pthread_join`(3POSIX)

NAME

pthread_mutexattr_init, pthread_mutexattr_destroy – initialize or destroy a mutex attribute object

SYNOPSIS

```
#include <pthread.h>
int pthread_mutexattr_init(pthread_mutexattr_t * attr);

int pthread_mutexattr_destroy(pthread_mutexattr_t * attr);
```

DESCRIPTION

Warning: no mutex attributes are currently supported.

The *pthread_mutexattr_init* function initializes the mutex attribute object referenced by *attr* with the default values for all mutex attributes. When subsequently used by *pthread_mutex_init* , it specifies attributes for the mutex being initialized. Modification to an attribute object has no effect on mutexes already initialized from that object.

The *pthread_mutexattr_destroy* function deletes a mutex attribute object.

RETURN VALUE

The *pthread_mutexattr_init* and *pthread_mutexattr_destroy* functions always return zero.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`pthread_mutex_init`(3POSIX)

NAME

pthread_mutexattr_init, pthread_mutexattr_destroy – initialize or destroy a mutex attribute object

SYNOPSIS

```
#include <pthread.h>
int pthread_mutexattr_init(pthread_mutexattr_t * attr);

int pthread_mutexattr_destroy(pthread_mutexattr_t * attr);
```

DESCRIPTION

Warning: no mutex attributes are currently supported.

The *pthread_mutexattr_init* function initializes the mutex attribute object referenced by *attr* with the default values for all mutex attributes. When subsequently used by *pthread_mutex_init* , it specifies attributes for the mutex being initialized. Modification to an attribute object has no effect on mutexes already initialized from that object.

The *pthread_mutexattr_destroy* function deletes a mutex attribute object.

RETURN VALUE

The *pthread_mutexattr_init* and *pthread_mutexattr_destroy* functions always return zero.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`pthread_mutex_init`(3POSIX)

NAME

pthread_mutex_init, pthread_mutex_destroy, pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock – initialize and use a mutex

SYNOPSIS

```
#include <pthread.h>
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_init(pthread_mutex_t * mutex, const pthread_mutexattr_t * attr);

int pthread_mutex_destroy(pthread_mutex_t * mutex);

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 *pthread_mutex_init* function initializes the mutex referenced by *mutex* with attributes obtained from *attr* . If *attr* is NULL, the default mutex attributes are used.

A mutex that is statically allocated may be initialized with the initializer macro PTHREAD_MUTEX_INITIALIZER. The effect is the same as dynamic initialization using *pthread_mutex_init* with *attr* equal to NULL, except that no error checks are performed.

The *pthread_mutex_destroy* function deletes the mutex designated by *mutex* . Deletion of a locked mutex results in undefined behavior.

The *pthread_mutex_lock* function locks the mutex referenced by *mutex* . If the mutex is already locked by another thread, the calling thread blocks until the mutex becomes available. The *pthread_mutex_lock* function returns with the mutex in the locked state. Calls to *pthread_mutex_lock* may not be nested; an attempt by a thread to lock the same mutex a second time will result in deadlock.

The *pthread_mutex_trylock* function is the same as *pthread_mutex_lock* except that if *mutex* is already locked (by any thread, including the current thread), the call returns immediately with an error code indicating failure.

The *pthread_mutex_unlock* function releases the mutex referenced by *mutex* . If the caller is not the thread which locked the mutex most recently, or if the mutex is not locked, the behavior is undefined. If there are threads blocked on the mutex, the one waiting the longest is awakened so that it may lock the mutex and return from its *pthread_mutex_lock* call.

RETURN VALUE

Upon successful completion, all functions listed above return zero. Otherwise an error code is returned. (NOTE: These calls do not set *errno* .)

ERRORS

[EINVAL]	The *mutex* argument does not refer to a valid mutex. A pointer argument contains an address outside the current actor’s address space.

[EBUSY]	*pthread_mutex_trylock* failed to lock the mutex because it was already locked.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

pthread_mutex_init, pthread_mutex_destroy, pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock – initialize and use a mutex

SYNOPSIS

#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int **pthread_mutex_init**(pthread_mutex_t * *mutex*, const pthread_mutexattr_t * *attr*);

int **pthread_mutex_destroy**(pthread_mutex_t * *mutex*);

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 *pthread_mutex_init* function initializes the mutex referenced by *mutex* with attributes obtained from *attr* . If *attr* is NULL, the default mutex attributes are used.

A mutex that is statically allocated may be initialized with the initializer macro PTHREAD_MUTEX_INITIALIZER. The effect is the same as dynamic initialization using *pthread_mutex_init* with *attr* equal to NULL, except that no error checks are performed.

The *pthread_mutex_destroy* function deletes the mutex designated by *mutex* . Deletion of a locked mutex results in undefined behavior.

The *pthread_mutex_lock* function locks the mutex referenced by *mutex* . If the mutex is already locked by another thread, the calling thread blocks until the mutex becomes available. The *pthread_mutex_lock* function returns with the mutex in the locked state. Calls to *pthread_mutex_lock* may not be nested; an attempt by a thread to lock the same mutex a second time will result in deadlock.

The *pthread_mutex_trylock* function is the same as *pthread_mutex_lock* except that if *mutex* is already locked (by any thread, including the current thread), the call returns immediately with an error code indicating failure.

The *pthread_mutex_unlock* function releases the mutex referenced by *mutex* . If the caller is not the thread which locked the mutex most recently, or if the mutex is not locked, the behavior is undefined. If there are threads blocked on the mutex, the one waiting the longest is awakened so that it may lock the mutex and return from its *pthread_mutex_lock* call.

RETURN VALUE

Upon successful completion, all functions listed above return zero. Otherwise an error code is returned. (NOTE: These calls do not set *errno* .)

ERRORS

[EINVAL]	The *mutex* argument does not refer to a valid mutex. A pointer argument contains an address outside the current actor's address space.

[EBUSY]	*pthread_mutex_trylock* failed to lock the mutex because it was already locked.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

pthread_mutex_init, pthread_mutex_destroy, pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock – initialize and use a mutex

SYNOPSIS

```
#include <pthread.h>
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_init(pthread_mutex_t * mutex, const pthread_mutexattr_t * attr);

int pthread_mutex_destroy(pthread_mutex_t * mutex);

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 *pthread_mutex_init* function initializes the mutex referenced by *mutex* with attributes obtained from *attr* . If *attr* is NULL, the default mutex attributes are used.

A mutex that is statically allocated may be initialized with the initializer macro PTHREAD_MUTEX_INITIALIZER. The effect is the same as dynamic initialization using *pthread_mutex_init* with *attr* equal to NULL, except that no error checks are performed.

The *pthread_mutex_destroy* function deletes the mutex designated by *mutex* . Deletion of a locked mutex results in undefined behavior.

The *pthread_mutex_lock* function locks the mutex referenced by *mutex* . If the mutex is already locked by another thread, the calling thread blocks until the mutex becomes available. The *pthread_mutex_lock* function returns with the mutex in the locked state. Calls to *pthread_mutex_lock* may not be nested; an attempt by a thread to lock the same mutex a second time will result in deadlock.

The *pthread_mutex_trylock* function is the same as *pthread_mutex_lock* except that if *mutex* is already locked (by any thread, including the current thread), the call returns immediately with an error code indicating failure.

The *pthread_mutex_unlock* function releases the mutex referenced by *mutex* . If the caller is not the thread which locked the mutex most recently, or if the mutex is not locked, the behavior is undefined. If there are threads blocked on the mutex, the one waiting the longest is awakened so that it may lock the mutex and return from its *pthread_mutex_lock* call.

RETURN VALUE

Upon successful completion, all functions listed above return zero. Otherwise an error code is returned. (NOTE: These calls do not set *errno* .)

ERRORS

[EINVAL]	The *mutex* argument does not refer to a valid mutex. A pointer argument contains an address outside the current actor's address space.

[EBUSY]	*pthread_mutex_trylock* failed to lock the mutex because it was already locked.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

pthread_mutex_init, pthread_mutex_destroy, pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock – initialize and use a mutex

SYNOPSIS

#include <pthread.h>
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int **pthread_mutex_init**(pthread_mutex_t * *mutex*, const pthread_mutexattr_t * *attr*);

int **pthread_mutex_destroy**(pthread_mutex_t * *mutex*);

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 *pthread_mutex_init* function initializes the mutex referenced by *mutex* with attributes obtained from *attr* . If *attr* is NULL, the default mutex attributes are used.

A mutex that is statically allocated may be initialized with the initializer macro PTHREAD_MUTEX_INITIALIZER. The effect is the same as dynamic initialization using *pthread_mutex_init* with *attr* equal to NULL, except that no error checks are performed.

The *pthread_mutex_destroy* function deletes the mutex designated by *mutex* . Deletion of a locked mutex results in undefined behavior.

The *pthread_mutex_lock* function locks the mutex referenced by *mutex* . If the mutex is already locked by another thread, the calling thread blocks until the mutex becomes available. The *pthread_mutex_lock* function returns with the mutex in the locked state. Calls to *pthread_mutex_lock* may not be nested; an attempt by a thread to lock the same mutex a second time will result in deadlock.

The *pthread_mutex_trylock* function is the same as *pthread_mutex_lock* except that if *mutex* is already locked (by any thread, including the current thread), the call returns immediately with an error code indicating failure.

The *pthread_mutex_unlock* function releases the mutex referenced by *mutex* . If the caller is not the thread which locked the mutex most recently, or if the mutex is not locked, the behavior is undefined. If there are threads blocked on the mutex, the one waiting the longest is awakened so that it may lock the mutex and return from its *pthread_mutex_lock* call.

RETURN VALUE

Upon successful completion, all functions listed above return zero. Otherwise an error code is returned. (NOTE: These calls do not set *errno* .)

ERRORS

[EINVAL]	The *mutex* argument does not refer to a valid mutex. A pointer argument contains an address outside the current actor’s address space.

[EBUSY]	*pthread_mutex_trylock* failed to lock the mutex because it was already locked.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

pthread_mutex_init, pthread_mutex_destroy, pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock – initialize and use a mutex

SYNOPSIS

```
#include <pthread.h>
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_init(pthread_mutex_t * mutex, const pthread_mutexattr_t * attr);

int pthread_mutex_destroy(pthread_mutex_t * mutex);

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 *pthread_mutex_init* function initializes the mutex referenced by *mutex* with attributes obtained from *attr* . If *attr* is NULL, the default mutex attributes are used.

A mutex that is statically allocated may be initialized with the initializer macro PTHREAD_MUTEX_INITIALIZER. The effect is the same as dynamic initialization using *pthread_mutex_init* with *attr* equal to NULL, except that no error checks are performed.

The *pthread_mutex_destroy* function deletes the mutex designated by *mutex* . Deletion of a locked mutex results in undefined behavior.

The *pthread_mutex_lock* function locks the mutex referenced by *mutex* . If the mutex is already locked by another thread, the calling thread blocks until the mutex becomes available. The *pthread_mutex_lock* function returns with the mutex in the locked state. Calls to *pthread_mutex_lock* may not be nested; an attempt by a thread to lock the same mutex a second time will result in deadlock.

The *pthread_mutex_trylock* function is the same as *pthread_mutex_lock* except that if *mutex* is already locked (by any thread, including the current thread), the call returns immediately with an error code indicating failure.

The *pthread_mutex_unlock* function releases the mutex referenced by *mutex* . If the caller is not the thread which locked the mutex most recently, or if the mutex is not locked, the behavior is undefined. If there are threads blocked on the mutex, the one waiting the longest is awakened so that it may lock the mutex and return from its *pthread_mutex_lock* call.

RETURN VALUE

Upon successful completion, all functions listed above return zero. Otherwise an error code is returned. (NOTE: These calls do not set *errno* .)

ERRORS

[EINVAL]	The *mutex* argument does not refer to a valid mutex. A pointer argument contains an address outside the current actor’s address space.

[EBUSY]	*pthread_mutex_trylock* failed to lock the mutex because it was already locked.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

pthread_once – initialize a library dynamically

SYNOPSIS

```
#include <pthread.h>
pthread_once_t once_control = PTHREAD_ONCE_INIT;
int pthread_once(pthread_once_t *once_control, void (*init_routine)(void));
```

DESCRIPTION

The *pthread_once* function is a synchronization tool used for dynamic initialization of library packages that are invoked concurrently from multiple threads. The first call to *pthread_once* by any thread in an actor, with a *once_control* provided, will invoke the *init_routine* function with no arguments. Subsequent calls of *pthread_once* will not call *init_routine*. On return from *pthread_once* it is guaranteed that *init_routine* has completed. The variable referenced by the *once_control* argument is used to determine whether the associated initialization routine has been called.

If *once_control* has automatic storage duration or was not initialized using PTHREAD_ONCE_INIT, the behavior of *pthread_once* is undefined.

RETURN VALUE

The *pthread_once* function always returns zero.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

pthread_self – get the identifier of the calling thread

SYNOPSIS

```
#include <pthread.h>
pthread_t pthread_self(void);
```

DESCRIPTION

The *pthread_self* function returns the identifier of the calling thread.

RETURN VALUE

See DESCRIPTION above.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`pthread_create`(3POSIX), `pthread_equal`(3POSIX)

NAME

pthread_setschedparam, pthread_getschedparam – set or get current scheduling policy and parameters of a thread

SYNOPSIS

#include <pthread.h>
#include <sched.h>
int **pthread_setschedparam**(pthread_t *thread*, int *policy*, const struct sched_param * *param*);

int **pthread_getschedparam**(pthread_t *thread*, int * *policy*, struct sched_param * *param*);

DESCRIPTION

The *pthread_setschedparam* and *pthread_getschedparam* functions respectively modify and retrieve the dynamic scheduling policy and scheduling parameters of individual threads. The only settable parameter for the SCHED_RR and SCHED_FIFO policies (and thus the only member of *struct sched_param*) is thread priority (see *pthread_attr_setscope* (3POSIX)).

The *pthread_setschedparam* function sets the scheduling policy and scheduling parameters for the thread designated by *thread* to the policy and parameters provided in *policy* and *param* , respectively. The value of *policy* may be either SCHED_RR, SCHED_FIFO, or SCHED_OTHER which is equivalent to SCHED_RR. Thread priority is stored in the *sched_priority* member of *param* .

The *pthread_getschedparam* function retrieves the scheduling policy and scheduling parameters for the thread designated by *thread* , and stores those values in *policy* and *param* , respectively. The policy and priority returned by *pthread_getschedparam* are the values specified by the most recent *pthread_setschedparam* or *pthread_create* call affecting the target thread.

RETURN VALUE

Upon successful completion, *pthread_setschedparam* and *pthread_getschedparam* return zero. Otherwise an error code is returned. (NOTE: These calls do not set *errno* .)

ERRORS

[EINVAL]	The *thread* argument is not a valid thread identifier. Either the *policy* or the *param* argument contains an invalid value (*pthread_setschedparam* only).
[ESRCH]	No active thread corresponding to the identifier *thread* was found.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

pthread_setspecific, pthread_getspecific – set or get the thread-specific value associated with a key

SYNOPSIS

```
#include <pthread.h>
int pthread_setspecific(pthread_key_t key, const void * value);

void *pthread_getspecific(pthread_key_t key);
```

DESCRIPTION

The *pthread_setspecific* function associates 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 *pthread_getspecific* function returns the value currently bound to the specified *key* in the calling thread. This function is implemented as a macro. The effect of calling *pthread_getspecific* with a *key* value not obtained from *pthread_key_create* or after *key* has been deleted with *pthread_key_delete* is undefined.

Both *pthread_setspecific* and *pthread_getspecific* may be called, either explicitly or implicitly, from a thread-specific data destructor function. Calling *pthread_setspecific* from a destructor could result in lost storage.

RETURN VALUE

The *pthread_getspecific* function returns the thread-specific data value associated with the given *key* . If no thread-specific data value is associated with *key* in the calling thread, the value NULL is returned.

Upon successful completion, *pthread_setspecific* returns zero. Otherwise, an error code is returned. (NOTE: *errno* is not set.)

ERRORS

The following error codes apply only to *pthread_setspecific* .

[EINVAL]	The key value is invalid.
[ENOMEM]	There is nsufficient memory to associate the value with the key.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`pthread_key_create`(3POSIX)

NAME

pthread_yield, sched_yield – yield processor to another thread

SYNOPSIS

#include <pthread.h>
void **pthread_yield**(void);

void **sched_yield**(void);

DESCRIPTION

The *pthread_yield* function yields the processor to a runnable thread queued at the same priority as the current thread, if there is one. If there are several, the thread that has been waiting the longest will be executed. The thread that invoked *pthread_yield* remains runnable and will be re-queued at the end of the list of threads waiting to run at the given priority level.

If there are no threads waiting to run at the same priority, *pthread_yield* returns immediately.

The *sched_yield* function is identical to *pthread_yield* .

RETURN VALUE

The *pthread_yield* and *sched_yield* functions do not return a value.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

directory, opendir, readdir, telldir, seekdir, rewinddir, closedir – directory operations

SYNOPSIS

#include <sys/types.h>
#include <dirent.h>
DIR * **opendir**(const char * *filename*);

struct dirent * **readdir**(DIR * *dirp*);

long **telldir**(const DIR * *dirp*);

void **seekdir**(DIR * *dirp*, long *loc*);

void **rewinddir**(DIR * *dirp*);

int **closedir**(DIR * *dirp*);

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

The *opendir* function opens the directory named by *filename* , associates a directory stream with it and returns a pointer to be used to identify the directory stream in subsequent operations. The `NULL` pointer is returned if *filename* cannot be accessed, or if it cannot *malloc(3STDC)* enough memory to hold all of it.

The *readdir* function returns a pointer to the next directory entry. It returns `NULL` upon reaching the end of the directory or detecting an invalid *seekdir* operation.

The *telldir* function returns the current location associated with the named directory stream.

The *seekdir* function sets the position of the next *readdir* operation on the directory stream. The new position reverts to the one associated with the directory stream when the *telldir* operation was performed. Values returned by *telldir* are valid only for the lifetime of the `DIR` pointer, *dirp* , from which they are derived. If the directory is closed and then reopened, the *telldir* value may be invalidated due to undetected directory compaction. It is safe to use a previous *telldir* value immediately after a call to *opendir* and before any calls to *readdir* .

The *rewinddir* function resets the position of the named directory stream to the beginning of the directory.

The *closedir* function closes the named directory stream and frees the structure associated with the *dirp* pointer, returning 0 on success. On failure, –1 is returned and the global variable *errno* is set to indicate the error.

Sample code which searches a directory for the “name” entry is:

```
len = strlen(name);
dirp = opendir(".");
if (dirp) {
   while ((dp = readdir(dirp)) != NULL)
 if (dp->d_namlen == len && !strcmp(dp->d_name, name)) {
  (void) closedir(dirp);
  return FOUND;
         }
   (void) closedir(dirp);
   }
return NOT_FOUND;
```

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`open`(2POSIX) , `close`(2POSIX) , `read`(2POSIX) , `lseek`(2POSIX)

HISTORY

The *opendir, readdir, telldir, seekdir, rewinddir* and *closedir* functions appeared in 4.2 BSD.

RESTRICTIONS

These library calls do not support multi-threaded applications.

NAME

recno – record number database access method

SYNOPSIS

```
#include <sys/types.h>
#include <db.h>
```

FEATURES

UFS

DESCRIPTION

The *dbopen* routine is the library interface to database files. One of the file formats supported is record number files. The general description of the database access methods is in *dbopen(3POSIX),* this manual page describes only the recno—specific information.

The record number data structure is either variable or fixed-length records stored in a flat-file format, accessed by the logical record number. The existence of a record number five implies the existence of records one to four, and the deletion of record number one causes record number five to be renumbered to record number four, as well as the cursor, if positioned after record number one, to shift down one record.

The recno access method—specific data structure provided to *dbopen* is defined in the <db.h> include file as follows:

```
typedef struct {
    u_long flags;
    u_int cachesize;
    u_int  psize;
    int    lorder;
    size_t  reclen;
    u_char  bval;
    char* bfname;
} RECNOINFO;
```

The elements of this structure are defined as follows:

flags The flag value is specified by *or*'ing any of the following values:

R_FIXEDLEN The records are fixed-length, not byte delimited. The structure element *reclen* specifies the length of the record, and the structure element *bval* is used as the pad character.

R_NOKEY In the interface specified by *dbopen*, the sequential record retrieval fills in both the caller's key and data structures. If the R_NOKEY flag is specified, the *cursor* routines are not required to fill in the key structure. This permits applications to

		retrieve records at the end of files without reading all of the intervening records.
	R_SNAPSHOT	This flag requires that a snapshot of the file be taken when *dbopen* is called, instead of permitting any unmodified records to be read from the original file.

cachesize	A suggested maximum size, in bytes, of the memory cache. This value is `only` advisory, and the access method will allocate more memory rather than fail. If *cachesize* is 0 (no size is specified) a default cache is used.
psize	The recno access method stores the in-memory copies of its records in a btree. This value is the size (in bytes) of the pages used for nodes in that tree. If *psize* is 0 (no page size is specified) a page size is chosen based on the underlying file system I/O block size. See *btree(3POSIX)* for more information.
lorder	The byte order for integers in the stored database metadata. The number should represent the order as an integer; for example, big endian order would be the number 4,321. If `lorder` is 0 (no order is specified) the current host order is used.
reclen	The length of a fixed-length record.
bval	The delimiting byte to be used to mark the end of a record for variable-length records, and the pad character for fixed-length records. If no value is specified, newlines (“\n”) are used to mark the end of variable-length records and fixed-length records are padded with spaces.
bfname	The recno access method stores the in-memory copies of its records in a btree. If bfname is non-NULL, it specifies the name of the btree file, as if specified as the file name for a dbopen of a btree file.

The data part of the key/data pair used by the recno access method is the same as other access methods. The key is different. The *data* field of the key should be a pointer to a memory location of type *recno_t*, as defined in the <db.h> include file. This type is normally the largest unsigned integral type available to the implementation. The `size` field of the key should be the size of that type.

In the interface specified by *dbopen*, using the *put* interface to create a new record will cause the creation of multiple, empty records if the record number is more than one greater than the largest record currently in the database.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`dbopen`(3POSIX), `hash`(3POSIX), `mpool`(3POSIX), `recno`(3POSIX)

Document Processing in a Relational Database System, Michael Stonebraker, Heidi Stettner, Joseph Kalash, Antonin Guttman, Nadene Lynn, Memorandum No. UCB/ERL M82/32, May 1982.

BUGS

Only big and little endian byte order is supported.

RESTRICTIONS

These library calls does not support multithreaded applications.

NAME

directory, opendir, readdir, telldir, seekdir, rewinddir, closedir – directory operations

SYNOPSIS

#include <sys/types.h>
#include <dirent.h>
DIR * **opendir**(const char * *filename*);

struct dirent * **readdir**(DIR * *dirp*);

long **telldir**(const DIR * *dirp*);

void **seekdir**(DIR * *dirp*, long *loc*);

void **rewinddir**(DIR * *dirp*);

int **closedir**(DIR * *dirp*);

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

The *opendir* function opens the directory named by *filename* , associates a directory stream with it and returns a pointer to be used to identify the directory stream in subsequent operations. The `NULL` pointer is returned if *filename* cannot be accessed, or if it cannot *malloc(3STDC)* enough memory to hold all of it.

The *readdir* function returns a pointer to the next directory entry. It returns `NULL` upon reaching the end of the directory or detecting an invalid *seekdir* operation.

The *telldir* function returns the current location associated with the named directory stream.

The *seekdir* function sets the position of the next *readdir* operation on the directory stream. The new position reverts to the one associated with the directory stream when the *telldir* operation was performed. Values returned by *telldir* are valid only for the lifetime of the `DIR` pointer, *dirp* , from which they are derived. If the directory is closed and then reopened, the *telldir* value may be invalidated due to undetected directory compaction. It is safe to use a previous *telldir* value immediately after a call to *opendir* and before any calls to *readdir* .

The *rewinddir* function resets the position of the named directory stream to the beginning of the directory.

The *closedir* function closes the named directory stream and frees the structure associated with the *dirp* pointer, returning 0 on success. On failure, –1 is returned and the global variable *errno* is set to indicate the error.

Sample code which searches a directory for the “name” entry is:

```
len = strlen(name);
dirp = opendir(".");
if (dirp) {
   while ((dp = readdir(dirp)) != NULL)
 if (dp->d_namlen == len && !strcmp(dp->d_name, name)) {
  (void) closedir(dirp);
  return FOUND;
         }
   (void) closedir(dirp);
   }
return NOT_FOUND;
```

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`open`(2POSIX) , `close`(2POSIX) , `read`(2POSIX) , `lseek`(2POSIX)

HISTORY

The *opendir, readdir, telldir, seekdir, rewinddir* and *closedir* functions appeared in 4.2 BSD.

RESTRICTIONS

These library calls do not support multi-threaded applications.

NAME

sched_get_priority_max, sched_get_priority_min, sched_rr_get_interval – get priority and time quantum information for scheduling policy

SYNOPSIS

#include <sched.h>
#include <time.h>
int **sched_get_priority_max**(int *policy*);

int **sched_get_priority_min**(int *policy*);

int **sched_rr_get_interval**(int *id*, struct timespec * *interval*);

DESCRIPTION

The *sched_get_priority_max* and *sched_get_priority_min* functions respectively return the maximum and minimum priority values in the range defined for the scheduling policy specified by *policy* . One of the policies defined in `<sched.h>` must be specified.

The *sched_rr_get_interval* function stores, in the location referenced by *interval* , the time quantum currently in effect under the SCHED_RR policy. The time quantum is defined as the execution time limit after which a rescheduling decision may be made if another thread at the same priority is ready to execute. The `id` field must be set to zero.

RETURN VALUE

Upon successful completion, *sched_get_priority_max* and *sched_get_priority_min* return the appropriate maximum or minimum values, respectively. Otherwise, they return a value of –1 and set *errno* to indicate the error condition.

Upon successful completion, *sched_rr_get_interval* returns zero. Otherwise, it returns a value of –1 and sets *errno* to indicate the error condition.

ERRORS

[EINVAL]	The *policy* argument does not identify a defined scheduling policy (*sched_get_priority_max* and *sched_get_priority_min* only).
[ESRCH]	The `id` argument is not zero (*sched_rr_get_interval* only).

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

sched_get_priority_max, sched_get_priority_min, sched_rr_get_interval – get priority and time quantum information for scheduling policy

SYNOPSIS

#include <sched.h>
#include <time.h>
int **sched_get_priority_max**(int *policy*);

int **sched_get_priority_min**(int *policy*);

int **sched_rr_get_interval**(int *id*, struct timespec * *interval*);

DESCRIPTION

The *sched_get_priority_max* and *sched_get_priority_min* functions respectively return the maximum and minimum priority values in the range defined for the scheduling policy specified by *policy* . One of the policies defined in `<sched.h>` must be specified.

The *sched_rr_get_interval* function stores, in the location referenced by *interval* , the time quantum currently in effect under the SCHED_RR policy. The time quantum is defined as the execution time limit after which a rescheduling decision may be made if another thread at the same priority is ready to execute. The `id` field must be set to zero.

RETURN VALUE

Upon successful completion, *sched_get_priority_max* and *sched_get_priority_min* return the appropriate maximum or minimum values, respectively. Otherwise, they return a value of –1 and set *errno* to indicate the error condition.

Upon successful completion, *sched_rr_get_interval* returns zero. Otherwise, it returns a value of –1 and sets *errno* to indicate the error condition.

ERRORS

[EINVAL]	The *policy* argument does not identify a defined scheduling policy (*sched_get_priority_max* and *sched_get_priority_min* only).
[ESRCH]	The `id` argument is not zero (*sched_rr_get_interval* only).

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

sched_get_priority_max, sched_get_priority_min, sched_rr_get_interval – get priority and time quantum information for scheduling policy

SYNOPSIS

```
#include <sched.h>
#include <time.h>
int sched_get_priority_max(int policy);

int sched_get_priority_min(int policy);

int sched_rr_get_interval(int id, struct timespec * interval);
```

DESCRIPTION

The *sched_get_priority_max* and *sched_get_priority_min* functions respectively return the maximum and minimum priority values in the range defined for the scheduling policy specified by *policy* . One of the policies defined in `<sched.h>` must be specified.

The *sched_rr_get_interval* function stores, in the location referenced by *interval* , the time quantum currently in effect under the SCHED_RR policy. The time quantum is defined as the execution time limit after which a rescheduling decision may be made if another thread at the same priority is ready to execute. The `id` field must be set to zero.

RETURN VALUE

Upon successful completion, *sched_get_priority_max* and *sched_get_priority_min* return the appropriate maximum or minimum values, respectively. Otherwise, they return a value of –1 and set *errno* to indicate the error condition.

Upon successful completion, *sched_rr_get_interval* returns zero. Otherwise, it returns a value of –1 and sets *errno* to indicate the error condition.

ERRORS

[EINVAL]	The *policy* argument does not identify a defined scheduling policy (*sched_get_priority_max* and *sched_get_priority_min* only).
[ESRCH]	The `id` argument is not zero (*sched_rr_get_interval* only).

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

pthread_yield, sched_yield – yield processor to another thread

SYNOPSIS

#include <pthread.h>
void **pthread_yield**(void);

void **sched_yield**(void);

DESCRIPTION

The *pthread_yield* function yields the processor to a runnable thread queued at the same priority as the current thread, if there is one. If there are several, the thread that has been waiting the longest will be executed. The thread that invoked *pthread_yield* remains runnable and will be re-queued at the end of the list of threads waiting to run at the given priority level.

If there are no threads waiting to run at the same priority, *pthread_yield* returns immediately.

The *sched_yield* function is identical to *pthread_yield* .

RETURN VALUE

The *pthread_yield* and *sched_yield* functions do not return a value.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

directory, opendir, readdir, telldir, seekdir, rewinddir, closedir – directory operations

SYNOPSIS

#include <sys/types.h>
#include <dirent.h>
DIR * **opendir**(const char * *filename*);

struct dirent * **readdir**(DIR * *dirp*);

long **telldir**(const DIR * *dirp*);

void **seekdir**(DIR * *dirp*, long *loc*);

void **rewinddir**(DIR * *dirp*);

int **closedir**(DIR * *dirp*);

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

The *opendir* function opens the directory named by *filename* , associates a directory stream with it and returns a pointer to be used to identify the directory stream in subsequent operations. The `NULL` pointer is returned if *filename* cannot be accessed, or if it cannot *malloc(3STDC)* enough memory to hold all of it.

The *readdir* function returns a pointer to the next directory entry. It returns `NULL` upon reaching the end of the directory or detecting an invalid *seekdir* operation.

The *telldir* function returns the current location associated with the named directory stream.

The *seekdir* function sets the position of the next *readdir* operation on the directory stream. The new position reverts to the one associated with the directory stream when the *telldir* operation was performed. Values returned by *telldir* are valid only for the lifetime of the `DIR` pointer, *dirp* , from which they are derived. If the directory is closed and then reopened, the *telldir* value may be invalidated due to undetected directory compaction. It is safe to use a previous *telldir* value immediately after a call to *opendir* and before any calls to *readdir* .

The *rewinddir* function resets the position of the named directory stream to the beginning of the directory.

The *closedir* function closes the named directory stream and frees the structure associated with the *dirp* pointer, returning 0 on success. On failure, –1 is returned and the global variable *errno* is set to indicate the error.

Sample code which searches a directory for the “name” entry is:

```
len = strlen(name);
dirp = opendir(".");
if (dirp) {
   while ((dp = readdir(dirp)) != NULL)
 if (dp->d_namlen == len && !strcmp(dp->d_name, name)) {
  (void) closedir(dirp);
  return FOUND;
         }
   (void) closedir(dirp);
   }
return NOT_FOUND;
```

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`open`(2POSIX) , `close`(2POSIX) , `read`(2POSIX) , `lseek`(2POSIX)

HISTORY

The *opendir, readdir, telldir, seekdir, rewinddir* and *closedir* functions appeared in 4.2 BSD.

RESTRICTIONS

These library calls do not support multi-threaded applications.

NAME

sem_init, sem_destroy, sem_wait, sem_trywait, sem_post, sem_getvalue – initialize and use a semaphore

SYNOPSIS

```
#include <semaphore.h>
int sem_init(sem_t * sem, int pshared, unsigned int value);

int sem_destroy(sem_t * sem);

int sem_wait(sem_t * sem);

int sem_trywait(sem_t * sem);

int sem_post(sem_t * sem);

int sem_getvalue(sem_t * sem, int * sval);
```

DESCRIPTION

The *sem_init* function initializes the counting semaphore referenced by *sem* , setting the initial counter value to *value* . The *pshared* argument must be zero, as inter-process sharing is not defined in the CHORUS / POSIX Micro Realtime Profile.

The *sem_destroy* function deletes the semaphore referenced by *sem* .

The *sem_wait* function “locks” the semaphore by decrementing the counter value. If the result is negative, the caller is blocked until either the counter is incremented by a subsequent *sem_post* , or the calling thread is aborted (see *threadAbort* (2K)). (NOTE: thread abort is not directly supported by the CHORUS / POSIX Micro Realtime Profile (see *mrtp* (3POSIX)).) The *sem_trywait* function “locks” the semaphore only if it can do so without blocking. It atomically examines the counter value, decrements it only if it is currently positive, and returns immediately. If *sem_trywait* fails, it returns an error code of EAGAIN; in this case it has no effect on the counter or on any threads blocked on the semaphore.

The *sem_post* function “unlocks” the semaphore by incrementing the counter value. If the result is negative or zero, a thread that is blocked behind the semaphore is awakened and allowed to return successfully from its call to *sem_wait* . The thread that has been blocked for the longest time will be selected to be awakened.

The *sem_getvalue* function stores in the location referenced by *sval* the instantaneous counter value of the semaphore *sem* . It has no effect on the state of the semaphore. In some cases, recent counter increments may not be reflected, and hence the stored value may be lower than the actual semaphore counter (it will never be higher). Furthermore, concurrent operations may cause the actual semaphore counter to have changed by the time the caller obtains the stored value.

If the counter value stored by *sem_getvalue* indicates that the semaphore is unavailable (zero or negative), the absolute value of the counter is the number of threads blocked behind the semaphore.

RETURN VALUE

Upon successful completion, *sem_init* , *sem_wait* , *sem_trywait* , *sem_post* , and *sem_getvalue* return zero. In case of error a value of –1 is returned and *errno* is set to indicate the error condition.

The *sem_destroy* function always returns zero.

ERRORS

[EINVAL]	The *pshared* argument is non-zero, or the *value* argument is greater than SEM_VALUE_MAX (*sem_init* only). A pointer argument contains an address outside the current actor's address space. The semaphore addressed by *sem* was not initialized or has been corrupted (*sem_wait* , *sem_trywait* , *sem_post*).
[EINTR]	*sem_wait* or *sem_trywait* was interrupted by an abort.
[EAGAIN]	*sem_trywait* was unable to lock the semaphore.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

sem_init, sem_destroy, sem_wait, sem_trywait, sem_post, sem_getvalue – initialize and use a semaphore

SYNOPSIS

#include <semaphore.h>
int **sem_init**(sem_t * *sem*, int *pshared*, unsigned int *value*);

int **sem_destroy**(sem_t * *sem*);

int **sem_wait**(sem_t * *sem*);

int **sem_trywait**(sem_t * *sem*);

int **sem_post**(sem_t * *sem*);

int **sem_getvalue**(sem_t * *sem*, int * *sval*);

DESCRIPTION

The *sem_init* function initializes the counting semaphore referenced by *sem* , setting the initial counter value to *value* . The *pshared* argument must be zero, as inter-process sharing is not defined in the CHORUS / POSIX Micro Realtime Profile.

The *sem_destroy* function deletes the semaphore referenced by *sem* .

The *sem_wait* function “locks” the semaphore by decrementing the counter value. If the result is negative, the caller is blocked until either the counter is incremented by a subsequent *sem_post* , or the calling thread is aborted (see *threadAbort* (2K)). (NOTE: thread abort is not directly supported by the CHORUS / POSIX Micro Realtime Profile (see *mrtp* (3POSIX)).) The *sem_trywait* function “locks” the semaphore only if it can do so without blocking. It atomically examines the counter value, decrements it only if it is currently positive, and returns immediately. If *sem_trywait* fails, it returns an error code of EAGAIN; in this case it has no effect on the counter or on any threads blocked on the semaphore.

The *sem_post* function “unlocks” the semaphore by incrementing the counter value. If the result is negative or zero, a thread that is blocked behind the semaphore is awakened and allowed to return successfully from its call to *sem_wait* . The thread that has been blocked for the longest time will be selected to be awakened.

The *sem_getvalue* function stores in the location referenced by *sval* the instantaneous counter value of the semaphore *sem* . It has no effect on the state of the semaphore. In some cases, recent counter increments may not be reflected, and hence the stored value may be lower than the actual semaphore counter (it will never be higher). Furthermore, concurrent operations may cause the actual semaphore counter to have changed by the time the caller obtains the stored value.

If the counter value stored by *sem_getvalue* indicates that the semaphore is unavailable (zero or negative), the absolute value of the counter is the number of threads blocked behind the semaphore.

RETURN VALUE

Upon successful completion, *sem_init* , *sem_wait* , *sem_trywait* , *sem_post* , and *sem_getvalue* return zero. In case of error a value of –1 is returned and *errno* is set to indicate the error condition.

The *sem_destroy* function always returns zero.

ERRORS

[EINVAL]	The *pshared* argument is non-zero, or the *value* argument is greater than SEM_VALUE_MAX (*sem_init* only). A pointer argument contains an address outside the current actor's address space. The semaphore addressed by *sem* was not initialized or has been corrupted (*sem_wait* , *sem_trywait* , *sem_post*).
[EINTR]	*sem_wait* or *sem_trywait* was interrupted by an abort.
[EAGAIN]	*sem_trywait* was unable to lock the semaphore.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

sem_init, sem_destroy, sem_wait, sem_trywait, sem_post, sem_getvalue – initialize and use a semaphore

SYNOPSIS

#include <semaphore.h>

int **sem_init**(sem_t * *sem*, int *pshared*, unsigned int *value*);

int **sem_destroy**(sem_t * *sem*);

int **sem_wait**(sem_t * *sem*);

int **sem_trywait**(sem_t * *sem*);

int **sem_post**(sem_t * *sem*);

int **sem_getvalue**(sem_t * *sem*, int * *sval*);

DESCRIPTION

The *sem_init* function initializes the counting semaphore referenced by *sem* , setting the initial counter value to *value* . The *pshared* argument must be zero, as inter-process sharing is not defined in the CHORUS / POSIX Micro Realtime Profile.

The *sem_destroy* function deletes the semaphore referenced by *sem* .

The *sem_wait* function “locks” the semaphore by decrementing the counter value. If the result is negative, the caller is blocked until either the counter is incremented by a subsequent *sem_post* , or the calling thread is aborted (see *threadAbort* (2K)). (NOTE: thread abort is not directly supported by the CHORUS / POSIX Micro Realtime Profile (see *mrtp* (3POSIX)).) The *sem_trywait* function “locks” the semaphore only if it can do so without blocking. It atomically examines the counter value, decrements it only if it is currently positive, and returns immediately. If *sem_trywait* fails, it returns an error code of EAGAIN; in this case it has no effect on the counter or on any threads blocked on the semaphore.

The *sem_post* function “unlocks” the semaphore by incrementing the counter value. If the result is negative or zero, a thread that is blocked behind the semaphore is awakened and allowed to return successfully from its call to *sem_wait* . The thread that has been blocked for the longest time will be selected to be awakened.

The *sem_getvalue* function stores in the location referenced by *sval* the instantaneous counter value of the semaphore *sem* . It has no effect on the state of the semaphore. In some cases, recent counter increments may not be reflected, and hence the stored value may be lower than the actual semaphore counter (it will never be higher). Furthermore, concurrent operations may cause the actual semaphore counter to have changed by the time the caller obtains the stored value.

If the counter value stored by *sem_getvalue* indicates that the semaphore is unavailable (zero or negative), the absolute value of the counter is the number of threads blocked behind the semaphore.

RETURN VALUE

Upon successful completion, *sem_init* , *sem_wait* , *sem_trywait* , *sem_post* , and *sem_getvalue* return zero. In case of error a value of –1 is returned and *errno* is set to indicate the error condition.

The *sem_destroy* function always returns zero.

ERRORS

[EINVAL]	The *pshared* argument is non-zero, or the *value* argument is greater than SEM_VALUE_MAX (*sem_init* only). A pointer argument contains an address outside the current actor's address space. The semaphore addressed by *sem* was not initialized or has been corrupted (*sem_wait* , *sem_trywait* , *sem_post*).
[EINTR]	*sem_wait* or *sem_trywait* was interrupted by an abort.
[EAGAIN]	*sem_trywait* was unable to lock the semaphore.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

sem_init, sem_destroy, sem_wait, sem_trywait, sem_post, sem_getvalue – initialize and use a semaphore

SYNOPSIS

```
#include <semaphore.h>
int sem_init(sem_t * sem, int pshared, unsigned int value);

int sem_destroy(sem_t * sem);

int sem_wait(sem_t * sem);

int sem_trywait(sem_t * sem);

int sem_post(sem_t * sem);

int sem_getvalue(sem_t * sem, int * sval);
```

DESCRIPTION

The *sem_init* function initializes the counting semaphore referenced by *sem* , setting the initial counter value to *value* . The *pshared* argument must be zero, as inter-process sharing is not defined in the CHORUS / POSIX Micro Realtime Profile.

The *sem_destroy* function deletes the semaphore referenced by *sem* .

The *sem_wait* function “locks” the semaphore by decrementing the counter value. If the result is negative, the caller is blocked until either the counter is incremented by a subsequent *sem_post* , or the calling thread is aborted (see *threadAbort* (2K)). (NOTE: thread abort is not directly supported by the CHORUS / POSIX Micro Realtime Profile (see *mrtp* (3POSIX)).) The *sem_trywait* function “locks” the semaphore only if it can do so without blocking. It atomically examines the counter value, decrements it only if it is currently positive, and returns immediately. If *sem_trywait* fails, it returns an error code of EAGAIN; in this case it has no effect on the counter or on any threads blocked on the semaphore.

The *sem_post* function “unlocks” the semaphore by incrementing the counter value. If the result is negative or zero, a thread that is blocked behind the semaphore is awakened and allowed to return successfully from its call to *sem_wait* . The thread that has been blocked for the longest time will be selected to be awakened.

The *sem_getvalue* function stores in the location referenced by *sval* the instantaneous counter value of the semaphore *sem* . It has no effect on the state of the semaphore. In some cases, recent counter increments may not be reflected, and hence the stored value may be lower than the actual semaphore counter (it will never be higher). Furthermore, concurrent operations may cause the actual semaphore counter to have changed by the time the caller obtains the stored value.

If the counter value stored by *sem_getvalue* indicates that the semaphore is unavailable (zero or negative), the absolute value of the counter is the number of threads blocked behind the semaphore.

RETURN VALUE

Upon successful completion, *sem_init* , *sem_wait* , *sem_trywait* , *sem_post* , and *sem_getvalue* return zero. In case of error a value of –1 is returned and *errno* is set to indicate the error condition.

The *sem_destroy* function always returns zero.

ERRORS

[EINVAL]	The *pshared* argument is non-zero, or the *value* argument is greater than SEM_VALUE_MAX (*sem_init* only). A pointer argument contains an address outside the current actor's address space. The semaphore addressed by *sem* was not initialized or has been corrupted (*sem_wait* , *sem_trywait* , *sem_post*).
[EINTR]	*sem_wait* or *sem_trywait* was interrupted by an abort.
[EAGAIN]	*sem_trywait* was unable to lock the semaphore.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

sem_init, sem_destroy, sem_wait, sem_trywait, sem_post, sem_getvalue – initialize and use a semaphore

SYNOPSIS

#include <semaphore.h>

int **sem_init**(sem_t * *sem*, int *pshared*, unsigned int *value*);

int **sem_destroy**(sem_t * *sem*);

int **sem_wait**(sem_t * *sem*);

int **sem_trywait**(sem_t * *sem*);

int **sem_post**(sem_t * *sem*);

int **sem_getvalue**(sem_t * *sem*, int * *sval*);

DESCRIPTION

The *sem_init* function initializes the counting semaphore referenced by *sem* , setting the initial counter value to *value* . The *pshared* argument must be zero, as inter-process sharing is not defined in the CHORUS / POSIX Micro Realtime Profile.

The *sem_destroy* function deletes the semaphore referenced by *sem* .

The *sem_wait* function “locks” the semaphore by decrementing the counter value. If the result is negative, the caller is blocked until either the counter is incremented by a subsequent *sem_post* , or the calling thread is aborted (see *threadAbort* (2K)). (NOTE: thread abort is not directly supported by the CHORUS / POSIX Micro Realtime Profile (see *mrtp* (3POSIX)).) The *sem_trywait* function “locks” the semaphore only if it can do so without blocking. It atomically examines the counter value, decrements it only if it is currently positive, and returns immediately. If *sem_trywait* fails, it returns an error code of EAGAIN; in this case it has no effect on the counter or on any threads blocked on the semaphore.

The *sem_post* function “unlocks” the semaphore by incrementing the counter value. If the result is negative or zero, a thread that is blocked behind the semaphore is awakened and allowed to return successfully from its call to *sem_wait* . The thread that has been blocked for the longest time will be selected to be awakened.

The *sem_getvalue* function stores in the location referenced by *sval* the instantaneous counter value of the semaphore *sem* . It has no effect on the state of the semaphore. In some cases, recent counter increments may not be reflected, and hence the stored value may be lower than the actual semaphore counter (it will never be higher). Furthermore, concurrent operations may cause the actual semaphore counter to have changed by the time the caller obtains the stored value.

If the counter value stored by *sem_getvalue* indicates that the semaphore is unavailable (zero or negative), the absolute value of the counter is the number of threads blocked behind the semaphore.

RETURN VALUE

Upon successful completion, *sem_init* , *sem_wait* , *sem_trywait* , *sem_post* , and *sem_getvalue* return zero. In case of error a value of –1 is returned and *errno* is set to indicate the error condition.

The *sem_destroy* function always returns zero.

ERRORS

[EINVAL]	The *pshared* argument is non-zero, or the *value* argument is greater than SEM_VALUE_MAX (*sem_init* only). A pointer argument contains an address outside the current actor's address space. The semaphore addressed by *sem* was not initialized or has been corrupted (*sem_wait* , *sem_trywait* , *sem_post*).
[EINTR]	*sem_wait* or *sem_trywait* was interrupted by an abort.
[EAGAIN]	*sem_trywait* was unable to lock the semaphore.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

sem_init, sem_destroy, sem_wait, sem_trywait, sem_post, sem_getvalue – initialize and use a semaphore

SYNOPSIS

#include <semaphore.h>

int **sem_init**(sem_t * *sem*, int *pshared*, unsigned int *value*);

int **sem_destroy**(sem_t * *sem*);

int **sem_wait**(sem_t * *sem*);

int **sem_trywait**(sem_t * *sem*);

int **sem_post**(sem_t * *sem*);

int **sem_getvalue**(sem_t * *sem*, int * *sval*);

DESCRIPTION

The *sem_init* function initializes the counting semaphore referenced by *sem* , setting the initial counter value to *value* . The *pshared* argument must be zero, as inter-process sharing is not defined in the CHORUS / POSIX Micro Realtime Profile.

The *sem_destroy* function deletes the semaphore referenced by *sem* .

The *sem_wait* function “locks” the semaphore by decrementing the counter value. If the result is negative, the caller is blocked until either the counter is incremented by a subsequent *sem_post* , or the calling thread is aborted (see *threadAbort* (2K)). (NOTE: thread abort is not directly supported by the CHORUS / POSIX Micro Realtime Profile (see *mrtp* (3POSIX)).) The *sem_trywait* function “locks” the semaphore only if it can do so without blocking. It atomically examines the counter value, decrements it only if it is currently positive, and returns immediately. If *sem_trywait* fails, it returns an error code of EAGAIN; in this case it has no effect on the counter or on any threads blocked on the semaphore.

The *sem_post* function “unlocks” the semaphore by incrementing the counter value. If the result is negative or zero, a thread that is blocked behind the semaphore is awakened and allowed to return successfully from its call to *sem_wait* . The thread that has been blocked for the longest time will be selected to be awakened.

The *sem_getvalue* function stores in the location referenced by *sval* the instantaneous counter value of the semaphore *sem* . It has no effect on the state of the semaphore. In some cases, recent counter increments may not be reflected, and hence the stored value may be lower than the actual semaphore counter (it will never be higher). Furthermore, concurrent operations may cause the actual semaphore counter to have changed by the time the caller obtains the stored value.

If the counter value stored by *sem_getvalue* indicates that the semaphore is unavailable (zero or negative), the absolute value of the counter is the number of threads blocked behind the semaphore.

RETURN VALUE

Upon successful completion, *sem_init* , *sem_wait* , *sem_trywait* , *sem_post* , and *sem_getvalue* return zero. In case of error a value of –1 is returned and *errno* is set to indicate the error condition.

The *sem_destroy* function always returns zero.

ERRORS

[EINVAL]	The *pshared* argument is non-zero, or the *value* argument is greater than SEM_VALUE_MAX (*sem_init* only). A pointer argument contains an address outside the current actor's address space. The semaphore addressed by *sem* was not initialized or has been corrupted (*sem_wait* , *sem_trywait* , *sem_post*).
[EINTR]	*sem_wait* or *sem_trywait* was interrupted by an abort.
[EAGAIN]	*sem_trywait* was unable to lock the semaphore.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

getnetent, getnetbyaddr, getnetbyname, setnetent, endnetent – get network entry

SYNOPSIS

#include <netdb.h>
struct netent * **getnetent**(void);

struct netent * **getnetbyname**(char * *name*);

struct netent * **getnetbyaddr**(long *net*, int *type*);

void **setnetent**(int *stayopen*);

void **endnetent**(void);

DESCRIPTION

The *getnetent* , *getnetbyname* , and *getnetbyaddr* functions each return a pointer to an object containing the broken-out fields of a line in the network data base */etc/networks* . The object has the following structure:

```
struct netent {
   char*    n_name;        /* official name of net */
   char**   n_aliases;     /* alias list */
   int      n_addrtype;    /* net number type */
   unsigned long   n_net;  /* net number */
};
```

The members of this structure are:

`n_name`	The official name of the network.
`n_aliases`	A zero terminated list of alternate names for the network.
`n_addrtype`	The type of the network number returned; currently only AF_INET.
`n_net`	The network number. Network numbers are returned in machine byte order.

The *getnetent* function reads the next line of the file, opening the file if necessary.

The *setnetent* function opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to *getnetbyname* or *getnetbyaddr* .

The *endnetent* function closes the file.

The *getnetbyname* and *getnetbyaddr* functions sequentially search from the beginning of the file until a matching net name or net address and type is found, or until EOF is encountered. Network numbers are supplied in host order.

FILES

/etc/networks

DIAGNOSTICS

A null pointer (0) is returned when EOF or an error is encountered.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`networks`(4CC)

BUGS

The data space used by these functions is static; if the data will be required in the future, it should be copied before any subsequent calls to these functions overwrite it. Only Internet network numbers are currently understood.

NAME

getnetgrent, innetgr, setnetgrent, endnetgrent – netgroup database operations

SYNOPSIS

int **getnetgrent**(char ** *host*, char ** *user*, char ** *domain*);

int **innetgr**(const char * *netgroup*, const char * *host*, const char * *user*, const char * *domain*);

void **setnetgrent**(const char * *netgroup*);

void **endnetgrent**(void);

DESCRIPTION

These functions operate on the netgroup database file `/etc/netgroup` which is described in *netgroup(4CC)* . The database defines a set of netgroups, each made up of one or more triples:

(host, user, domain)

which define a combination of host, user and domain. Any of the three fields may be specified as "wildcards" which match any string.

The *getnetgrent* function sets the three pointer arguments to the strings of the next member of the current netgroup. If any of the string pointers are (`char` *)0 that field is considered a wildcard.

The *setnetgrent* and *endnetgrent* functions set the current netgroup and terminate the current netgroup, respectively. If *setnetgrent* is called with a different netgroup from the previous call, an *endnetgrent* is implied. The *setnetgrent* function also sets the offset to the first member of the netgroup.

The *innetgr* function searches for a match of all fields within the specified group. If any of the *host* or *domain* arguments are (`char` *)0 those fields will match any string value in the netgroup member.

RETURN VALUES

The *getnetgrent* function returns 0 for "no more netgroup members" and 1 otherwise. The *innetgr* function returns 1 for a successful match and 0 otherwise. The *setnetgrent* and *endnetgrent* functions have no return value.

FILES

`/etc/netgroup` netgroup database file

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`netgroup`(4CC)

COMPATIBILITY

The netgroup members have three string fields to maintain compatibility with other vendor implementations. However, the applicability of the *domain* string within BSD is unclear.

BUGS

The *getnetgrent* function returns pointers to dynamically allocated data areas that are freed when the *endnetgrent* function is called.

RESTRICTIONS

These library calls do not support multi-threaded applications.

NAME

getprotoent, getprotobynumber, getprotobyname, setprotoent, endprotoent – get protocol entry

SYNOPSIS

#include <netdb.h>
struct protoent * **getprotoent**(void);

struct protoent * **getprotobyname**(char * *name*);

structprotoent ***getprotobynumber**(int *proto*);

void **setprotoent**(int *stayopen*);

void **endprotoent**(void);

DESCRIPTION

The `getprotoent()`, `getprotobyname()`, and `getprotobynumber()` functions each return a pointer to an object containing the broken-out fields of a line in the network protocol data base, `/etc/protocols`. The object has the following structure:

```
struct protoent {
   char*    p_name;      /* official name of protocol */
   char**   p_aliases;   /* alias list */
   int      p_proto;     /* protocol number */
};
```

The members of this structure are:

`p_name` The official name of the protocol.

`p_aliases` A zero terminated list of alternate names for the protocol.

`p_proto` The protocol number.

The `getprotoent()` function reads the next line of the file, opening the file if necessary.

The `setprotoent()` function opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to `getprotobyname()` or `getprotobynumber()`.

The `endprotoent()` function closes the file.

The `getprotobyname()` and `getprotobynumber()` functions sequentially search from the beginning of the file until a matching protocol name or protocol number is found, or until EOF is encountered (see RESTRICTIONS).

RETURN VALUES

A `NULL` pointer (`0`) is returned when `EOF` or an error is encountered.

FILES

`/etc/protocols`

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `protocols`(4CC)

BUGS These functions use a static data space; if the data is needed for future use, it should be copied before any subsequent calls overwrite it. Only the Internet protocols are currently understood.

RESTRICTIONS The `getprotobynumber()` function is not yet implemented in ChorusOS 4.0.

NAME

getservent, getservbyname, getservbyport, setservent, endservent – get service entry

#include <netdb.h>
struct servent * **getservent**(void);

struct servent * **getservbyname**(const char * *name*, const char * *proto*);

struct servent * **getservbyport**(int *port*, const char * *proto*);

void **setservent**(int *stayopen*);

void **endservent**(void);

DESCRIPTION

The `getservent()` , `getservbyname()` , `getservbyport()` functions each return a pointer to an object with the following structure containing the broken-out fields of a line in the network services data base, `/etc/services` .

```
struct  servent {
        char    *s_name;        /* official name of service */
        char    **s_aliases;    /* alias list */
        int     s_port;         /* port service resides at */
        char    *s_proto;       /* protocol to use */
};
```

The members of this structure are:

s_name	The official name of the service.
s_aliases	A zero-terminated list of alternate names for the service.
s_port	The port number at which the service resides. Port numbers are returned in network byte order.
s_proto	The name of the protocol to use when contacting the service.

The `getservent()` function reads the next line of the file, opening the file if necessary.

The `setservent()` function opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to `getservbyname()` or `getservbyport()` .

The `endservent()` function closes the file.

The `getservbyname()` and `getservbyport()` functions sequentially search from the beginning of the file until a matching protocol name or port number is found, or until `EOF` is encountered. If a protocol name is also supplied (non-`NULL`), searches must also match the protocol.

FILES

`/etc/services`

DIAGNOSTICS

Null pointer (`0`) returned on `EOF` or error.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`getprotoent`(3POSIX) , `services`(4CC)

NAME

sysconf – get configurable system variables

SYNOPSIS

#include <posix/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 *name* argument represents the system variable to be queried. The symbolic constants used for *name* are defined in `<posix/unistd.h>` and appear in the right-hand column of the table that follows.

Some of the variables in the left-hand column of the following table are independent of the system configuration and are also defined in `<limits.h>` or `<posix/unistd.h>` if the _POSIX_MRTP_SOURCE feature test macro is defined before these headers are included.

Variable	**Name Value**
ARG_MAX	_SC_ARG_MAX
DELAYTIMER_MAX	_SC_DELAYTIMER_MAX
_MQ_OPEN_MAX	_SC_MQ_OPEN_MAX
_MQ_PRIO_MAX	_SC_MQ_PRIO_MAX
_MQ_DFL_MSGSIZE	_SC_MQ_DFL_MSGSIZE
_MQ_DFL_MAXMSGNB	_SC_MQ_DFL_MAXMSGNB
_MQ_PATHMAX	_SC_MQ_PATHMAX
NGROUPS_MAX	_SC_NGROUPS_MAX
OPEN_MAX	_SC_OPEN_MAX
PAGESIZE	_SC_PAGESIZE
PTHREAD_DESTRUCTOR_ITERATIONS	_SC_PTHREAD_DESTRUCTOR_ITERATIONS
PTHREAD_KEYS_MAX	_SC_PTHREAD_KEYS_MAX
PTHREAD_STACK_MIN	_SC_PTHREAD_STACK_MIN
PTHREAD_THREADS_MAX	_SC_PTHREAD_THREADS_MAX
SEM_VALUE_MAX	_SC_SEM_VALUE_MAX
SHM_PATHMAX	_SC_SHM_PATHMAX
TIMER_MAX	_SC_TIMER_MAX
TZNAME_MAX	_SC_TZNAME_MAX

Variable	Name Value
_POSIX_MESSAGE_PASSING	_SC_MESSAGE_PASSING
_POSIX_SEMAPHORES	_SC_SEMAPHORES
_POSIX_SHARED_MEMORY_OBJECTS	_SC_SHARED_MEMORY_OBJECTS
_POSIX_THREADS	_SC_THREADS
_POSIX_THREAD_ATTR_STACKADDR	_SC_THREAD_ATTR_STACKADDR
_POSIX_THREAD_ATTR_STACKSIZE	_SC_THREAD_ATTR_STACKSIZE
_POSIX_THREAD_PRIORITY_SCHEDULING	_SC_THREAD_PRIORITY_SCHEDULING
_POSIX_THREAD_PRIO_INHERIT	_SC_THREAD_PRIO_INHERIT
_POSIX_THREAD_PRIO_PROTECT	_SC_THREAD_PRIO_PROTECT
_POSIX_THREAD_PROCESS_SHARED	_SC_THREAD_PROCESS_SHARED
_POSIX_THREAD_SAFE_FUNCTIONS	_SC_THREAD_SAFE_FUNCTIONS
_POSIX_TIMERS	_SC_TIMERS
_POSIX_VERSION	_SC_VERSION

RETURN VALUES

If *name* is an invalid value, `sysconf()` returns `-1`. If the variable corresponding to *name* is associated with a functionality that is not supported by the system, `sysconf()` returns `-1` without changing the value of `errno`.

Otherwise, `sysconf()` returns the current value of the variable on the system. The value returned is no more restrictive than the corresponding value passed to the application when it was compiled with `<limits.h>` or `<posix/unistd.h>`. The value does not change during the lifetime of the calling actor.

ERRORS

[EINVAL] The *name* argument is invalid.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

sysctl, sysctlbyname – get or set system information

SYNOPSIS

#include <sys/types.h>
#include <sys/sysctl.h>
int **`sysctl`**(int * *name*, u_int *namelen*, void * *oldp*, size_t * *oldlenp*, void * *newp*, size_t *newlen*);

int **`sysctlbyname`**(const char * *name*, void * *oldp*, size_t * *oldlenp*, void * *newp*, size_t *newlen*);

DESCRIPTION

`sysctl()` retrieves system information and allows processes with appropriate privileges to set system information. The information available from `sysctl()` consists of integers, strings, and tables. Information may be retrieved and set from the command interface using `sysctl`(1M) .

The state is described using a Management Information Base (MIB)-style name, listed in name, which is a *namelen* length array of integers.

The `sysctlbyname()` function accepts an ASCII representation of the name and internally looks up the integer name vector. Apart from that, it behaves in the same way as the standard `sysctl()` function.

EXTENDED DESCRIPTION

Unless explicitly noted below, `sysctl()` returns a consistent snapshot of the data requested. Consistency is obtained by locking the destination buffer into memory so that the data may be copied out without blocking. Calls to `sysctl()` are serialized to avoid deadlock.

The information is copied into the buffer specified by *oldp* . The size of the buffer is given by the location specified by *oldlenp* before the call, and that location gives the amount of data copied after a successful call and after a call that returns with the error code `ENOMEM` . If the amount of data available is greater than the size of the buffer supplied, the call supplies as much data as fits into the buffer provided and returns with the error code `ENOMEM` . If the old value is not desired, *oldp* and *oldlenp* should be set to `NULL` .

The size of the available data can be determined by calling `sysctl()` with a `NULL` parameter for *oldp* . The size of the available data will be returned in the location pointed to by *oldlenp* . For some operations, the amount of space may change often. For these operations, the system attempts to round up so that the returned size is large enough for a call to return the data shortly thereafter.

To set a new value, *newp* is set to point to a buffer of length *newlen* from which the requested value is to be taken. If a new value is not to be set, *newp* should be set to `NULL` and *newlen* set to `0` .

The top level names are defined with a `CTL_` prefix in `<sys/sysctl.h>` , and are as follows. The next and subsequent levels down are found in the header files listed here, and described below.

Name	Next level names	Description
CTL_HW	`sys/sysctl.h`	Generic CPU, I/O
CTL_KERN	`sys/sysctl.h`	High kernel limits
CTL_MQ	`posix/unistd.h`	Message queue
CTL_NET	`sys/socket.h`	Networking
CTL_SHM	`posix/unistd.h`	Shared memory
CTL_VFS	`sys/sysctl.h`	File system

`CTL_HW` The string and integer information available for the `CTL_HW` level is detailed below. The changeable column shows whether a process with appropriate privilege may change the value.

Second level name	Type	Changeable
`HW_MACHINE`	string	no

`HW_MACHINE`

The machine class.

`CTL_KERN` The string and integer information available for the `CTL_KERN` level is detailed below. The changeable column shows whether a process with appropriate privilege may change the value. The types of data currently available are process information, system vnodes, the open file entries, routing table entries, virtual memory statistics, load average history, and clock rate information.

Second level name	Type	Changeable
`KERN_BOOTTIME`	struct timeval	no
`KERN_FILE`	struct file	no
`KERN_HOSTID`	integer	yes
`KERN_HOSTNAME`	string	yes
`KERN_MAXFILES`	integer	yes
`KERN_MAXFILESPERPROC`	integer	yes
`KERN_MAXSOCKBUF`	integer	yes
`KERN_MAXVNODES`	integer	yes
`KERN_NGROUPS`	integer	no
`KERN_NISDOMAINNAME`	string	yes
`KERN_OSRELDATE`	integer	no

Second level name	**Type**	**Changeable**
KERN_OSRELEASE	string	no
KERN_OSREV	integer	no
KERN_OSTYPE	string	no
KERN_SOMAXCONN	integer	yes
KERN_UPDATEINTERVAL	integer	no
KERN_VERSION	string	no

KERN_BOOTTIME

A `struct timeval` structure is returned. This structure contains the time that the system was booted.

KERN_FILE

Returns the entire file table. The returned data consists of a single `struct filehead` followed by an array of `struct file` , whose size depends on the current number of these types of objects in the system.

KERN_HOSTID

Get or set the host ID .

KERN_HOSTNAME

Get or set the hostname.

KERN_MAXFILES

The maximum number of files that may be open in the system.

KERN_MAXFILESPERPROC

The maximum number of files that may be open for a single process. This limit only applies to processes with an effective `uid` of nonzero at the time of the open request. Files that have already been opened are not affected if the limit or the effective `uid` is changed.

KERN_MAXSOCKBUF

The maximum size of a socket buffer.

KERN_MAXVNODES

The maximum number of vnodes available on the system.

KERN_NGROUPS

The maximum number of supplementary groups.

KERN_NISDOMAINNAME

The name of the current YP / NIS domain.

KERN_OSRELDATE

The system release date in YYYYMM format (January 1996 is encoded as 199601).

KERN_OSRELEASE

The system release string.

KERN_OSREV

The system revision string.

KERN_OSTYPE

The system type string.

KERN_SOMAXCONN

The maximum number of connections when listening for events.

KERN_UPDATEINTERVAL

The system version string.

KERN_VERSION

Returns the entire vnode table. Note that the vnode table is not necessarily a consistent snapshot of the system. The returned data consists of an array whose size depends on the current number of these objects in the system. Each element of the array contains the kernel address of a vnode `struct vnode *` followed by the vnode itself `struct vnode` .

CTL_SHM The integer information available for the CTL_SHM level is detailed below. The `Changeable` column shows whether a process with appropriate privilege may change the value.

Second level name	**Type**	**Changeable**
_SC_SHARED_MEMORY_OBJECTS	int	no
_SC_SHM_PATHMAX	int	yes

_SC_SHARED_MEMORY_OBJECTS

Toggles the POSIX_SHM(5FEA) feature.

_SC_SHM_PATHMAX

Maximum path length for a shared memory object.

`CTL_NET` The string and integer information available for the `CTL_NET` level is detailed below. The changeable column shows whether a process with appropriate privilege may change the value.

Second level name	**Type**	**Changeable**
`PF_ROUTE`	routing messages	no
`PF_INET`	internet values	yes

`PF_ROUTE`

Returns the entire routing table or a subset of it. The data is returned as a sequence of routing messages. See `route`(4CC) for the header file format and meaning. The length of each message is contained in the message header.

The third level name is a protocol number, which is currently always `0` . The fourth level name is an address family, which may be set to `0` to select all address families. The fifth and sixth level names are as follows:

Fifth level name	**Sixth level is:**
`NET_RT_DUMP`	None
`NET_RT_FLAGS`	rtflags
`NET_RT_IFLIST`	None

The fifth level names are defined as follows:

`NET_RT_DUMP`	Dump internal routing protocol.
`NET_RT_FLAGS`	Resolve internal routing protocol.
`NET_RT_IFLIST`	Survey interface list.

`PF_INET`

Get or set various global information about the internet protocols. The information available for the five subtypes of the `PF_INET` level are detailed below. The `Changeable` column in each table shows whether a process with appropriate privilege may change the value.

The variables related to the `IPPROTO_ICMP` subtype are as follows:

Fourth level name	**Type**	**Changeable**
`ICMPCTL_MASKREPL`	int	yes
`ICMPCTL_STATS`	struct	no

ICMPCTL_MASKREPL	Netmask requests
ICMPCTL_STATS	Statistics

The variable related to the IPPROTO_IGMP subtype is as follows:

Fourth level name	Type	Changeable
IGMPCTL_STATS	struct	no

IGMPCTL_STATS	Statistics

The variables related to the IPPROTO_IP subtype are as follows:

Fourth level name	Type	Changeable
IPCTL_ACCEPTSOURCEROUTE	int	yes
IPCTL_FORWARDING	int	yes
IPCTL_INTRQDROPS	int	no
IPCTL_INTRQMAXLEN	int	no
IPCTL_SENDREDIRECTS	int	yes
IPCTL_RTEXPIRE	int	yes
IPCTL_RTMAXCACHE	int	yes
IPCTL_RTMINEXPIRE	int	yes
IPCTL_SOURCEROUTE	int	yes
IPCTL_DEFTTL	int	yes

IPCTL_ACCEPTSOURCEROUTE	Accept source routed packets
IPCTL_FORWARDING	Act as router
IPCTL_INTRQDROPS	Number of netisr queue drops
IPCTL_INTRQMAXLEN	Maximum length of netisr queue
IPCTL_SENDREDIRECTS	Send redirects when forwarding
IPCTL_RTEXPIRE	Cloned route expiration time
IPCTL_RTMAXCACHE	Trigger level for dynamic expire
IPCTL_RTMINEXPIRE	Minimum value for expiration time
IPCTL_SOURCEROUTE	Perform source routes
IPCTL_DEFTTL	Default TTL

The variables related to the IPPROTO_TCP subtype are as follows:

Fourth level name	Type	Changeable
TCPCTL_KEEPIDLE	int	yes
TCPCTL_KEEPINIT	int	yes
TCPCTL_KEEPINTVL	int	yes
TCPCTL_MSSDFLT	int	yes
TCPCTL_RECVSPACE	int	yes
TCPCTL_DO_RFC1323	int	yes
TCPCTL_DO_RFC1644	int	yes
TCPCTL_RTTDFLT	int	yes
TCPCTL_SENDSPACE	int	yes
TCPCTL_STATS	struct	no

TCPCTL_KEEPIDLE	Maximum before probing
TCPCTL_KEEPINIT	Maximum idle time during connect
TCPCTL_KEEPINTVL	Default probe interval
TCPCTL_MSSDFLT	MSS default
TCPCTL_RECVSPACE	Receive buffer space
TCPCTL_DO_RFC1323	*RFC1323* extensions
TCPCTL_DO_RFC1644	*RFC1644* extensions
TCPCTL_RTTDFLT	Default RTT estimate
TCPCTL_SENDSPACE	Send buffer space
TCPCTL_STATS	Statistics

The variables related to the IPPROTO_UDP subtype are as follows:

Fourth level name	Type	Changeable
UDPCTL_CHECKSUM	int	yes
UDPCTL_MAXDGRAM	int	yes
UDPCTL_RECVSPACE	int	yes
UDPCTL_STATS	int	no

UDPCTL_CHECKSUM	Checksum UDP packets
UDPCTL_MAXDGRAM	Maximum datagram size

UDPCTL_RECVSPACE	Default receive buffer space
UDPCTL_STATS	Statistics

CTL_MQ

The information available for the CTL_MQ level is detailed below. The Changeable column shows whether a process with appropriate privilege may change the value.

Second level name	Type	Changeable
_SC_MQ_OPEN_MAX	long	no
_SC_MQ_PRIO_MAX	long	no
_SC_MQ_DFL_MSGSIZE	long	no
_SC_MQ_MAXMSGNB	long	no
_SC_MQ_PATHMAX	long	no

_SC_MQ_OPEN_MAX

Maximum number of open message queues.

_SC_MQ_PRIO_MAX

Maximum number of message priorities.

_SC_MQ_MSGSIZE

Default message size of a message queue.

_SC_MQ_DFL_MAXMSGNB

Default maximum message number of a message queue.

_SC_MQ_PATHMAX

Maximum message queue object name size.

CTL_VFS

The information available for the CTL_VFS level is detailed below. The Changeable column shows whether a process with appropriate privilege may change the value.

Second level name	Type	Changeable
VFS_VFSCONF	struct	no
NFS_NFSSTATS	struct	no
NFS_NFSPRIVPORT	int	yes

VFS_VFSCONF

Get configured file systems.

`NFS_NFSSTATS`

Get NFS statistics.

`NFS_NFSPRIVPORT`

Prohibit NFS to resvports.

RETURN VALUES

If the call to `sysctl()` is successful, `0` is returned. Otherwise `-1` is returned and `errno` is set appropriately.

ERRORS

The following errors may be reported:

`EFAULT`	The buffer name, *oldp* , *newp* , or length pointer *oldlenp* contains an invalid address.
`EINVAL`	The name array is less than two or greater than `CTL_MAX_NAME` .
`EINVAL`	A non-null *newp* is given and its specified length in *newlen* is too large or too small.
`ENOMEM`	The length pointed to by *oldlenp* is too short to hold the requested value.
`ENOTDIR`	The name array specifies an intermediate rather than terminal name.
`EOPNOTSUPP`	The name array specifies a value that is unknown.
`EPERM`	An attempt was made to set a read-only value.
`EPERM`	A process without appropriate privilege attempted to set a value.

EXAMPLES

The following example retrieves the maximum number of processes allowed in the system:

```
int mib[2], maxproc;
size_t len;
mib[0] = CTL_KERN;
mib[1] = KERN_MAXPROC;
len = sizeof(maxproc);
sysctl(mib, 2, &maxproc, &len, NULL, 0);
```

The following example retrieves the standard search path for the system utilities:

```
int mib[2];
size_t len;
char *p;
mib[0] = CTL_USER;
mib[1] = USER_CS_PATH;
sysctl(mib, 2, NULL, &len, NULL, 0);
p = malloc(len);
sysctl(mib, 2, p, &len, NULL, 0);
```

FILES

For more information see the following files:

`<netinet/icmp_var.h>`	Definitions for fourth level ICMP identifiers.
`<netinet/in.h>`	Definitions for third level Internet identifiers and fourth level IP identifiers.
`<netinet/udp_var.h>`	Definitions for fourth level UDP identifiers.
`<sys/gmon.h>`	Definitions for third level profiling identifiers.
`<sys/socket.h>`	Definitions for second level network identifiers.
`<sys/sysctl.h>`	Definitions for top level identifiers, second level kernel and hardware identifiers, and user level identifiers.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`sysctl`(1M)

NAME

sysctl, sysctlbyname – get or set system information

SYNOPSIS

#include <sys/types.h>
#include <sys/sysctl.h>
int **sysctl**(int * *name*, u_int *namelen*, void * *oldp*, size_t * *oldlenp*, void * *newp*, size_t *newlen*);

int **sysctlbyname**(const char * *name*, void * *oldp*, size_t * *oldlenp*, void * *newp*, size_t *newlen*);

DESCRIPTION

`sysctl( )` retrieves system information and allows processes with appropriate privileges to set system information. The information available from `sysctl( )` consists of integers, strings, and tables. Information may be retrieved and set from the command interface using `sysctl`(1M) .

The state is described using a Management Information Base (MIB)-style name, listed in name, which is a *namelen* length array of integers.

The `sysctlbyname( )` function accepts an ASCII representation of the name and internally looks up the integer name vector. Apart from that, it behaves in the same way as the standard `sysctl( )` function.

EXTENDED DESCRIPTION

Unless explicitly noted below, `sysctl( )` returns a consistent snapshot of the data requested. Consistency is obtained by locking the destination buffer into memory so that the data may be copied out without blocking. Calls to `sysctl( )` are serialized to avoid deadlock.

The information is copied into the buffer specified by *oldp* . The size of the buffer is given by the location specified by *oldlenp* before the call, and that location gives the amount of data copied after a successful call and after a call that returns with the error code `ENOMEM` . If the amount of data available is greater than the size of the buffer supplied, the call supplies as much data as fits into the buffer provided and returns with the error code `ENOMEM` . If the old value is not desired, *oldp* and *oldlenp* should be set to `NULL` .

The size of the available data can be determined by calling `sysctl( )` with a `NULL` parameter for *oldp* . The size of the available data will be returned in the location pointed to by *oldlenp* . For some operations, the amount of space may change often. For these operations, the system attempts to round up so that the returned size is large enough for a call to return the data shortly thereafter.

To set a new value, *newp* is set to point to a buffer of length *newlen* from which the requested value is to be taken. If a new value is not to be set, *newp* should be set to `NULL` and *newlen* set to `0` .

The top level names are defined with a `CTL_` prefix in `<sys/sysctl.h>` , and are as follows. The next and subsequent levels down are found in the header files listed here, and described below.

Name	Next level names	Description
CTL_HW	`sys/sysctl.h`	Generic CPU, I/O
CTL_KERN	`sys/sysctl.h`	High kernel limits
CTL_MQ	`posix/unistd.h`	Message queue
CTL_NET	`sys/socket.h`	Networking
CTL_SHM	`posix/unistd.h`	Shared memory
CTL_VFS	`sys/sysctl.h`	File system

`CTL_HW` The string and integer information available for the `CTL_HW` level is detailed below. The changeable column shows whether a process with appropriate privilege may change the value.

Second level name	Type	Changeable
`HW_MACHINE`	string	no

`HW_MACHINE`

The machine class.

`CTL_KERN` The string and integer information available for the `CTL_KERN` level is detailed below. The changeable column shows whether a process with appropriate privilege may change the value. The types of data currently available are process information, system vnodes, the open file entries, routing table entries, virtual memory statistics, load average history, and clock rate information.

Second level name	Type	Changeable
`KERN_BOOTTIME`	struct timeval	no
`KERN_FILE`	struct file	no
`KERN_HOSTID`	integer	yes
`KERN_HOSTNAME`	string	yes
`KERN_MAXFILES`	integer	yes
`KERN_MAXFILESPERPROC`	integer	yes
`KERN_MAXSOCKBUF`	integer	yes
`KERN_MAXVNODES`	integer	yes
`KERN_NGROUPS`	integer	no
`KERN_NISDOMAINNAME`	string	yes
`KERN_OSRELDATE`	integer	no

Second level name	Type	Changeable
KERN_OSRELEASE	string	no
KERN_OSREV	integer	no
KERN_OSTYPE	string	no
KERN_SOMAXCONN	integer	yes
KERN_UPDATEINTERVAL	integer	no
KERN_VERSION	string	no

KERN_BOOTTIME

A `struct timeval` structure is returned. This structure contains the time that the system was booted.

KERN_FILE

Returns the entire file table. The returned data consists of a single `struct filehead` followed by an array of `struct file` , whose size depends on the current number of these types of objects in the system.

KERN_HOSTID

Get or set the host ID .

KERN_HOSTNAME

Get or set the hostname.

KERN_MAXFILES

The maximum number of files that may be open in the system.

KERN_MAXFILESPERPROC

The maximum number of files that may be open for a single process. This limit only applies to processes with an effective `uid` of nonzero at the time of the open request. Files that have already been opened are not affected if the limit or the effective `uid` is changed.

KERN_MAXSOCKBUF

The maximum size of a socket buffer.

KERN_MAXVNODES

The maximum number of vnodes available on the system.

KERN_NGROUPS

The maximum number of supplementary groups.

KERN_NISDOMAINNAME

The name of the current YP / NIS domain.

KERN_OSRELDATE

The system release date in YYYYMM format (January 1996 is encoded as 199601).

KERN_OSRELEASE

The system release string.

KERN_OSREV

The system revision string.

KERN_OSTYPE

The system type string.

KERN_SOMAXCONN

The maximum number of connections when listening for events.

KERN_UPDATEINTERVAL

The system version string.

KERN_VERSION

Returns the entire vnode table. Note that the vnode table is not necessarily a consistent snapshot of the system. The returned data consists of an array whose size depends on the current number of these objects in the system. Each element of the array contains the kernel address of a vnode `struct vnode *` followed by the vnode itself `struct vnode` .

CTL_SHM The integer information available for the CTL_SHM level is detailed below. The `Changeable` column shows whether a process with appropriate privilege may change the value.

Second level name	**Type**	**Changeable**
_SC_SHARED_MEMORY_OBJECTS	int	no
_SC_SHM_PATHMAX	int	yes

_SC_SHARED_MEMORY_OBJECTS

Toggles the POSIX_SHM(5FEA) feature.

_SC_SHM_PATHMAX

Maximum path length for a shared memory object.

`CTL_NET` The string and integer information available for the `CTL_NET` level is detailed below. The changeable column shows whether a process with appropriate privilege may change the value.

Second level name	Type	Changeable
`PF_ROUTE`	routing messages	no
`PF_INET`	internet values	yes

`PF_ROUTE`

Returns the entire routing table or a subset of it. The data is returned as a sequence of routing messages. See `route`(4CC) for the header file format and meaning. The length of each message is contained in the message header.

The third level name is a protocol number, which is currently always `0` . The fourth level name is an address family, which may be set to `0` to select all address families. The fifth and sixth level names are as follows:

Fifth level name	Sixth level is:
`NET_RT_DUMP`	None
`NET_RT_FLAGS`	rtflags
`NET_RT_IFLIST`	None

The fifth level names are defined as follows:

`NET_RT_DUMP`	Dump internal routing protocol.
`NET_RT_FLAGS`	Resolve internal routing protocol.
`NET_RT_IFLIST`	Survey interface list.

`PF_INET`

Get or set various global information about the internet protocols. The information available for the five subtypes of the `PF_INET` level are detailed below. The `Changeable` column in each table shows whether a process with appropriate privilege may change the value.

The variables related to the `IPPROTO_ICMP` subtype are as follows:

Fourth level name	Type	Changeable
`ICMPCTL_MASKREPL`	int	yes
`ICMPCTL_STATS`	struct	no

ICMPCTL_MASKREPL	Netmask requests
ICMPCTL_STATS	Statistics

The variable related to the IPPROTO_IGMP subtype is as follows:

Fourth level name	Type	Changeable
IGMPCTL_STATS	struct	no

IGMPCTL_STATS	Statistics

The variables related to the IPPROTO_IP subtype are as follows:

Fourth level name	Type	Changeable
IPCTL_ACCEPTSOURCEROUTE	int	yes
IPCTL_FORWARDING	int	yes
IPCTL_INTRQDROPS	int	no
IPCTL_INTRQMAXLEN	int	no
IPCTL_SENDREDIRECTS	int	yes
IPCTL_RTEXPIRE	int	yes
IPCTL_RTMAXCACHE	int	yes
IPCTL_RTMINEXPIRE	int	yes
IPCTL_SOURCEROUTE	int	yes
IPCTL_DEFTTL	int	yes

IPCTL_ACCEPTSOURCEROUTE	Accept source routed packets
IPCTL_FORWARDING	Act as router
IPCTL_INTRQDROPS	Number of netisr queue drops
IPCTL_INTRQMAXLEN	Maximum length of netisr queue
IPCTL_SENDREDIRECTS	Send redirects when forwarding
IPCTL_RTEXPIRE	Cloned route expiration time
IPCTL_RTMAXCACHE	Trigger level for dynamic expire
IPCTL_RTMINEXPIRE	Minimum value for expiration time
IPCTL_SOURCEROUTE	Perform source routes
IPCTL_DEFTTL	Default TTL

The variables related to the IPPROTO_TCP subtype are as follows:

Fourth level name	**Type**	**Changeable**
`TCPCTL_KEEPIDLE`	int	yes
`TCPCTL_KEEPINIT`	int	yes
`TCPCTL_KEEPINTVL`	int	yes
`TCPCTL_MSSDFLT`	int	yes
`TCPCTL_RECVSPACE`	int	yes
`TCPCTL_DO_RFC1323`	int	yes
`TCPCTL_DO_RFC1644`	int	yes
`TCPCTL_RTTDFLT`	int	yes
`TCPCTL_SENDSPACE`	int	yes
`TCPCTL_STATS`	struct	no

`TCPCTL_KEEPIDLE`	Maximum before probing
`TCPCTL_KEEPINIT`	Maximum idle time during connect
`TCPCTL_KEEPINTVL`	Default probe interval
`TCPCTL_MSSDFLT`	MSS default
`TCPCTL_RECVSPACE`	Receive buffer space
`TCPCTL_DO_RFC1323`	*RFC1323* extensions
`TCPCTL_DO_RFC1644`	*RFC1644* extensions
`TCPCTL_RTTDFLT`	Default RTT estimate
`TCPCTL_SENDSPACE`	Send buffer space
`TCPCTL_STATS`	Statistics

The variables related to the `IPPROTO_UDP` subtype are as follows:

Fourth level name	**Type**	**Changeable**
`UDPCTL_CHECKSUM`	int	yes
`UDPCTL_MAXDGRAM`	int	yes
`UDPCTL_RECVSPACE`	int	yes
`UDPCTL_STATS`	int	no

`UDPCTL_CHECKSUM`	Checksum UDP packets
`UDPCTL_MAXDGRAM`	Maximum datagram size

UDPCTL_RECVSPACE	Default receive buffer space
UDPCTL_STATS	Statistics

CTL_MQ

The information available for the CTL_MQ level is detailed below. The Changeable column shows whether a process with appropriate privilege may change the value.

Second level name	**Type**	**Changeable**
_SC_MQ_OPEN_MAX	long	no
_SC_MQ_PRIO_MAX	long	no
_SC_MQ_DFL_MSGSIZE	long	no
_SC_MQ_MAXMSGNB	long	no
_SC_MQ_PATHMAX	long	no

_SC_MQ_OPEN_MAX

Maximum number of open message queues.

_SC_MQ_PRIO_MAX

Maximum number of message priorities.

_SC_MQ_MSGSIZE

Default message size of a message queue.

_SC_MQ_DFL_MAXMSGNB

Default maximum message number of a message queue.

_SC_MQ_PATHMAX

Maximum message queue object name size.

CTL_VFS

The information available for the CTL_VFS level is detailed below. The Changeable column shows whether a process with appropriate privilege may change the value.

Second level name	**Type**	**Changeable**
VFS_VFSCONF	struct	no
NFS_NFSSTATS	struct	no
NFS_NFSPRIVPORT	int	yes

VFS_VFSCONF

Get configured file systems.

`NFS_NFSSTATS`

Get NFS statistics.

`NFS_NFSPRIVPORT`

Prohibit NFS to resvports.

RETURN VALUES

If the call to `sysctl()` is successful, `0` is returned. Otherwise `-1` is returned and `errno` is set appropriately.

ERRORS

The following errors may be reported:

`EFAULT`	The buffer name, *oldp* , *newp* , or length pointer *oldlenp* contains an invalid address.
`EINVAL`	The name array is less than two or greater than `CTL_MAX_NAME` .
`EINVAL`	A non-null *newp* is given and its specified length in *newlen* is too large or too small.
`ENOMEM`	The length pointed to by *oldlenp* is too short to hold the requested value.
`ENOTDIR`	The name array specifies an intermediate rather than terminal name.
`EOPNOTSUPP`	The name array specifies a value that is unknown.
`EPERM`	An attempt was made to set a read-only value.
`EPERM`	A process without appropriate privilege attempted to set a value.

EXAMPLES

The following example retrieves the maximum number of processes allowed in the system:

```
int mib[2], maxproc;
size_t len;
mib[0] = CTL_KERN;
mib[1] = KERN_MAXPROC;
len = sizeof(maxproc);
sysctl(mib, 2, &maxproc, &len, NULL, 0);
```

The following example retrieves the standard search path for the system utilities:

```
int mib[2];
size_t len;
char *p;
mib[0] = CTL_USER;
mib[1] = USER_CS_PATH;
sysctl(mib, 2, NULL, &len, NULL, 0);
p = malloc(len);
sysctl(mib, 2, p, &len, NULL, 0);
```

FILES

For more information see the following files:

`<netinet/icmp_var.h>`	Definitions for fourth level ICMP identifiers.
`<netinet/in.h>`	Definitions for third level Internet identifiers and fourth level IP identifiers.
`<netinet/udp_var.h>`	Definitions for fourth level UDP identifiers.
`<sys/gmon.h>`	Definitions for third level profiling identifiers.
`<sys/socket.h>`	Definitions for second level network identifiers.
`<sys/sysctl.h>`	Definitions for top level identifiers, second level kernel and hardware identifiers, and user level identifiers.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`sysctl`(1M)

NAME

tcsetattr, tcgetattr, cfgetispeed, cfsetispeed, cfgetospeed, cfsetospeed, cfsetspeed, cfmakeraw – manipulating the termios structure

SYNOPSIS

#include <termios.h>
speed_t **cfgetispeed**(struct termios * *t*);

int **cfsetispeed**(struct termios * *t*, speed_t *speed*);

speed_t **cfgetospeed**(struct termios * *t*);

int **cfsetospeed**(struct termios * *t*, speed_t *speed*);

int **cfsetspeed**(struct termios * *t*, speed_t *speed*);

void **cfmakeraw**(struct termios * *t*);

int **tcgetattr**(int *fd*, struct termios * *t*);

int **tcsetattr**(int *fd*, int *action*, struct termios * *t*);

FEATURES

VTTY

DESCRIPTION

The `cfmakeraw()`, `tcgetattr()` and `tcsetattr()` functions are provided for getting and setting the termios structure.

The `cfgetispeed()`, `cfsetispeed()`, `cfgetospeed()`, `cfsetospeed()` and `cfsetspeed()` functions are provided for getting and setting the baud rate values in the termios structure. The effects of the functions on the terminal as described below do not become effective, nor are all errors detected, until the `tcsetattr()` function is called. Certain values for baud rates set in the termios structure and passed to `tcsetattr()` have special meanings. These are discussed in the portion of the manual page that describes the `tcsetattr()` function.

GETTING AND SETTING THE BAUD RATE

The input and output baud rates are found in the termios structure. The unsigned integer `speed_t` is `typdef` ed in the include file `<termios.h>`. The value of the integer corresponds directly to the baud rate being represented, however, the following symbolic values are defined:

```
#define B0          0
#define B50         50
#define B75         75
#define B110        110
#define B134        134
#define B150        150
#define B200        200
#define B300        300
#define B600        600
#define B1200       1200
#define B1800       1800
#define B2400       2400
#define B4800       4800
#define B9600       9600
```

```
#define B19200      19200
#define B38400      38400
#ifndef _POSIX_SOURCE
#define EXTA        19200
#define EXTB        38400
#endif  /*_POSIX_SOURCE */
```

The `cfgetispeed( )` function returns the input baud rate in the termios structure referenced by *t* .

The `cfsetispeed( )` function sets the input baud rate in the termios structure referenced by *t* to *speed* . The `cfgetospeed( )` function returns the output baud rate in the termios structure referenced by *t* . The `cfsetospeed( )` function sets the output baud rate in the termios structure referenced by *t* to *speed* .

The `cfsetspeed( )` function sets both the input and output baud rate in the termios structure referenced by *t* to *speed* .

Upon successful completion, the functions `cfsetispeed( )` , `cfsetospeed( )` and `cfsetspeed( )` return a value of `0` . Otherwise, a value of `-1` is returned and the global variable `errno` is set to indicate the error.

GETTING AND SETTING THE TERMIOS STATE

This section describes the functions that are used to control the general terminal interface. Unless otherwise noted for a specific command, these functions are restricted from use by background processes. Attempts to perform these operations will cause the process group to be sent a `SIGTTOU` signal. If the calling process is blocking or ignoring `SIGTTOU` signals, the process is allowed to perform the operation and the `SIGTTOU` signal is not sent. See RESTRICTIONS below.

In all the functions, although *fd* is an open file descriptor, the functions affect the underlying terminal file, not just the open file description associated with the particular file descriptor.

The `cfmakeraw( )` function sets the flags stored in the termios structure to a state disabling all input and output processing, giving a raw I/O path. It should be noted that there is no function to reverse this effect. This is because there are a variety of processing options that could be re-enabled, and the correct method is for an application to snapshot the current terminal state using the `tcgetattr( )` function, setting raw mode using `cfmakeraw( )` and the subsequent `tcsetattr( )` , and then using another `tcsetattr( )` with the saved state to revert to the previous terminal state.

The `tcgetattr( )` function copies the parameters associated with the terminal referenced by *fd* in the termios structure referenced by *t* . This function is allowed from a background process (see RESTRICTIONS); however, the terminal attributes may subsequently be changed by a foreground process.

The `tcsetattr( )` function sets the parameters associated with the terminal from the termios structure referenced by *tp* . The *action* field is created by `or`-ing the following values, as specified in the include file `<termios.h>` .

TCSANOW	The change occurs immediately.
TCSADRAIN	The change occurs after all output written to *fd* has been transmitted to the terminal. This value of *action* should be used when changing parameters that affect output.
TCSAFLUSH	The change occurs after all output written to has been transmitted to the terminal. Additionally, any input that has been received but not read is discarded.
TCSASOFT	If this value is or'ed into the *action* value, the values of the *c_cflag* , *c_ispeed* , and *c_ospeed* fields are ignored.

The `0` baud rate is used to terminate the connection. If `0` is specified as the output speed to the function `tcsetattr( )` , modem control will no longer be asserted on the terminal, disconnecting the terminal.

If `0` is specified as the input speed to the function `tcsetattr( )` , the input baud rate will be set to the same value as that specified by the output baud rate.

If `tcsetattr( )` is unable to make any of the requested changes, it returns `-1` and sets `errno` . Otherwise, it makes all of the requested changes it can. If the specified input and output baud rates differ and are a combination that is not supported, neither baud rate is changed.

Upon successful completion, the functions `tcgetattr( )` and `tcsetattr( )` return a value of `0` . Otherwise, they return `-1` and the global variable `errno` is set to indicate one of the following error conditions:

[EBADF]	The *fd* argument to `tcgetattr( )` or `tcsetattr( )` was not a valid file descriptor.
[EINTR]	The `tcsetattr( )` function was interrupted by a signal. See RESTRICTIONS below.
[EINVAL]	The *action* argument to the `tcsetattr( )` function was not valid, or an attempt was made to change an attribute represented in the termios structure to an unsupported value.
[ENOTTY]	The file associated with the *fd* argument to `tcgetattr( )` or `tcsetattr( )` is not a terminal.

STANDARDS

The `cfgetispeed( )` , `cfsetispeed( )` , `cfgetospeed( )` , `cfsetospeed( )` , `tcgetattr( )` and `tcsetattr( )` functions are expected to be compliant with the POSIX 1003.1-88 specification. The `cfmakeraw( )` and `cfsetspeed( )` functions, as well as the `TCSASOFT` option to the `tcsetattr( )` function are extensions to the POSIX 1003.1-88 specification.

RESTRICTIONS

Signals and signals management are not supported.

These library functions (in `libbsd.a`) do not support multithreaded applications.

The background semantic is not supported.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

tcsetattr, tcgetattr, cfgetispeed, cfsetispeed, cfgetospeed, cfsetospeed, cfsetspeed, cfmakeraw – manipulating the termios structure

SYNOPSIS

#include <termios.h>
speed_t **cfgetispeed**(struct termios * *t*);

int **cfsetispeed**(struct termios * *t*, speed_t *speed*);

speed_t **cfgetospeed**(struct termios * *t*);

int **cfsetospeed**(struct termios * *t*, speed_t *speed*);

int **cfsetspeed**(struct termios * *t*, speed_t *speed*);

void **cfmakeraw**(struct termios * *t*);

int **tcgetattr**(int *fd*, struct termios * *t*);

int **tcsetattr**(int *fd*, int *action*, struct termios * *t*);

FEATURES

VTTY

DESCRIPTION

The `cfmakeraw()`, `tcgetattr()` and `tcsetattr()` functions are provided for getting and setting the termios structure.

The `cfgetispeed()`, `cfsetispeed()`, `cfgetospeed()`, `cfsetospeed()` and `cfsetspeed()` functions are provided for getting and setting the baud rate values in the termios structure. The effects of the functions on the terminal as described below do not become effective, nor are all errors detected, until the `tcsetattr()` function is called. Certain values for baud rates set in the termios structure and passed to `tcsetattr()` have special meanings. These are discussed in the portion of the manual page that describes the `tcsetattr()` function.

GETTING AND SETTING THE BAUD RATE

The input and output baud rates are found in the termios structure. The unsigned integer `speed_t` is `typdef` ed in the include file `<termios.h>`. The value of the integer corresponds directly to the baud rate being represented, however, the following symbolic values are defined:

```
#define B0          0
#define B50         50
#define B75         75
#define B110        110
#define B134        134
#define B150        150
#define B200        200
#define B300        300
#define B600        600
#define B1200       1200
#define B1800       1800
#define B2400       2400
#define B4800       4800
#define B9600       9600
```

```
#define B19200      19200
#define B38400      38400
#ifndef _POSIX_SOURCE
#define EXTA        19200
#define EXTB        38400
#endif  /*_POSIX_SOURCE */
```

The `cfgetispeed( )` function returns the input baud rate in the termios structure referenced by *t* .

The `cfsetispeed( )` function sets the input baud rate in the termios structure referenced by *t* to *speed* . The `cfgetospeed( )` function returns the output baud rate in the termios structure referenced by *t* . The `cfsetospeed( )` function sets the output baud rate in the termios structure referenced by *t* to *speed* .

The `cfsetspeed( )` function sets both the input and output baud rate in the termios structure referenced by *t* to *speed* .

Upon successful completion, the functions `cfsetispeed( )` , `cfsetospeed( )` and `cfsetspeed( )` return a value of `0` . Otherwise, a value of `-1` is returned and the global variable `errno` is set to indicate the error.

GETTING AND SETTING THE TERMIOS STATE

This section describes the functions that are used to control the general terminal interface. Unless otherwise noted for a specific command, these functions are restricted from use by background processes. Attempts to perform these operations will cause the process group to be sent a `SIGTTOU` signal. If the calling process is blocking or ignoring `SIGTTOU` signals, the process is allowed to perform the operation and the `SIGTTOU` signal is not sent. See RESTRICTIONS below.

In all the functions, although *fd* is an open file descriptor, the functions affect the underlying terminal file, not just the open file description associated with the particular file descriptor.

The `cfmakeraw( )` function sets the flags stored in the termios structure to a state disabling all input and output processing, giving a raw I/O path. It should be noted that there is no function to reverse this effect. This is because there are a variety of processing options that could be re-enabled, and the correct method is for an application to snapshot the current terminal state using the `tcgetattr( )` function, setting raw mode using `cfmakeraw( )` and the subsequent `tcsetattr( )` , and then using another `tcsetattr( )` with the saved state to revert to the previous terminal state.

The `tcgetattr( )` function copies the parameters associated with the terminal referenced by *fd* in the termios structure referenced by *t* . This function is allowed from a background process (see RESTRICTIONS); however, the terminal attributes may subsequently be changed by a foreground process.

The `tcsetattr( )` function sets the parameters associated with the terminal from the termios structure referenced by *tp* . The *action* field is created by `or`-ing the following values, as specified in the include file `<termios.h>` .

TCSANOW	The change occurs immediately.
TCSADRAIN	The change occurs after all output written to *fd* has been transmitted to the terminal. This value of *action* should be used when changing parameters that affect output.
TCSAFLUSH	The change occurs after all output written to has been transmitted to the terminal. Additionally, any input that has been received but not read is discarded.
TCSASOFT	If this value is or'ed into the *action* value, the values of the *c_cflag* , *c_ispeed* , and *c_ospeed* fields are ignored.

The `0` baud rate is used to terminate the connection. If `0` is specified as the output speed to the function `tcsetattr( )` , modem control will no longer be asserted on the terminal, disconnecting the terminal.

If `0` is specified as the input speed to the function `tcsetattr( )` , the input baud rate will be set to the same value as that specified by the output baud rate.

If `tcsetattr( )` is unable to make any of the requested changes, it returns `-1` and sets `errno` . Otherwise, it makes all of the requested changes it can. If the specified input and output baud rates differ and are a combination that is not supported, neither baud rate is changed.

Upon successful completion, the functions `tcgetattr( )` and `tcsetattr( )` return a value of `0` . Otherwise, they return `-1` and the global variable `errno` is set to indicate one of the following error conditions:

[EBADF]	The *fd* argument to `tcgetattr( )` or `tcsetattr( )` was not a valid file descriptor.
[EINTR]	The `tcsetattr( )` function was interrupted by a signal. See RESTRICTIONS below.
[EINVAL]	The *action* argument to the `tcsetattr( )` function was not valid, or an attempt was made to change an attribute represented in the termios structure to an unsupported value.
[ENOTTY]	The file associated with the *fd* argument to `tcgetattr( )` or `tcsetattr( )` is not a terminal.

STANDARDS

The `cfgetispeed( )` , `cfsetispeed( )` , `cfgetospeed( )` , `cfsetospeed( )` , `tcgetattr( )` and `tcsetattr( )` functions are expected to be compliant with the POSIX 1003.1-88 specification. The `cfmakeraw( )` and `cfsetspeed( )` functions, as well as the `TCSASOFT` option to the `tcsetattr( )` function are extensions to the POSIX 1003.1-88 specification.

RESTRICTIONS

Signals and signals management are not supported.

These library functions (in `libbsd.a`) do not support multithreaded applications.

The background semantic is not supported.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

directory, opendir, readdir, telldir, seekdir, rewinddir, closedir – directory operations

SYNOPSIS

#include <sys/types.h>
#include <dirent.h>
DIR * **opendir**(const char * *filename*);

struct dirent * **readdir**(DIR * *dirp*);

long **telldir**(const DIR * *dirp*);

void **seekdir**(DIR * *dirp*, long *loc*);

void **rewinddir**(DIR * *dirp*);

int **closedir**(DIR * *dirp*);

FEATURES

MSDOSFS, NFS_CLIENT, UFS

DESCRIPTION

The *opendir* function opens the directory named by *filename* , associates a directory stream with it and returns a pointer to be used to identify the directory stream in subsequent operations. The `NULL` pointer is returned if *filename* cannot be accessed, or if it cannot *malloc(3STDC)* enough memory to hold all of it.

The *readdir* function returns a pointer to the next directory entry. It returns `NULL` upon reaching the end of the directory or detecting an invalid *seekdir* operation.

The *telldir* function returns the current location associated with the named directory stream.

The *seekdir* function sets the position of the next *readdir* operation on the directory stream. The new position reverts to the one associated with the directory stream when the *telldir* operation was performed. Values returned by *telldir* are valid only for the lifetime of the `DIR` pointer, *dirp* , from which they are derived. If the directory is closed and then reopened, the *telldir* value may be invalidated due to undetected directory compaction. It is safe to use a previous *telldir* value immediately after a call to *opendir* and before any calls to *readdir* .

The *rewinddir* function resets the position of the named directory stream to the beginning of the directory.

The *closedir* function closes the named directory stream and frees the structure associated with the *dirp* pointer, returning 0 on success. On failure, –1 is returned and the global variable *errno* is set to indicate the error.

Sample code which searches a directory for the “name” entry is:

```
len = strlen(name);
dirp = opendir(".");
if (dirp) {
   while ((dp = readdir(dirp)) != NULL)
 if (dp->d_namlen == len && !strcmp(dp->d_name, name)) {
  (void) closedir(dirp);
  return FOUND;
         }
   (void) closedir(dirp);
   }
return NOT_FOUND;
```

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`open`(2POSIX) , `close`(2POSIX) , `read`(2POSIX) , `lseek`(2POSIX)

HISTORY

The *opendir, readdir, telldir, seekdir, rewinddir* and *closedir* functions appeared in 4.2 BSD.

RESTRICTIONS

These library calls do not support multi-threaded applications.

NAME

timer_create, timer_delete – create or delete a timer

SYNOPSIS

#include <time.h>
#include <signal.h>
int **timer_create**(clockid_t *clock_id*, struct sigevent * *evp*, timer_t * *timerid*);

int **timer_delete**(timer_t *timerid*);

DESCRIPTION

The *timer_create* function creates a timer in the current actor and returns the identifier of the timer in *timerid* . This identifier is valid during the life of the actor unless deleted using *timer_delete* . The *clock_id* must be set to CLOCK_REALTIME, the system realtime clock, which is the timing base for the new timer. The timer is disarmed on return from *timer_create* .

The *evp* argument must point to a *sigevent* structure allocated by the caller. Within this structure, the *sigev_notify* member must be equal to SIGEV_THREAD, and the *sigev_notify_function* member must point to a caller-provided function to be executed when the timer expires. This notify function is formally defined as follows:

```
void notify_function ( union sigval value );
```

where the *value* argument is obtained from the *sigev_value* member of the *sigevent* structure. The *sigevent* structure and *sigval* union are defined as follows (in <signal.h>):

```
struct sigevent {
    int            sigev_notify;
    int            sigev_signo;          /* not used  */
    union sigval   sigev_value;
    void           (*sigev_notify_function)(union sigval);
};
union sigval {
    int     sival_int;
    void*   sival_ptr;
};
```

At each timer expiration, the notify function is executed asynchronously in a separate handler thread associated with the timer. This handler thread is created automatically by *timer_create* and need not normally be manipulated by the user application. It is a pthread with *detachstate* set to PTHREAD_CREATE_DETACHED, a scheduling policy set to that of the caller of *timer_create* , and priority set one level higher than that of the caller of *timer_create* unless the caller's priority is already at the maximum for the policy (see *pthread_create* (3POSIX), *pthread_attr_setdetachstate* (3POSIX), *pthread_attr_setschedparam* (3POSIX), and *sched_get_priority_max* (3POSIX)). The handler thread is deleted automatically when the timer is deleted. If the thread is deleted for any other reason while its associated timer is still active, the timer will be disabled and further attempts to arm it will return an error (see *timer_settime* (3POSIX)).

The *timer_delete* function disarms (if necessary) and deletes the specified timer. If the associated notify function is still executing, it will be allowed to complete before the handler thread is deleted.

RETURN VALUE

Upon successful completion, *timer_create* and *timer_delete* return zero. In case of error a value of –1 is returned and *errno* is set to indicate the error condition.

ERRORS

[EINVAL]	The *clock_id* argument specifies a clock other than CLOCK_REALTIME. The *evp* argument is NULL, or the referenced *sigevent* structure is not initialized as specified above (*timer_create* only). The *timerid* argument is NULL (*timer_create*) or *timerid* does not reference a currently valid timer (*timer_delete*).
[EAGAIN]	Insufficient system resources are available to satisfy the request (*timer_create* only).
[ENOSYS]	*timer_create* was called from a thread which is not a pthread.
[EFAULT]	A pointer argument contains an address outside the current actor's address space.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`clock_gettime`(3POSIX) , `timer_settime`(3POSIX)

NAME

timer_create, timer_delete – create or delete a timer

SYNOPSIS

#include <time.h>
#include <signal.h>
int **timer_create**(clockid_t *clock_id*, struct sigevent * *evp*, timer_t * *timerid*);

int **timer_delete**(timer_t *timerid*);

DESCRIPTION

The *timer_create* function creates a timer in the current actor and returns the identifier of the timer in *timerid* . This identifier is valid during the life of the actor unless deleted using *timer_delete* . The *clock_id* must be set to CLOCK_REALTIME, the system realtime clock, which is the timing base for the new timer. The timer is disarmed on return from *timer_create* .

The *evp* argument must point to a *sigevent* structure allocated by the caller. Within this structure, the *sigev_notify* member must be equal to SIGEV_THREAD, and the *sigev_notify_function* member must point to a caller-provided function to be executed when the timer expires. This notify function is formally defined as follows:

```
void notify_function ( union sigval value );
```

where the *value* argument is obtained from the *sigev_value* member of the *sigevent* structure. The *sigevent* structure and *sigval* union are defined as follows (in <signal.h>):

```
struct sigevent {
    int           sigev_notify;
    int           sigev_signo;          /* not used  */
    union sigval  sigev_value;
    void          (*sigev_notify_function)(union sigval);
};
union sigval {
    int     sival_int;
    void*   sival_ptr;
};
```

At each timer expiration, the notify function is executed asynchronously in a separate handler thread associated with the timer. This handler thread is created automatically by *timer_create* and need not normally be manipulated by the user application. It is a pthread with *detachstate* set to PTHREAD_CREATE_DETACHED, a scheduling policy set to that of the caller of *timer_create* , and priority set one level higher than that of the caller of *timer_create* unless the caller’s priority is already at the maximum for the policy (see *pthread_create* (3POSIX), *pthread_attr_setdetachstate* (3POSIX), *pthread_attr_setschedparam* (3POSIX), and *sched_get_priority_max* (3POSIX)). The handler thread is deleted automatically when the timer is deleted. If the thread is deleted for any other reason while its associated timer is still active, the timer will be disabled and further attempts to arm it will return an error (see *timer_settime* (3POSIX)).

The *timer_delete* function disarms (if necessary) and deletes the specified timer. If the associated notify function is still executing, it will be allowed to complete before the handler thread is deleted.

RETURN VALUE

Upon successful completion, *timer_create* and *timer_delete* return zero. In case of error a value of –1 is returned and *errno* is set to indicate the error condition.

ERRORS

[EINVAL]	The *clock_id* argument specifies a clock other than CLOCK_REALTIME. The *evp* argument is NULL, or the referenced *sigevent* structure is not initialized as specified above (*timer_create* only). The *timerid* argument is NULL (*timer_create*) or *timerid* does not reference a currently valid timer (*timer_delete*).
[EAGAIN]	Insufficient system resources are available to satisfy the request (*timer_create* only).
[ENOSYS]	*timer_create* was called from a thread which is not a pthread.
[EFAULT]	A pointer argument contains an address outside the current actor's address space.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`clock_gettime`(3POSIX) , `timer_settime`(3POSIX)

NAME

timer_settime, timer_gettime, timer_getoverrun – set and arm or disarm a timer, obtain remaining interval for an active timer, or obtain current overrun count for a timer

SYNOPSIS

#include <time.h>
int **timer_settime**(timer_t *timerid*, int *flags*, const struct itimerspec * *value*, struct itimerspec * *ovalue*);

int **timer_gettime**(timer_t *timerid*, struct itimerspec * *value*);

int **timer_getoverrun**(timer_t *timerid*);

DESCRIPTION

The *timer_settime* function arms, resets, or disarms the timer specified by *timerid* . If the *it_value* member of the *value* argument is non-zero, the time of the next expiration is set accordingly (see next paragraph) and the timer is armed. If the timer was already armed, the time of the next expiration is modified accordingly. If the *it_value* member of *value* is zero, the timer is disarmed. Disarming or resetting a timer has no effect on either a pending notification, a concurrent execution of the notify function, or the timer's overrun count.

If the bit flag TIMER_ABSTIME is not set in the *flags* argument, the time of the next expiration is set to the interval specified in the *it_value* member of *value* relative to the current time. This means that the next expiration will occur *value–>it_value.tv_sec* seconds plus *value–>it_value.tv_nsec* nanoseconds after the *timer_settime* call. If the flag TIMER_ABSTIME is set in *flags* , the next expiration of the timer will occur when the clock associated with *timerid* reaches the value specified in the *it_value* member of *value* . If the time specified has already passed, *timer_settime* will succeed and the expiration notification will be sent.

If the timer is armed (or reset) by a call to *timer_settime*) and the *it_interval* member of *value* is non-zero, a periodic (repetitive) timer is specified. At each expiration, the timer is immediately and automatically re-armed from *value–>it_interval* . This value is treated as a relative interval regardless of the setting of the *flags* argument in the most recent *timer_settime* call.

Time values that are between two consecutive non-negative integer multiples of the resolution of the timer specified are rounded up to the larger multiple of the resolution. Any incremental quantity errors will not cause the timer to expire earlier than the rounded-up time value.

If the *ovalue* argument is not NULL, *timer_settime* will store, at the location referenced by *ovalue* , the previous amount of time remaining before the timer would have expired (zero if the timer was disarmed), and the previous reload value. The time remaining is stored as a relative interval even if the timer was armed with an absolute time. These values are stored before the state of the timer is changed in any way as a result of the current *timer_settime* call. The members of *ovalue* are subject to the resolution of the timer.

The *timer_gettime* function stores, at the location referenced by *value* , the amount of time remaining before the timer expires (zero if the timer is disarmed), and the reload value specified in the most recent *timer_settime* call. The *timer_gettime* function is equivalent to *timer_settime* with a NULL *value* argument, and returns the identical information.

At most, a single notification is active for a given timer at any one time. If the timer expires while its corresponding notify function (see *timer_create* (3POSIX)) is still in execution from a previous notification, an overrun occurs. When the notify function subsequently returns, it will be re-invoked immediately, and the *timer_getoverrun* call may then be used to obtain the overrun count. An overrun count value pertains to a particular execution of the notification function; the value returned by *timer_getoverrun* does not change during that execution. The overrun count is defined as the number of timer expirations that occurred after the previous invocation of the notify function, but before that invocation returned. Thus, for a periodic timer, an overrun count equal to one indicates that the current invocation was delayed, but by less than the period interval. The *timer_getoverrun* function returns a maximum value of _POSIX_DELAYTIMER_MAX if the actual overrun count is greater than or equal to that value.

Because the notify function is executed by a thread, timely notification of timer expiration can be impeded by activity elsewhere on the system of higher priority than the handler thread (see *timer_create* (3POSIX))..

RETURN VALUE

Upon successful completion, *timer_settime* and *timer_gettime* return zero, and *timer_getoverrun* returns the timer overrun count as described above. In case of error a value of –1 is returned by all three functions, and *errno* is set to indicate one of the following error conditions.

ERRORS

[EINVAL]	The *timerid* argument does not reference a currently valid timer, or the handler thread for the timer has been deleted. The *flags* argument contains an invalid value. The time specification in either the *it_value* member or the *it_interval* member of the *value* argument to *timer_settime* contains an invalid value.
[EFAULT]	A pointer argument contains an address outside the current actor's address space.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `timer_create`(3POSIX)

NAME

timer_settime, timer_gettime, timer_getoverrun – set and arm or disarm a timer, obtain remaining interval for an active timer, or obtain current overrun count for a timer

SYNOPSIS

#include <time.h>
int **timer_settime**(timer_t *timerid*, int *flags*, const struct itimerspec * *value*, struct itimerspec * *ovalue*);

int **timer_gettime**(timer_t *timerid*, struct itimerspec * *value*);

int **timer_getoverrun**(timer_t *timerid*);

DESCRIPTION

The *timer_settime* function arms, resets, or disarms the timer specified by *timerid* . If the *it_value* member of the *value* argument is non-zero, the time of the next expiration is set accordingly (see next paragraph) and the timer is armed. If the timer was already armed, the time of the next expiration is modified accordingly. If the *it_value* member of *value* is zero, the timer is disarmed. Disarming or resetting a timer has no effect on either a pending notification, a concurrent execution of the notify function, or the timer's overrun count.

If the bit flag TIMER_ABSTIME is not set in the *flags* argument, the time of the next expiration is set to the interval specified in the *it_value* member of *value* relative to the current time. This means that the next expiration will occur *value–>it_value.tv_sec* seconds plus *value–>it_value.tv_nsec* nanoseconds after the *timer_settime* call. If the flag TIMER_ABSTIME is set in *flags* , the next expiration of the timer will occur when the clock associated with *timerid* reaches the value specified in the *it_value* member of *value* . If the time specified has already passed, *timer_settime* will succeed and the expiration notification will be sent.

If the timer is armed (or reset) by a call to *timer_settime*) and the *it_interval* member of *value* is non-zero, a periodic (repetitive) timer is specified. At each expiration, the timer is immediately and automatically re-armed from *value–>it_interval* . This value is treated as a relative interval regardless of the setting of the *flags* argument in the most recent *timer_settime* call.

Time values that are between two consecutive non-negative integer multiples of the resolution of the timer specified are rounded up to the larger multiple of the resolution. Any incremental quantity errors will not cause the timer to expire earlier than the rounded-up time value.

If the *ovalue* argument is not NULL, *timer_settime* will store, at the location referenced by *ovalue* , the previous amount of time remaining before the timer would have expired (zero if the timer was disarmed), and the previous reload value. The time remaining is stored as a relative interval even if the timer was armed with an absolute time. These values are stored before the state of the timer is changed in any way as a result of the current *timer_settime* call. The members of *ovalue* are subject to the resolution of the timer.

The *timer_gettime* function stores, at the location referenced by *value* , the amount of time remaining before the timer expires (zero if the timer is disarmed), and the reload value specified in the most recent *timer_settime* call. The *timer_gettime* function is equivalent to *timer_settime* with a NULL *value* argument, and returns the identical information.

At most, a single notification is active for a given timer at any one time. If the timer expires while its corresponding notify function (see *timer_create* (3POSIX)) is still in execution from a previous notification, an overrun occurs. When the notify function subsequently returns, it will be re-invoked immediately, and the *timer_getoverrun* call may then be used to obtain the overrun count. An overrun count value pertains to a particular execution of the notification function; the value returned by *timer_getoverrun* does not change during that execution. The overrun count is defined as the number of timer expirations that occurred after the previous invocation of the notify function, but before that invocation returned. Thus, for a periodic timer, an overrun count equal to one indicates that the current invocation was delayed, but by less than the period interval. The *timer_getoverrun* function returns a maximum value of _POSIX_DELAYTIMER_MAX if the actual overrun count is greater than or equal to that value.

Because the notify function is executed by a thread, timely notification of timer expiration can be impeded by activity elsewhere on the system of higher priority than the handler thread (see *timer_create* (3POSIX))..

RETURN VALUE

Upon successful completion, *timer_settime* and *timer_gettime* return zero, and *timer_getoverrun* returns the timer overrun count as described above. In case of error a value of –1 is returned by all three functions, and *errno* is set to indicate one of the following error conditions.

ERRORS

[EINVAL]	The *timerid* argument does not reference a currently valid timer, or the handler thread for the timer has been deleted. The *flags* argument contains an invalid value. The time specification in either the *it_value* member or the *it_interval* member of the *value* argument to *timer_settime* contains an invalid value.
[EFAULT]	A pointer argument contains an address outside the current actor’s address space.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`timer_create`(3POSIX)

NAME

timer_settime, timer_gettime, timer_getoverrun – set and arm or disarm a timer, obtain remaining interval for an active timer, or obtain current overrun count for a timer

SYNOPSIS

#include <time.h>
int **timer_settime**(timer_t *timerid*, int *flags*, const struct itimerspec * *value*, struct itimerspec * *ovalue*);

int **timer_gettime**(timer_t *timerid*, struct itimerspec * *value*);

int **timer_getoverrun**(timer_t *timerid*);

DESCRIPTION

The *timer_settime* function arms, resets, or disarms the timer specified by *timerid* . If the *it_value* member of the *value* argument is non-zero, the time of the next expiration is set accordingly (see next paragraph) and the timer is armed. If the timer was already armed, the time of the next expiration is modified accordingly. If the *it_value* member of *value* is zero, the timer is disarmed. Disarming or resetting a timer has no effect on either a pending notification, a concurrent execution of the notify function, or the timer's overrun count.

If the bit flag TIMER_ABSTIME is not set in the *flags* argument, the time of the next expiration is set to the interval specified in the *it_value* member of *value* relative to the current time. This means that the next expiration will occur *value–>it_value.tv_sec* seconds plus *value–>it_value.tv_nsec* nanoseconds after the *timer_settime* call. If the flag TIMER_ABSTIME is set in *flags* , the next expiration of the timer will occur when the clock associated with *timerid* reaches the value specified in the *it_value* member of *value* . If the time specified has already passed, *timer_settime* will succeed and the expiration notification will be sent.

If the timer is armed (or reset) by a call to *timer_settime*) and the *it_interval* member of *value* is non-zero, a periodic (repetitive) timer is specified. At each expiration, the timer is immediately and automatically re-armed from *value–>it_interval* . This value is treated as a relative interval regardless of the setting of the *flags* argument in the most recent *timer_settime* call.

Time values that are between two consecutive non-negative integer multiples of the resolution of the timer specified are rounded up to the larger multiple of the resolution. Any incremental quantity errors will not cause the timer to expire earlier than the rounded-up time value.

If the *ovalue* argument is not NULL, *timer_settime* will store, at the location referenced by *ovalue* , the previous amount of time remaining before the timer would have expired (zero if the timer was disarmed), and the previous reload value. The time remaining is stored as a relative interval even if the timer was armed with an absolute time. These values are stored before the state of the timer is changed in any way as a result of the current *timer_settime* call. The members of *ovalue* are subject to the resolution of the timer.

The *timer_gettime* function stores, at the location referenced by *value* , the amount of time remaining before the timer expires (zero if the timer is disarmed), and the reload value specified in the most recent *timer_settime* call. The *timer_gettime* function is equivalent to *timer_settime* with a NULL *value* argument, and returns the identical information.

At most, a single notification is active for a given timer at any one time. If the timer expires while its corresponding notify function (see *timer_create* (3POSIX)) is still in execution from a previous notification, an overrun occurs. When the notify function subsequently returns, it will be re-invoked immediately, and the *timer_getoverrun* call may then be used to obtain the overrun count. An overrun count value pertains to a particular execution of the notification function; the value returned by *timer_getoverrun* does not change during that execution. The overrun count is defined as the number of timer expirations that occurred after the previous invocation of the notify function, but before that invocation returned. Thus, for a periodic timer, an overrun count equal to one indicates that the current invocation was delayed, but by less than the period interval. The *timer_getoverrun* function returns a maximum value of _POSIX_DELAYTIMER_MAX if the actual overrun count is greater than or equal to that value.

Because the notify function is executed by a thread, timely notification of timer expiration can be impeded by activity elsewhere on the system of higher priority than the handler thread (see *timer_create* (3POSIX))..

RETURN VALUE

Upon successful completion, *timer_settime* and *timer_gettime* return zero, and *timer_getoverrun* returns the timer overrun count as described above. In case of error a value of –1 is returned by all three functions, and *errno* is set to indicate one of the following error conditions.

ERRORS

[EINVAL]	The *timerid* argument does not reference a currently valid timer, or the handler thread for the timer has been deleted. The *flags* argument contains an invalid value. The time specification in either the *it_value* member or the *it_interval* member of the *value* argument to *timer_settime* contains an invalid value.
[EFAULT]	A pointer argument contains an address outside the current actor’s address space.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`timer_create`(3POSIX)

NAME

err, verr, verrx, warn, vwarn, warnx, vwarnx – formatted error messages

SYNOPSIS

#include <err.h>
void **err**(int *eval*, const char * *fmt*, ...);

void **verr**(int *eval*, const char * *fmt*, va_list *args*);

void **errx**(int *eval*, const char * *fmt*, ...);

void **verrx**(int *eval*, const char * *fmt*, va_list *args*);

void **warn**(const char * *fmt*, ...);

void **vwarn**(const char * *fmt*, va_list *args*);

void **warnx**(const char * *fmt*, ...);

void **vwarnx**(const char * *fmt*, va_list *args*);

DESCRIPTION

The *err* and *warn* family of functions display a formatted error message to the standard error output. If the *fmt* argument is not NULL , the formatted error message, a colon character, and a space are output. In the case of the *err* , *verr* , *warn* , and *vwarn* functions, the error message string affiliated with the current value of the global variable *errno* is output. In all cases, the output is followed by a newline character.

The *err* , *verr* , *errx* , and *verrx* functions do not return, but exit with the value of the argument *eval* .

EXAMPLES

Display the current errno information string and exit:

```
if ((p = malloc(size)) == NULL)
        err(1, NULL);
if ((fd = open(file_name, O_RDONLY, 0)) == -1)
        err(1, "%s", file_name);
```

Display an error message and exit:

```
if (tm.tm_hour < START_TIME)
        errx(1, "too early, wait until %s", start_time_string);
```

Warn of an error:

```
if ((fd = open(raw_device, O_RDONLY, 0)) == -1)
        warnx("%s: %s: trying the block device",
            raw_device, strerror(errno));
if ((fd = open(block_device, O_RDONLY, 0)) == -1)
        warn("%s", block_device);
```

HISTORY

The *err* and *warn* functions appeared in 4.4 BSD.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `strerror`(3STDC)

NAME

err, verr, verrx, warn, vwarn, warnx, vwarnx – formatted error messages

SYNOPSIS

#include <err.h>
void **err**(int *eval*, const char * *fmt*, ...);

void **verr**(int *eval*, const char * *fmt*, va_list *args*);

void **errx**(int *eval*, const char * *fmt*, ...);

void **verrx**(int *eval*, const char * *fmt*, va_list *args*);

void **warn**(const char * *fmt*, ...);

void **vwarn**(const char * *fmt*, va_list *args*);

void **warnx**(const char * *fmt*, ...);

void **vwarnx**(const char * *fmt*, va_list *args*);

DESCRIPTION

The *err* and *warn* family of functions display a formatted error message to the standard error output. If the *fmt* argument is not `NULL` , the formatted error message, a colon character, and a space are output. In the case of the *err* , *verr* , *warn* , and *vwarn* functions, the error message string affiliated with the current value of the global variable *errno* is output. In all cases, the output is followed by a newline character.

The *err* , *verr* , *errx* , and *verrx* functions do not return, but exit with the value of the argument *eval* .

EXAMPLES

Display the current errno information string and exit:

```
if ((p = malloc(size)) == NULL)
        err(1, NULL);
if ((fd = open(file_name, O_RDONLY, 0)) == -1)
        err(1, "%s", file_name);
```

Display an error message and exit:

```
if (tm.tm_hour < START_TIME)
        errx(1, "too early, wait until %s", start_time_string);
```

Warn of an error:

```
if ((fd = open(raw_device, O_RDONLY, 0)) == -1)
        warnx("%s: %s: trying the block device",
            raw_device, strerror(errno));
if ((fd = open(block_device, O_RDONLY, 0)) == -1)
        warn("%s", block_device);
```

HISTORY

The *err* and *warn* functions appeared in 4.4 BSD.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `strerror`(3STDC)

NAME

err, verr, verrx, warn, vwarn, warnx, vwarnx – formatted error messages

SYNOPSIS

#include <err.h>
void **err**(int *eval*, const char * *fmt*, ...);

void **verr**(int *eval*, const char * *fmt*, va_list *args*);

void **errx**(int *eval*, const char * *fmt*, ...);

void **verrx**(int *eval*, const char * *fmt*, va_list *args*);

void **warn**(const char * *fmt*, ...);

void **vwarn**(const char * *fmt*, va_list *args*);

void **warnx**(const char * *fmt*, ...);

void **vwarnx**(const char * *fmt*, va_list *args*);

DESCRIPTION

The *err* and *warn* family of functions display a formatted error message to the standard error output. If the *fmt* argument is not NULL , the formatted error message, a colon character, and a space are output. In the case of the *err* , *verr* , *warn* , and *vwarn* functions, the error message string affiliated with the current value of the global variable *errno* is output. In all cases, the output is followed by a newline character.

The *err* , *verr* , *errx* , and *verrx* functions do not return, but exit with the value of the argument *eval* .

EXAMPLES

Display the current errno information string and exit:

```
if ((p = malloc(size)) == NULL)
        err(1, NULL);
if ((fd = open(file_name, O_RDONLY, 0)) == -1)
        err(1, "%s", file_name);
```

Display an error message and exit:

```
if (tm.tm_hour < START_TIME)
        errx(1, "too early, wait until %s", start_time_string);
```

Warn of an error:

```
if ((fd = open(raw_device, O_RDONLY, 0)) == -1)
        warnx("%s: %s: trying the block device",
            raw_device, strerror(errno));
if ((fd = open(block_device, O_RDONLY, 0)) == -1)
        warn("%s", block_device);
```

HISTORY

The *err* and *warn* functions appeared in 4.4 BSD.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `strerror`(3STDC)

NAME

err, verr, verrx, warn, vwarn, warnx, vwarnx – formatted error messages

SYNOPSIS

#include <err.h>
void **err**(int *eval*, const char * *fmt*, ...);

void **verr**(int *eval*, const char * *fmt*, va_list *args*);

void **errx**(int *eval*, const char * *fmt*, ...);

void **verrx**(int *eval*, const char * *fmt*, va_list *args*);

void **warn**(const char * *fmt*, ...);

void **vwarn**(const char * *fmt*, va_list *args*);

void **warnx**(const char * *fmt*, ...);

void **vwarnx**(const char * *fmt*, va_list *args*);

DESCRIPTION

The *err* and *warn* family of functions display a formatted error message to the standard error output. If the *fmt* argument is not NULL , the formatted error message, a colon character, and a space are output. In the case of the *err* , *verr* , *warn* , and *vwarn* functions, the error message string affiliated with the current value of the global variable *errno* is output. In all cases, the output is followed by a newline character.

The *err* , *verr* , *errx* , and *verrx* functions do not return, but exit with the value of the argument *eval* .

EXAMPLES

Display the current errno information string and exit:

```
if ((p = malloc(size)) == NULL)
        err(1, NULL);
if ((fd = open(file_name, O_RDONLY, 0)) == -1)
        err(1, "%s", file_name);
```

Display an error message and exit:

```
if (tm.tm_hour < START_TIME)
        errx(1, "too early, wait until %s", start_time_string);
```

Warn of an error:

```
if ((fd = open(raw_device, O_RDONLY, 0)) == -1)
        warnx("%s: %s: trying the block device",
            raw_device, strerror(errno));
if ((fd = open(block_device, O_RDONLY, 0)) == -1)
        warn("%s", block_device);
```

HISTORY

The *err* and *warn* functions appeared in 4.4 BSD.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`strerror`(3STDC)

NAME

err, verr, verrx, warn, vwarn, warnx, vwarnx – formatted error messages

SYNOPSIS

#include <err.h>
void **err**(int *eval*, const char * *fmt*, ...);

void **verr**(int *eval*, const char * *fmt*, va_list *args*);

void **errx**(int *eval*, const char * *fmt*, ...);

void **verrx**(int *eval*, const char * *fmt*, va_list *args*);

void **warn**(const char * *fmt*, ...);

void **vwarn**(const char * *fmt*, va_list *args*);

void **warnx**(const char * *fmt*, ...);

void **vwarnx**(const char * *fmt*, va_list *args*);

DESCRIPTION

The *err* and *warn* family of functions display a formatted error message to the standard error output. If the *fmt* argument is not `NULL` , the formatted error message, a colon character, and a space are output. In the case of the *err* , *verr* , *warn* , and *vwarn* functions, the error message string affiliated with the current value of the global variable *errno* is output. In all cases, the output is followed by a newline character.

The *err* , *verr* , *errx* , and *verrx* functions do not return, but exit with the value of the argument *eval* .

EXAMPLES

Display the current errno information string and exit:

```
if ((p = malloc(size)) == NULL)
        err(1, NULL);
if ((fd = open(file_name, O_RDONLY, 0)) == -1)
        err(1, "%s", file_name);
```

Display an error message and exit:

```
if (tm.tm_hour < START_TIME)
        errx(1, "too early, wait until %s", start_time_string);
```

Warn of an error:

```
if ((fd = open(raw_device, O_RDONLY, 0)) == -1)
        warnx("%s: %s: trying the block device",
            raw_device, strerror(errno));
if ((fd = open(block_device, O_RDONLY, 0)) == -1)
        warn("%s", block_device);
```

HISTORY

The *err* and *warn* functions appeared in 4.4 BSD.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`strerror`(3STDC)

NAME

err, verr, verrx, warn, vwarn, warnx, vwarnx – formatted error messages

SYNOPSIS

#include <err.h>
void **err**(int *eval*, const char * *fmt*, ...);

void **verr**(int *eval*, const char * *fmt*, va_list *args*);

void **errx**(int *eval*, const char * *fmt*, ...);

void **verrx**(int *eval*, const char * *fmt*, va_list *args*);

void **warn**(const char * *fmt*, ...);

void **vwarn**(const char * *fmt*, va_list *args*);

void **warnx**(const char * *fmt*, ...);

void **vwarnx**(const char * *fmt*, va_list *args*);

DESCRIPTION

The *err* and *warn* family of functions display a formatted error message to the standard error output. If the *fmt* argument is not NULL , the formatted error message, a colon character, and a space are output. In the case of the *err* , *verr* , *warn* , and *vwarn* functions, the error message string affiliated with the current value of the global variable *errno* is output. In all cases, the output is followed by a newline character.

The *err* , *verr* , *errx* , and *verrx* functions do not return, but exit with the value of the argument *eval* .

EXAMPLES

Display the current errno information string and exit:

```
if ((p = malloc(size)) == NULL)
        err(1, NULL);
if ((fd = open(file_name, O_RDONLY, 0)) == -1)
        err(1, "%s", file_name);
```

Display an error message and exit:

```
if (tm.tm_hour < START_TIME)
        errx(1, "too early, wait until %s", start_time_string);
```

Warn of an error:

```
if ((fd = open(raw_device, O_RDONLY, 0)) == -1)
        warnx("%s: %s: trying the block device",
            raw_device, strerror(errno));
if ((fd = open(block_device, O_RDONLY, 0)) == -1)
        warn("%s", block_device);
```

HISTORY

The *err* and *warn* functions appeared in 4.4 BSD.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `strerror`(3STDC)

Index