|  |
| --- |
| **0.1 Disclaimer** |

This manual is originated from nuttx.org user guide, with supported features only. For more details, please refer to www.nuttx.org

|  |
| --- |
| **1.0 Introduction** |

This manual provides general usage information for the NuttX RTOS from the perspective of the firmware developer.

|  |
| --- |
| **1.1 Document Overview** |

This user's manual is divided into three sections plus a index:

* **Section 1.0, Introduction:** This section provides an overview of the NuttX user's manual.
* **Section 2.0, OS Interfaces**: This section details the program interfaces provided by NuttX. This section is divided into several paragraphs that describe different groups of OS interfaces:
  + Paragraph 2.1 Task Control Interfaces
  + Paragraph 2.2 Task Scheduling Interfaces
  + Paragraph 2.3 Task Control Interfaces
  + Paragraph 2.4 Named Message Queue Interfaces
  + Paragraph 2.5 Counting Semaphore Interfaces
  + Paragraph 2.6 Clocks and Timers
  + Paragraph 2.7 Signal Interfaces
  + Paragraph 2.8 Pthread Interfaces
  + Paragraph 2.9 Environment Variables
  + Paragraph 2.10 File System Interfaces
* **Section 3.0, OS Data Structures**: This section documents the data structures that are used at the NuttX interface.
  + Paragraph 3.1 Scalar Types
  + Paragraph 3.2 Hidden Interface Structures
  + Paragraph 3.3 Access to the errno Variable
  + Paragraph 3.4 User Interface Structures
* Index

|  |
| --- |
| **1.2 Intended Audience and Scope** |

The intended audience for this document are firmware developers who are implementing applications on NuttX. Specifically, this documented is limited to addressing only NuttX RTOS APIs that are available to the application developer. As such, this document does not focus on any technical details of the organization or implementation of NuttX. Those technical details are provided in the [NuttX Porting Guide](http://nuttx.org/Documentation/NuttxPortingGuide.html).

Information about configuring and building NuttX is also needed by the application developer. That information can also be found in the [NuttX Porting Guide](http://nuttx.org/Documentation/NuttxPortingGuide.html#configandbuild).

|  |
| --- |
| **2.0 OS Interfaces** |

This section describes each C-callable interface to the NuttX Operating System. The description of each interface is presented in the following format:

**Function Prototype:** The C prototype of the interface function is provided.

**Description:** The operation performed by the interface function is discussed.

**Input Parameters:** All input parameters are listed along with brief descriptions of each input parameter.

**Returned Value:** All possible values returned by the interface function are listed. Values returned as side-effects (through pointer input parameters or through global variables) will be addressed in the description of the interface function.

**Assumptions/Limitations:** Any unusual assumptions made by the interface function or any non-obvious limitations to the use of the interface function will be indicated here.

**POSIX Compatibility:** Any significant differences between the NuttX interface and its corresponding POSIX interface will be noted here.

NOTE: In order to achieve an independent name space for the NuttX interface functions, differences in function names and types are to be expected and will not be identified as differences in these paragraphs.

|  |
| --- |
| **2.1 Task Control Interfaces** |

**Tasks**. NuttX is a flat address OS. As such it does not support *processes* in the way that, say, Linux does. NuttX only supports simple threads running within the same address space. However, the programming model makes a distinction between *tasks* and *pthreads*:

* *tasks* are threads which have a degree of independence
* pthreads share some resources.

**File Descriptors and Streams**. This applies, in particular, in the area of opened file descriptors and streams. When a task is started using the interfaces in this section, it will be created with at most three open files.

If CONFIG\_DEV\_CONSOLE is defined, the first three file descriptors (corresponding to stdin, stdout, stderr) will be duplicated for the new task. Since these file descriptors are duplicated, the child task can free close them or manipulate them in any way without effecting the parent task. File-related operations (open, close, etc.) within a task will have no effect on other tasks. Since the three file descriptors are duplicated, it is also possible to perform some level of redirection.

pthreads, on the other hand, will always share file descriptors with the parent thread. In this case, file operations will have effect only all pthreads the were started from the same parent thread.

**Executing Programs within a File System**. NuttX also provides internal interfaces for the execution of separately built programs that reside in a file system. These internal interfaces are, however, non-standard and are documented with the NuttX [binary loader](http://nuttx.org/Documentation/NuttXBinfmt.html) and [NXFLAT](http://nuttx.org/Documentation/NuttXNxFlat.html#binfmt).

**Task Control Interfaces**. The following task control interfaces are provided by NuttX:

Non-standard task control interfaces inspired by VxWorks interfaces:

* 2.1.1 task\_create
* 2.1.2 task\_init
* 2.1.3 task\_activate
* 2.1.4 task\_delete
* 2.1.5 task\_restart

Non-standard extensions to VxWorks-like interfaces to support POSIX [Cancellation Points](http://www.nuttx.org/doku.php?id=wiki:nxinternal:cancellation-points).

* 2.1.6 task\_setcancelstate
* 2.1.7 task\_setcanceltype
* 2.1.8 task\_testcancel

Standard interfaces

* 2.1.9 exit
* 2.1.10 getpid

Standard vfork and exec[v|l] interfaces:

* 2.1.11 vfork
* 2.1.12 exec
* 2.1.13 execv
* 2.1.14 execl

Standard posix\_spawn interfaces:

* 2.1.14 posix\_spawn and posix\_spawnp
* 2.1.15 posix\_spawn\_file\_actions\_init
* 2.1.16 posix\_spawn\_file\_actions\_destroy
* 2.1.17 posix\_spawn\_file\_actions\_addclose
* 2.1.18 posix\_spawn\_file\_actions\_adddup2
* 2.1.19 posix\_spawn\_file\_actions\_addopen
* 2.1.20 posix\_spawnattr\_init
* 2.1.21 posix\_spawnattr\_getflags
* 2.1.22 posix\_spawnattr\_getschedparam
* 2.1.23 posix\_spawnattr\_getschedpolicy
* 2.1.24 posix\_spawnattr\_getsigmask
* 2.1.25 posix\_spawnattr\_setflags
* 2.1.26 posix\_spawnattr\_setschedparam
* 2.1.27 posix\_spawnattr\_setschedpolicy
* 2.1.28 posix\_spawnattr\_setsigmask

Non-standard task control interfaces inspired by posix\_spawn:

* 2.1.29 task\_spawn
* 2.1.30 task\_spawnattr\_getstacksize
* 2.1.31 task\_spawnattr\_setstacksize
* 2.1.32 posix\_spawn\_file\_actions\_init

**2.1.1 task\_create**

**Function Prototype:**

#include <sched.h>

int task\_create(char \*name, int priority, int stack\_size, main\_t entry, char \* const argv[]);

**Description:** This function creates and activates a new task with a specified priority and returns its system-assigned ID.

The entry address entry is the address of the "main" function of the task. This function will be called once the C environment has been set up. The specified function will be called with four arguments. Should the specified routine return, a call to exit() will automatically be made.

Note that an arbitrary number of arguments may be passed to the spawned functions.

The arguments are copied (via strdup) so that the life of the passed strings is not dependent on the life of the caller to task\_create().

The newly created task does not inherit scheduler characteristics from the parent task: The new task is started at the default system priority and with the SCHED\_FIFO scheduling policy. These characteristics may be modified after the new task has been started.

The newly created task does inherit the first three file descriptors (corresponding to stdin, stdout, and stderr) and redirection of standard I/O is supported.

**Input Parameters:**

* name. Name of the new task
* priority. Priority of the new task
* stack\_size. size (in bytes) of the stack needed
* entry. Entry point of a new task
* argv. A pointer to an array of input parameters. The array should be terminated with a NULL argv[] value. If no parameters are required, argv may be NULL.

**Returned Value:**

* Returns the non-zero task ID of the new task or ERROR if memory is insufficient or the task cannot be created ([errno](http://nuttx.org/Documentation/NuttxUserGuide.html#ErrnoAccess) is not set).

**Assumptions/Limitations:**

**POSIX Compatibility:** This is a NON-POSIX interface. VxWorks provides the following similar interface:

int taskSpawn(char \*name, int priority, int options, int stackSize, FUNCPTR entryPt,

int arg1, int arg2, int arg3, int arg4, int arg5,

int arg6, int arg7, int arg8, int arg9, int arg10);

The NuttX task\_create() differs from VxWorks' taskSpawn() in the following ways:

* Interface name
* Various differences in types of arguments
* There is no options argument.
* A variable number of parameters can be passed to a task (VxWorks supports ten).

**2.1.2 task\_init**

**Function Prototype:**

#include <sched.h>

int task\_init(struct tcb\_s \*tcb, char \*name, int priority, uint32\_t \*stack, uint32\_t stack\_size,

maint\_t entry, char \* const argv[]);

**Description:**

This function initializes a Task Control Block (TCB) in preparation for starting a new thread. It performs a subset of the functionality of task\_create() (see above).

Unlike task\_create(), task\_init() does not activate the task. This must be done by calling task\_activate().

**Input Parameters:**

* tcb. Address of the new task's TCB
* name. Name of the new task (not used)
* priority. Priority of the new task
* stack. Start of the pre-allocated stack
* stack\_size. size (in bytes) of the pre-allocated stack
* entry. Entry point of a new task
* argv. A pointer to an array of input parameters. The array should be terminated with a NULL argv[] value. If no parameters are required, argv may be NULL.

**Returned Value:**

* OK, or ERROR if the task cannot be initialized.

This function can only failure is it is unable to assign a new, unique task ID to the TCB ([errno](http://nuttx.org/Documentation/NuttxUserGuide.html#ErrnoAccess) is not set).

**Assumptions/Limitations:**

* task\_init() is provided to support internal OS functionality. It is **not recommended** for normal usage. task\_create() is the preferred mechanism to initialize and start a new task.

**POSIX Compatibility:** This is a NON-POSIX interface. VxWorks provides the following similar interface:

STATUS taskInit(WIND\_TCB \*pTcb, char \*name, int priority, int options, uint32\_t \*pStackBase, int stackSize,

FUNCPTR entryPt, int arg1, int arg2, int arg3, int arg4, int arg5,

int arg6, int arg7, int arg8, int arg9, int arg10);

The NuttX task\_init() differs from VxWorks' taskInit() in the following ways:

* Interface name
* Various differences in types or arguments
* There is no options argument.
* A variable number of parameters can be passed to a task (VxWorks supports ten).

**2.1.3 task\_activate**

**Function Prototype:**

#include <sched.h>

int task\_activate(struct tcb\_s \*tcb);

**Description:** This function activates tasks created by task\_init(). Without activation, a task is ineligible for execution by the scheduler.

**Input Parameters:**

* tcb. The TCB for the task for the task (same as the task\_init argument).

**Returned Value:**

* OK, or ERROR if the task cannot be activated ([errno](http://nuttx.org/Documentation/NuttxUserGuide.html#ErrnoAccess) is not set).

**Assumptions/Limitations:**

* task\_activate() is provided to support internal OS functionality. It is **not recommended** for normal usage. task\_create() is the preferred mechanism to initialize and start a new task.

**POSIX Compatibility:** This is a NON-POSIX interface. VxWorks provides the following similar interface:

STATUS taskActivate(int tid);

The NuttX task\_activate() differs from VxWorks' taskActivate() in the following ways:

* Function name
* With VxWork's taskActivate, the pid argument is supposed to be the pointer to the WIND\_TCB cast to an integer.

**2.1.4 task\_delete**

**Function Prototype:**

#include <sched.h>

int task\_delete(pid\_t pid);

**Description:** This function causes a specified task to cease to exist. Its stack and TCB will be deallocated. This function is the companion to task\_create(). This is the version of the function exposed to the user; it is simply a wrapper around the internal, task\_terminate() function.

The logic in this function only deletes non-running tasks. If the  pid parameter refers to to the currently runing task, then processing is redirected to exit(). This can only happen if a task calls task\_delete() in order to delete itself.

This function obeys the semantics of pthread cancellation: task deletion is deferred if cancellation is disabled or if deferred cancellation is supported (with [Cancellation Points](http://www.nuttx.org/doku.php?id=wiki:nxinternal:cancellation-points) enabled).

**Input Parameters:**

* pid. The task ID of the task to delete. An ID of zero signifies the calling task. Any attempt by the calling task will be automatically re-directed to exit().

**Returned Value:**

* OK, or ERROR if the task cannot be deleted. The errno is set to indicate the nature of the failure. This function can fail, for example, if the provided pid does not correspond to a currently executing task.

**Assumptions/Limitations:**

task\_delete() must be used with caution: If the task holds resources (for example, allocated memory or semaphores needed by other tasks), then task\_delete() can strand those resources.

**POSIX Compatibility:** This is a NON-POSIX interface. VxWorks provides the following similar interface:

STATUS taskDelete(int tid);

The NuttX task\_delete() differs from VxWorks' taskDelete() in the following ways:

* No support is provided for calling the tasks deletion routines (because the VxWorks taskDeleteHookAdd() is not supported). However, if atexit() or on\_exit support is enabled, those will be called when the task deleted.
* Deletion of self is supported, but only because task\_delete() will re-direct processing to exit().

**2.1.5 task\_restart**

**Function Prototype:**

#include <sched.h>

int task\_restart(pid\_t pid);

**Description:** This function "restarts" a task. The task is first terminated and then reinitialized with same ID, priority, original entry point, stack size, and parameters it had when it was first started.

**NOTES:**

1. The normal task exit clean up is not performed. For example, file descriptors are not closed; any files opened prior to the restart will remain opened after the task is restarted.
2. Memory allocated by the task before it was restart is not freed. A task that is subject to being restart must be designed in such a way as to avoid memory leaks.
3. Initialized data is not reset. All global or static data is left in the same state as when the task was terminated. This *feature* may be used by restart task to detect that it has been restarted, for example.

**Input Parameters:**

* pid. The task ID of the task to delete. An ID of zero would signify the calling task (However, support for a task to restart itself has not been implemented).

**Returned Value:**

* OK, or ERROR if the task ID is invalid or the task could not be restarted. This function can fail if: (1) A pid of zero or the pid of the calling task is provided (functionality not implemented) (2) The pid is not associated with any task known to the system.

**Assumptions/Limitations:**

**POSIX Compatibility:** This is a NON-POSIX interface. VxWorks provides the following similar interface:

STATUS taskRestart (int tid);

The NuttX task\_restart() differs from VxWorks' taskRestart() in the following ways:

* Restart of the currently running task is not supported by NuttX.
* The VxWorks description says that the ID, priority, etc. take the value that they had when the task was *terminated*.

**2.1.6 task\_setcancelstate**

**Function Prototype:**

#include <sched.h>

int task\_setcancelstate(int state, int \*oldstate);

**Description:** The task\_setcancelstate() function atomically sets both the calling task's cancelability state to the indicated state and returns the previous cancelability state at the location referenced by oldstate. Legal values for state are TASK\_CANCEL\_ENABLE and TASK\_CANCEL\_DISABLE.

Any pending thread cancellation may occur at the time that the cancellation state is set to TASK\_CANCEL\_ENABLE.

The cancelability state and type of any newly created tasks are TASK\_CANCEL\_ENABLE and TASK\_CANCEL\_DEFERRED respectively.

**Input Parameters:**

* state New cancellation state. One of PTHREAD\_CANCEL\_ENABLE or PTHREAD\_CANCEL\_DISABLE.
* oldstate. Location to return the previous cancellation state.

**Returned Value:**

Zero (OK) on success; ERROR is returned on any failure with the errno value set appropriately:

* ESRCH. No thread could be found corresponding to that specified by the given thread ID.

**Assumptions/Limitations:**

**POSIX Compatibility:** This is a non-standard interface. It extends the functionality of pthread\_setcancelstate() to tasks and supports use of task\_delete().

**2.1.7 task\_setcanceltype**

**Function Prototype:**

#include <sched.h>

int task\_setcanceltype(int type, FAR int \*oldtype);

**Description:** The task\_setcanceltype() function atomically both sets the calling task's cancelability type to the indicated type and returns the previous cancelability type at the location referenced by oldtype. Legal values for type are TASK\_CANCEL\_DEFERRED and TASK\_CANCEL\_ASYNCHRONOUS.

The cancelability state and type of any newly created tasks are TASK\_CANCEL\_ENABLE and TASK\_CANCEL\_DEFERRED respectively.

**Input Parameters:**

* type New cancellation state. One of PTHREAD\_CANCEL\_DEFERRED or PTHREAD\_CANCEL\_ASYNCHRONOUS.
* oldtype. Location to return the previous cancellation type.

**Returned Value:**

Zero (OK) on success; ERROR is returned on any failure with the errno value set appropriately:

* ESRCH. No thread could be found corresponding to that specified by the given thread ID.

**POSIX Compatibility:** This is a non-standard interface. It extends the functionality of pthread\_setcanceltype() to tasks and supports use of task\_delete().

**2.1.8 task\_testcancel**

**Function Prototype:**

#include <sched.h>

void task\_testcancel(void);

**Description:**

The task\_testcancel() function creates a [Cancellation Point](http://www.nuttx.org/doku.php?id=wiki:nxinternal:cancellation-points) in the calling task. The task\_testcancel() function has no effect if cancelability is disabled.

**Input Parameters:** None

**Returned Value:** None

**POSIX Compatibility:** This is a non-standard interface. It extends the functionality of pthread\_testcancel() to tasks and supports use of task\_delete().

**2.1.9 exit**

**Function Prototype:**

#include <sched.h>

void exit(int code);

#include <nuttx/unistd.h>

void \_exit(int code);

**Description:** This function causes the calling task to cease to exist -- its stack and TCB will be deallocated. exit differs from \_exit in that it flushes streams, closes file descriptors and will execute any function registered with atexit() or on\_exit().

**Input Parameters:**

* code. (ignored)

**Returned Value:** None.

**Assumptions/Limitations:**

**POSIX Compatibility:** This is equivalent to the ANSI interface:

void exit(int code);

And the UNIX interface:

void \_exit(int code);

The NuttX exit() differs from ANSI exit() in the following ways:

* The code parameter is ignored.

**2.1.10 getpid**

**Function Prototype:**

#include <unistd.h>

pid\_t getpid(void);

**Description:** This function returns the task ID of the calling task. The task ID will be invalid if called at the interrupt level.

**Input Parameters:** None.

**Returned Value:**

* The task ID of the calling task.

**Assumptions/Limitations:**

**POSIX Compatibility:** Compatible with the POSIX interface of the same name.

**2.1.11 vfork**

**Function Prototype:**

#include <unistd.h>

pid\_t vfork(void);

**Description:** The vfork() function has the same effect as fork(), except that the behavior is undefined if the process created by vfork() either modifies any data other than a variable of type pid\_t used to store the return value from vfork(), or returns from the function in which vfork() was called, or calls any other function before successfully calling \_exit() or one of the exec family of functions.

**NOTE:** vfork() is not an independent NuttX feature, but is implemented in architecture-specific logic (using only helper functions from the NuttX core logic). As a result, vfork() may not be available on all architectures.

**Input Parameters:** None.

**Returned Value:** Upon successful completion, vfork() returns 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, -1 is returned to the parent, no child process is created, and errno is set to indicate the error.

**Assumptions/Limitations:**

**POSIX Compatibility:** Compatible with the BSD/Linux interface of the same name. POSIX marks this interface as Obsolete.

**2.1.12 exec**

**Function Prototype:**

#include <nuttx/binfmt/binfmt.h>

#ifndef CONFIG\_BUILD\_KERNEL

int exec(FAR const char \*filename, FAR char \* const \*argv,

FAR const struct symtab\_s \*exports, int nexports);

#endif

**Description:** This non-standard, NuttX function is similar to execv() and posix\_spawn() but differs in the following ways;

* Unlike execv() and posix\_spawn() this function accepts symbol table information as input parameters. This means that the symbol table used to link the application prior to execution is provided by the caller, not by the system.
* Unlike execv(), this function always returns.

This non-standard interface is included as a official NuttX API only because it is needed in certain build modes: exec() is probably the only want to load programs in the PROTECTED mode. Other file execution APIs rely on a symbol table provided by the OS. In the PROTECTED build mode, the OS cannot provide any meaningful symbolic information for execution of code in the user-space blob so that is the exec() function is really needed in that build case

The interface is available in the FLAT build mode although it is not really necessary in that case. It is currently used by some example code under the apps/ that that generate their own symbol tables for linking test programs. So althought it is not necessary, it can still be useful.

The interface would be completely useless and will not be supported in the KERNEL build mode where the contrary is true: An application process cannot provide any meaning symbolic information for use in linking a different process.

**NOTE**: This function is flawed and useless without CONFIG\_SCHED\_ONEXIT and CONFIG\_SCHED\_HAVE\_PARENT because without those features there is then no mechanism to unload the module once it exits.

**Input Parameters:**

* filename: The path to the program to be executed. If CONFIG\_BINFMT\_EXEPATH is defined in the configuration, then this may be a relative path from the current working directory. Otherwise, path must be the absolute path to the program.
* argv: A pointer to an array of string arguments. The end of the array is indicated with a NULL entry.
* exports: The address of the start of the caller-provided symbol table. This symbol table contains the addresses of symbols exported by the caller and made available for linking the module into the system.
* nexports: The number of symbols in the exports table.

**Returned Value:** Zero (OK) is returned on success; On any failure, exec() will return -1 (ERROR) and will set the errno value appropriately.

**Assumptions/Limitations:**

**POSIX Compatibility:** This is a non-standard inteface unique to NuttX. Motivation for inclusion of this non-standard interface in certain build modes is discussed above.

**2.1.13 execv**

**Function Prototype:**

#include <unistd.h>

#ifdef CONFIG\_LIBC\_EXECFUNCS

int execv(FAR const char \*path, FAR char \*const argv[]);

#endif

**Description:** The standard exec family of functions will replace the current process image with a new process image. The new image will be constructed from a regular, executable file called the new process image file. There will be no return from a successful exec, because the calling process image is overlaid by the new process image.

Simplified execl() and execv() functions are provided by NuttX for compatibility. NuttX is a tiny embedded RTOS that does not support processes and hence the concept of overlaying a tasks process image with a new process image does not make any sense. In NuttX, these functions are wrapper functions that:

1. Call the non-standard binfmt function exec(), and then
2. exit(0).

Note the inefficiency when execv() or execl() is called in the normal, two-step process: (1) first call vfork() to create a new thread, then (2) call execv() or execl() to replace the new thread with a program from the file system. Since the new thread will be terminated by the execv() or execl() call, it really served no purpose other than to support POSIX compatibility.

The non-standard binfmt function exec() needs to have (1) a symbol table that provides the list of symbols exported by the base code, and (2) the number of symbols in that table. This information is currently provided to exec() from execv() or execl() via NuttX configuration settings:

* CONFIG\_LIBC\_EXECFUNCS: Enable execv() and execl() support
* CONFIG\_EXECFUNCS\_SYMTAB\_ARRAY: Name of the ymbol table used by execv() or execl().
* CONFIG\_EXECFUNCS\_NSYMBOLS\_VAR: Name of the int variable holding the number of symbols in the symbol table

As a result of the above, the current implementations of execl() and execv() suffer from some incompatibilities that may or may not be addressed in a future version of NuttX. Other than just being an inefficient use of MCU resource, the most serious of these is that the exec'ed task will not have the same task ID as the vfork'ed function. So the parent function cannot know the ID of the exec'ed task.

**Input Parameters:**

* path: The path to the program to be executed. If CONFIG\_BINFMT\_EXEPATH is defined in the configuration, then this may be a relative path from the current working directory. Otherwise, path must be the absolute path to the program.

argv: A pointer to an array of string arguments. The end of the array is indicated with a NULL entry.

**Returned Value:** This function does not return on success. On failure, it will return -1 (ERROR) and will set the errno value appropriately.

**Assumptions/Limitations:**

**POSIX Compatibility:** Similar with the POSIX interface of the same name. There are, however, several compatibility issues as detailed in the description above.

**2.1.14 execl**

**Function Prototype:**

#include <unistd.h>

#ifdef CONFIG\_LIBC\_EXECFUNCS

int execl(FAR const char \*path, ...);

#endif

**Description:** execl() is functionally equivalent to execv(), differing only in the form of its input parameters. See the decription of execv() for additional information.

**Input Parameters:**

* path: The path to the program to be executed. If CONFIG\_BINFMT\_EXEPATH is defined in the configuration, then this may be a relative path from the current working directory. Otherwise, path must be the absolute path to the program.

...: A list of the string arguments to be recevied by the program. Zero indicates the end of the list.

**Returned Value:** This function does not return on success. On failure, it will return -1 (ERROR) and will set the errno value appropriately.

**Assumptions/Limitations:**

**POSIX Compatibility:** Similar with the POSIX interface of the same name. There are, however, several compatibility issues as detailed in the description of execv().

**2.1.14 posix\_spawn and posix\_spawnp**

**Function Prototype:**

#include <spawn.h>

int posix\_spawn(FAR pid\_t \*pid, FAR const char \*path,

FAR const posix\_spawn\_file\_actions\_t \*file\_actions,

FAR const posix\_spawnattr\_t \*attr,

FAR char \*const argv[], FAR char \*const envp[]);

int posix\_spawnp(FAR pid\_t \*pid, FAR const char \*file,

FAR const posix\_spawn\_file\_actions\_t \*file\_actions,

FAR const posix\_spawnattr\_t \*attr,

FAR char \*const argv[], FAR char \*const envp[]);

**Description:** The posix\_spawn() and posix\_spawnp() functions will create a new, child task, constructed from a regular executable file.

**Input Parameters:**

* pid: Upon successful completion, posix\_spawn() and posix\_spawnp() will return the task ID of the child task to the parent task, in the variable pointed to by a non-NULL pid argument. If the pid argument is a null pointer, the process ID of the child is not returned to the caller.
* path or file: The path argument to posix\_spawn() is the absolute path that identifies the file to execute. The file argument to posix\_spawnp() may also be a relative path and will be used to construct a pathname that identifies the file to execute. In the case of a relative path, the path prefix for the file will be obtained by a search of the directories passed as the environment variable PATH.

NOTE: NuttX provides only one implementation: If CONFIG\_BINFMT\_EXEPATH is defined, then only posix\_spawnp() behavior is supported; otherwise, only posix\_spawn behavior is supported.

* file\_actions: If file\_actions is a null pointer, then file descriptors open in the calling process will remain open in the child process (unless CONFIG\_FDCLONE\_STDIO is defined). If file\_actions is not NULL, then the file descriptors open in the child process will be those open in the calling process as modified by the spawn file actions object pointed to by file\_actions.
* attr: If the value of the attr parameter is NULL, the all default values for the POSIX spawn attributes will be used. Otherwise, the attributes will be set according to the spawn flags. The posix\_spawnattr\_t spawn attributes object type is defined in spawn.h. It will contains these attributes, not all of which are supported by NuttX:
  + POSIX\_SPAWN\_SETPGROUP: Setting of the new task's process group is not supported. NuttX does not support process groups.
  + POSIX\_SPAWN\_SETSCHEDPARAM: Set new tasks priority to the sched\_param value.
  + POSIX\_SPAWN\_SETSCHEDULER: Set the new task's scheduler policy to the sched\_policy value.
  + POSIX\_SPAWN\_RESETIDS Resetting of the effective user ID of the child process is not supported. NuttX does not support effective user IDs.
  + POSIX\_SPAWN\_SETSIGMASK: Set the new task's signal mask.
  + POSIX\_SPAWN\_SETSIGDEF: Resetting signal default actions is not supported. NuttX does not support default signal actions.
* argv: argv[] is the argument list for the new task.  argv[] is an array of pointers to null-terminated strings. The list is terminated with a null pointer.
* envp: The envp[] argument is not used by NuttX and may be NULL. In standard implementations, envp[] is an array of character pointers to null-terminated strings that provide the environment for the new process image. The environment array is terminated by a null pointer. In NuttX, the envp[] argument is ignored and the new task will inherit the environment of the parent task unconditionally.

**Returned Value:** posix\_spawn() and posix\_spawnp() will return zero on success. Otherwise, an error number will be returned as the function return value to indicate the error:

* EINVAL: The value specified by file\_actions or attr is invalid.
* Any errors that might have been return if vfork() and excec[l|v]() had been called.

**Assumptions/Limitations:**

* NuttX provides only posix\_spawn() or posix\_spawnp() behavior depending upon the setting of CONFIG\_BINFMT\_EXEPATH: If CONFIG\_BINFMT\_EXEPATH is defined, then only posix\_spawnp() behavior is supported; otherwise, only posix\_spawn() behavior is supported.
* The envp argument is not used and the environ variable is not altered (NuttX does not support the environ variable).
* Process groups are not supported (See POSIX\_SPAWN\_SETPGROUP above).
* Effective user IDs are not supported (See POSIX\_SPAWN\_RESETIDS above).
* Signal default actions cannot be modified in the newly task executed because NuttX does not support default signal actions (See POSIX\_SPAWN\_SETSIGDEF).

**POSIX Compatibility:** The value of the argv[0] received by the child task is assigned by NuttX. For the caller of posix\_spawn(), the provided argv[0] will correspond to argv[1] received by the new task.

**2.1.15 posix\_spawn\_file\_actions\_init**

**Function Prototype:**

#include <spawn.h>

int posix\_spawn\_file\_actions\_init(FAR posix\_spawn\_file\_actions\_t \*file\_actions);

**Description:** The posix\_spawn\_file\_actions\_init() function initializes the object referenced by file\_actions to an empty set of file actions for subsequent use in a call to posix\_spawn() or posix\_spawnp().

**Input Parameters:**

* file\_actions: The address of the posix\_spawn\_file\_actions\_t to be initialized.

**Returned Value:** On success, this function returns 0; on failure it will return an error number from <errno.h>.

**2.1.16 posix\_spawn\_file\_actions\_destroy**

**Function Prototype:**

#include <spawn.h>

int posix\_spawn\_file\_actions\_destroy(FAR posix\_spawn\_file\_actions\_t \*file\_actions);

**Description:** The posix\_spawn\_file\_actions\_destroy() function destroys the object referenced by file\_actions which was previously initialized by posix\_spawn\_file\_actions\_init(), returning any resources obtained at the time of initialization to the system for subsequent reuse. A posix\_spawn\_file\_actions\_t may be reinitialized after having been destroyed, but must not be reused after destruction, unless it has been reinitialized.

**Input Parameters:**

* file\_actions: The address of the posix\_spawn\_file\_actions\_t to be destroyed.

**Returned Value:** On success, this function returns 0; on failure it will return an error number from <errno.h>

**2.1.17 posix\_spawn\_file\_actions\_addclose**

**Function Prototype:**

#include <spawn.h>

int posix\_spawn\_file\_actions\_addclose(FAR posix\_spawn\_file\_actions\_t \*file\_actions, int fd);

**Description:** The posix\_spawn\_file\_actions\_addclose() function adds a *close* operation to the list of operations associated with the object referenced by file\_actions, for subsequent use in a call to posix\_spawn() or posix\_spawnp(). The descriptor referred to by fdis closed as if close() had been called on it prior to the new child process starting execution.

**Input Parameters:**

* file\_actions: The address of the posix\_spawn\_file\_actions\_t object to which the *close* operation will be appended.
* fd: The file descriptor to be closed.

**Returned Value:** On success, this function returns 0; on failure it will return an error number from <errno.h>

**2.1.18 posix\_spawn\_file\_actions\_adddup2**

**Function Prototype:**

#include <spawn.h>

int posix\_spawn\_file\_actions\_adddup2(FAR posix\_spawn\_file\_actions\_t \*file\_actions, int fd1, int fd2);

**Description:** The posix\_spawn\_file\_actions\_adddup2() function adds a *dup2* operation to the list of operations associated with the object referenced by file\_actions, for subsequent use in a call to posix\_spawn() or posix\_spawnp(). The descriptor referred to by fd2 is created as if dup2() had been called on fd1 prior to the new child process starting execution.

**Input Parameters:**

* file\_actions: The address of the posix\_spawn\_file\_actions\_t object to which the *dup2* operation will be appended.
* fd1: The file descriptor to be be duplicated. The first file descriptor to be argument to dup2().
* fd2: The file descriptor to be be created. The second file descriptor to be argument to dup2().

**Returned Value:** On success, this function returns 0; on failure it will return an error number from <errno.h>

**2.1.19 posix\_spawn\_file\_actions\_addopen**

**Function Prototype:**

#include <spawn.h>

int posix\_spawn\_file\_actions\_addopen(FAR posix\_spawn\_file\_actions\_t \*file\_actions,

int fd, FAR const char \*path, int oflags, mode\_t mode);

**Description:** The posix\_spawn\_file\_actions\_addopen() function adds an *open* operation to the list of operations associated with the object referenced by file\_actions, for subsequent use in a call to posix\_spawn() or posix\_spawnp(). The descriptor referred to by fdis opened using the path, oflag, and mode arguments as if open() had been called on it prior to the new child process starting execution. The string path is copied by the posix\_spawn\_file\_actions\_addopen() function during this process, so storage need not be persistent in the caller.

**Input Parameters:**

* file\_actions: The address of the posix\_spawn\_file\_actions\_t object to which the *open* operation will be appended.
* fd: The file descriptor to be opened.
* path: The path to be opened.
* oflags: Open flags.
* mode: File creation mode/

**Returned Value:** On success, this function returns 0; on failure it will return an error number from <errno.h>

**2.1.20 posix\_spawnattr\_init**

**Function Prototype:**

#include <spawn.h>

int posix\_spawnattr\_init(FAR posix\_spawnattr\_t \*attr);

**Description:** The posix\_spawnattr\_init() function initializes the object referenced by attr, to an empty set of spawn attributes for subsequent use in a call to  posix\_spawn() or  posix\_spawnp().

Then the spawn attributes are no longer needed, they should be destroyed by calling posix\_spawnattr\_destroyed(). In NuttX, however, posix\_spawnattr\_destroyed() is just stub:

#define posix\_spawnattr\_destroy(attr) (0)

For portability, the convention of calling posix\_spawnattr\_destroyed() when the attributes are not longer needed should still be followed.

**Input Parameters:**

* attr: The address of the spawn attributes to be initialized.

**Returned Value:** On success, this function returns 0; on failure it will return an error number from <errno.h>

**2.1.21 posix\_spawnattr\_getflags**

**Function Prototype:**

#include <spawn.h>

int posix\_spawnattr\_getflags(FAR const posix\_spawnattr\_t \*attr, FAR short \*flags);

**Description:** The posix\_spawnattr\_getflags() function will obtain the value of the *spawn-flags* attribute from the attributes object referenced by attr.

**Input Parameters:**

* attr: The address spawn attributes to be queried.
* flags: The location to return the spawn flags

**Returned Value:** On success, this function returns 0; on failure it will return an error number from <errno.h>

**2.1.22 posix\_spawnattr\_getschedparam**

**Function Prototype:**

#include <spawn.h>

int posix\_spawnattr\_getschedparam(FAR const posix\_spawnattr\_t \*attr, FAR struct sched\_param \*param);

**Description:** The posix\_spawnattr\_getschedparam() function will obtain the value of the *spawn-schedparam* attribute from the attributes object referenced by attr.

**Input Parameters:**

* attr: The address spawn attributes to be queried.
* param: The location to return the *spawn-schedparam* value.

**Returned Value:** On success, this function returns 0; on failure it will return an error number from <errno.h>

**2.1.23 posix\_spawnattr\_getschedpolicy**

**Function Prototype:**

#include <spawn.h>

int posix\_spawnattr\_getschedpolicy(FAR const posix\_spawnattr\_t \*attr, FAR int \*policy);

**Description:** The posix\_spawnattr\_getschedpolicy() function will obtain the value of the *spawn-schedpolicy* attribute from the attributes object referenced by attr.

**Input Parameters:**

* attr: The address spawn attributes to be queried.
* policy: The location to return the *spawn-schedpolicy* value.

**Returned Value:** On success, this function returns 0; on failure it will return an error number from <errno.h>

**2.1.24 posix\_spawnattr\_getsigmask**

**Function Prototype:**

#include <spawn.h>

#ifndef CONFIG\_DISABLE\_SIGNALS

int posix\_spawnattr\_getsigmask(FAR const posix\_spawnattr\_t \*attr, FAR sigset\_t \*sigmask);

#endif

**Description:** The posix\_spawnattr\_getsigdefault() function will obtain the value of the *spawn-sigmask* attribute from the attributes object referenced by attr.

**Input Parameters:**

* attr: The address spawn attributes to be queried.
* sigmask: The location to return the *spawn-sigmask* value.

**Returned Value:** On success, this function returns 0; on failure it will return an error number from <errno.h>

**2.1.25 posix\_spawnattr\_setflags**

**Function Prototype:**

#include <spawn.h>

int posix\_spawnattr\_setflags(FAR posix\_spawnattr\_t \*attr, short flags);

**Description:** The posix\_spawnattr\_setflags() function will set the *spawn-flags* attribute in an initialized attributes object referenced by attr.

**Input Parameters:**

* attr: The address spawn attributes to be used.
* flags: The new value of the *spawn-flags* attribute.

**Returned Value:** On success, this function returns 0; on failure it will return an error number from <errno.h>

**2.1.26 posix\_spawnattr\_setschedparam**

**Function Prototype:**

#include <spawn.h>

int posix\_spawnattr\_setschedparam(FAR posix\_spawnattr\_t \*attr, FAR const struct sched\_param \*param);

**Description:** The posix\_spawnattr\_setschedparam() function will set the *spawn-schedparam* attribute in an initialized attributes object referenced by attr.

**Input Parameters:**

* attr: The address spawn attributes to be used.
* param: The new value of the *spawn-schedparam* attribute.

**Returned Value:** On success, this function returns 0; on failure it will return an error number from <errno.h>

**2.1.27 posix\_spawnattr\_setschedpolicy**

**Function Prototype:**

#include <spawn.h>

int posix\_spawnattr\_setschedpolicy(FAR posix\_spawnattr\_t \*attr, int policy);

**Description:** The posix\_spawnattr\_setschedpolicy() function will set the *spawn-schedpolicy* attribute in an initialized attributes object referenced by attr.

**Input Parameters:**

* attr: The address spawn attributes to be used.
* policy: The new value of the *spawn-schedpolicy* attribute.

**Returned Value:** On success, this function returns 0; on failure it will return an error number from <errno.h>

**2.1.28 posix\_spawnattr\_setsigmask**

**Function Prototype:**

#include <spawn.h>

#ifndef CONFIG\_DISABLE\_SIGNALS

int posix\_spawnattr\_setsigmask(FAR posix\_spawnattr\_t \*attr, FAR const sigset\_t \*sigmask);

#endif

**Description:** The posix\_spawnattr\_setsigmask() function will set the *spawn-sigmask* attribute in an initialized attributes object referenced by attr.

**Input Parameters:**

* attr: The address spawn attributes to be used.
* sigmask: The new value of the *spawn-sigmask* attribute.

**Returned Value:** On success, this function returns 0; on failure it will return an error number from <errno.h>

**2.1.29 task\_spawn**

**Function Prototype:**

#include <spawn.h>

int task\_spawn(FAR pid\_t \*pid, FAR const char \*name, main\_t entry,

FAR const posix\_spawn\_file\_actions\_t \*file\_actions,

FAR const posix\_spawnattr\_t \*attr,

FAR char \*const argv[], FAR char \*const envp[]);

**Description:** The task\_spawn() function will create a new, child task, where the entry point to the task is an address in memory.

* pid: Upon successful completion, task\_spawn() will return the task ID of the child task to the parent task, in the variable pointed to by a non-NULL pid argument. If the pid argument is a null pointer, the process ID of the child is not returned to the caller.
* name: The name to assign to the child task.
* entry: The child task's entry point (an address in memory).
* file\_actions: If file\_actions is a null pointer, then file descriptors open in the calling process will remain open in the child process (unless CONFIG\_FDCLONE\_STDIO is defined). If file\_actions is not NULL, then the file descriptors open in the child process will be those open in the calling process as modified by the spawn file actions object pointed to by file\_actions.
* attr: If the value of the attr parameter is NULL, the all default values for the POSIX spawn attributes will be used. Otherwise, the attributes will be set according to the spawn flags. The posix\_spawnattr\_t spawn attributes object type is defined in spawn.h. It will contains these attributes, not all of which are supported by NuttX:
  + POSIX\_SPAWN\_SETPGROUP: Setting of the new task's process group is not supported. NuttX does not support process groups.
  + POSIX\_SPAWN\_SETSCHEDPARAM: Set new tasks priority to the sched\_param value.
  + POSIX\_SPAWN\_SETSCHEDULER: Set the new task's scheduler policy to the sched\_policy value.
  + POSIX\_SPAWN\_RESETIDS Resetting of the effective user ID of the child process is not supported. NuttX does not support effective user IDs.
  + POSIX\_SPAWN\_SETSIGMASK: Set the new task's signal mask.
  + POSIX\_SPAWN\_SETSIGDEF: Resetting signal default actions is not supported. NuttX does not support default signal actions.

And the non-standard:

* + TASK\_SPAWN\_SETSTACKSIZE: Set the stack size for the new task.
* argv: argv[] is the argument list for the new task.  argv[] is an array of pointers to null-terminated strings. The list is terminated with a null pointer.
* envp: The envp[] argument is not used by NuttX and may be NULL.

**Returned Value:** task\_spawn() will return zero on success. Otherwise, an error number will be returned as the function return value to indicate the error:

**POSIX Compatibility:** This is a non-standard interface inspired by posix\_spawn().

**2.1.30 task\_spawnattr\_getstacksize**

**Function Prototype:**

#include <spawn.h>

int task\_spawnattr\_getstacksize(FAR const posix\_spawnattr\_t \*attr, FAR size\_t \*stacksize);

**Description:** The task\_spawnattr\_getstacksize() function will obtain the value of the *spawn-stacksize* attribute from the attributes object referenced by attr.

**Input Parameters:**

* attr: The address spawn attributes to be queried.
* policy: The location to return the *spawn-stacksize* value.

**Returned Value:** On success, this function returns 0; on failure it will return an error number from <errno.h>

**2.1.31 task\_spawnattr\_setstacksize**

**Function Prototype:**

#include <spawn.h>

int task\_spawnattr\_setstacksize(FAR posix\_spawnattr\_t \*attr, size\_t stacksize);

**Description:** The task\_spawnattr\_setstacksize() function will set the *spawn-stacksize* attribute in an initialized attributes object referenced by attr.

**Input Parameters:**

* attr: The address spawn attributes to be used.
* policy: The new value of the *spawn-stacksize* attribute.

**Returned Value:** On success, this function returns 0; on failure it will return an error number from <errno.h>

**2.1.32 posix\_spawn\_file\_actions\_init**

**Function Prototype:**

#include <spawn.h>

int posix\_spawn\_file\_actions\_init(FAR posix\_spawn\_file\_actions\_t \*file\_actions);

**Description:** The posix\_spawn\_file\_actions\_init() function initializes the object referenced by file\_actions to an empty set of file actions for subsequent use in a call to posix\_spawn() or posix\_spawnp().

**Input Parameters:**

* file\_actions: The address of the posix\_spawn\_file\_actions\_t to be initialized.

**Returned Value:** On success, this function returns 0; on failure it will return an error number from <errno.h>.

|  |
| --- |
| **2.2 Task Scheduling Interfaces** |

By default, NuttX performs strict priority scheduling: Tasks of higher priority have exclusive access to the CPU until they become blocked. At that time, the CPU is available to tasks of lower priority. Tasks of equal priority are scheduled FIFO.

Optionally, a Nuttx task or thread can be configured with round-robin or *sporadic* scheduler. The round-robin is similar to priority scheduling *except* that tasks with equal priority and share CPU time via *time-slicing*. The time-slice interval is a constant determined by the configuration setting CONFIG\_RR\_INTERVAL to a positive, non-zero value. Sporadic scheduling scheduling is more complex, varying the priority of a thread over a *replenishment* period. Support for sporadic scheduling is enabled by the configuration option CONFIG\_SCHED\_SPORADIC.

The OS interfaces described in the following paragraphs provide a POSIX- compliant interface to the NuttX scheduler:

* 2.2.1 sched\_setparam
* 2.2.2 sched\_getparam
* 2.2.3 sched\_setscheduler
* 2.2.4 sched\_getscheduler
* 2.2.5 sched\_yield
* 2.2.6 sched\_get\_priority\_max
* 2.2.7 sched\_get\_priority\_min
* 2.2.8 sched\_get\_rr\_interval

**2.2.1 sched\_setparam**

**Function Prototype:**

#include <sched.h>

int sched\_setparam(pid\_t pid, const struct sched\_param \*param);

**Description:** This function sets the priority of the task specified by pid input parameter.

NOTE: Setting a task's priority to the same value has the similar effect to sched\_yield(): The task will be moved to after all other tasks with the same priority.

**Input Parameters:**

* pid. The task ID of the task. If pid is zero, the priority of the calling task is set.
* param. A structure whose member sched\_priority is the integer priority. The range of valid priority numbers is from SCHED\_PRIORITY\_MIN through SCHED\_PRIORITY\_MAX.

**Returned Value:** On success, sched\_setparam() returns 0 (OK). On error, -1 (ERROR) is returned, and errno is set appropriately.

* EINVAL. The parameter param is invalid or does not make sense for the current scheduling policy.
* EPERM. The calling task does not have appropriate privileges.
* ESRCH. The task whose ID is pid could not be found.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name. Differences from the full POSIX implementation include:

* The range of priority values for the POSIX call is 0 to 255.

**2.2.2 sched\_getparam**

**Function Prototype:**

#include <sched.h>

int sched\_getparam (pid\_t pid, struct sched\_param \*param);

**Description:** This function gets the scheduling priority of the task specified by pid.

**Input Parameters:**

* pid. The task ID of the task. If pid is zero, the priority of the calling task is returned.
* param. A structure whose member sched\_priority is the integer priority. The task's priority is copied to the sched\_priorityelement of this structure.

**Returned Value:**

* 0 (OK) if successful, otherwise -1 (ERROR).

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.2.3 sched\_setscheduler**

**Function Prototype:**

#include <sched.h>

int sched\_setscheduler (pid\_t pid, int policy, const struct sched\_param \*param);

**Description:** sched\_setscheduler() sets both the scheduling policyand the priority for the task identified by pid. If pid equals zero, the scheduler of the calling thread will be set. The parameter param holds the priority of the thread under the new policy.

**Input Parameters:**

* pid. The task ID of the task. If pid is zero, the priority of the calling task is set.
* policy. Scheduling policy requested (either SCHED\_FIFO or SCHED\_RR).
* param. A structure whose member sched\_priority is the integer priority. The range of valid priority numbers is from SCHED\_PRIORITY\_MIN through SCHED\_PRIORITY\_MAX.

**Returned Value:** On success, sched\_setscheduler() returns OK (zero). On error, ERROR (-1) is returned, and errno is set appropriately:

* EINVAL: The scheduling policy is not one of the recognized policies.
* ESRCH: The task whose ID is pid could not be found.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.2.4 sched\_getscheduler**

**Function Prototype:**

#include <sched.h>

int sched\_getscheduler (pid\_t pid);

**Description:** sched\_getscheduler() returns the scheduling policy currently applied to the task identified by pid. If pid equals zero, the policy of the calling process will be retrieved.

**Input Parameters:**

* pid. The task ID of the task to query. If pid is zero, the calling task is queried.

**Returned Value:**

* On success, sched\_getscheduler() returns the policy for the task (either SCHED\_FIFO or SCHED\_RR). On error,  ERROR (-1) is returned, and errno is set appropriately:
  + ESRCH: The task whose ID is pid could not be found.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.2.5 sched\_yield**

**Function Prototype:**

#include <sched.h>

int sched\_yield(void);

**Description:** This function forces the calling task to give up the CPU (only to other tasks at the same priority).

**Input Parameters:** None.

**Returned Value:**

* 0 (OK) or -1 (ERROR)

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.2.6 sched\_get\_priority\_max**

**Function Prototype:**

#include <sched.h>

int sched\_get\_priority\_max (int policy)

**Description:** This function returns the value of the highest possible task priority for a specified scheduling policy.

**Input Parameters:**

* policy. Scheduling policy requested.

**Returned Value:**

* The maximum priority value or -1 (ERROR).

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.2.7 sched\_get\_priority\_min**

**Function Prototype:**

#include <sched.h>

int sched\_get\_priority\_min (int policy);

**Description:** This function returns the value of the lowest possible task priority for a specified scheduling policy.

**Input Parameters:**

* policy. Scheduling policy requested.

**Returned Value:**

* The minimum priority value or -1 (ERROR)

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.2.8 sched\_get\_rr\_interval**

**Function Prototype:**

#include <sched.h>

int sched\_get\_rr\_interval (pid\_t pid, struct timespec \*interval);

**Description:** sched\_rr\_get\_interval() writes the timeslice interval for task identified by pid into the timespec structure pointed to by interval. If pid is zero, the timeslice for the calling process is written into 'interval. The identified process should be running under the SCHED\_RR scheduling policy.'

**Input Parameters:**

* pid. The task ID of the task. If pid is zero, the priority of the calling task is returned.
* interval. A structure used to return the time slice.

**Returned Value:** On success, sched\_rr\_get\_interval() returns OK (0). On error, ERROR (-1) is returned, and errno is set to:

* EFAULT: Cannot copy to interval
* EINVAL: Invalid pid.
* ENOSYS: The system call is not yet implemented.
* ESRCH: The process whose ID is pid could not be found.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

|  |
| --- |
| **2.3 Task Control Interfaces** |

**Task Control Interfaces**.

* **Scheduler locking interfaces**. This *non-standard* interfaces are used to enable and disable pre-emption and to test is pre-emption is currently enabled.
  + 2.3.1 sched\_lock
  + 2.3.2 sched\_unlock
  + 2.3.3 sched\_lockcount
* **Task synchronization interfaces**. wait(), waitpid() or waitid() may be used to wait for termination of child tasks.
  + 2.3.4 waitpid
  + 2.3.5 waitid
  + 2.3.6 wait
* **Task Exit Hooks**. atexit() and on\_exit() may be use to register callback functions that are executed when a *task group*terminates. A task group is the functional analog of a process: It is a group that consists of the main task thread and of all of the pthreads created by the main task thread or any of the other pthreads within the task group. Members of a task group share certain resources such as environment variables, file descriptors, FILE streams, sockets, pthread keys and open message queues.

**NOTE:** Behavior of features related to *task group*s depend of NuttX configuration settings. See the discussion of "Parent and Child Tasks," below. See also the [NuttX Threading Wiki](http://www.nuttx.org/doku.php?id=wiki:nxinternal:nxtasking) page and the [Tasks vs. Threads FAQ](http://www.nuttx.org/doku.php?id=wiki:nxinternal:tasksnthreads) for additional information on tasks and threads in NuttX.

A *task group* terminates when the last thread within the group exits.

* + 2.3.7 atexit
  + 2.3.8 on\_exit

**Parent and Child Tasks**. The task synchronization interfaces historically depend upon parent and child relationships between tasks. But default, NuttX does not use any parent/child knowledge. However, there are three important configuration options that can change that.

* **CONFIG\_SCHED\_HAVE\_PARENT**. If this setting is defined, then it instructs NuttX to remember the task ID of the parent task when each new child task is created. This support enables some additional features (such as SIGCHLD) and modifies the behavior of other interfaces. For example, it makes waitpid() more standards complete by restricting the waited-for tasks to the children of the caller.
* **CONFIG\_SCHED\_CHILD\_STATUS** If this option is selected, then the exit status of the child task will be retained after the child task exits. This option should be selected if you require knowledge of a child process' exit status. Without this setting, wait(), waitpid() or waitid() may fail. For example, if you do:
  1. Start child task
  2. Wait for exit status (using wait(), waitpid() or waitid()).

This may fail because the child task may run to completion before the wait begins. There is a non-standard work-around in this case: The above sequence will work if you disable pre-emption using sched\_lock() prior to starting the child task, then re-enable pre-emption with sched\_unlock() after the wait completes. This works because the child task is not permitted to run until the wait is in place.

The standard solution would be to enable CONFIG\_SCHED\_CHILD\_STATUS. In this case the exit status of the child task is retained after the child exits and the wait will successful obtain the child task's exit status whether it is called before the child task exits or not.

* **CONFIG\_PREALLOC\_CHILDSTATUS**. To prevent runaway child status allocations and to improve allocation performance, child task exit status structures are pre-allocated when the system boots. This setting determines the number of child status structures that will be pre-allocated. If this setting is not defined or if it is defined to be zero then a value of 2\*MAX\_TASKS is used.

Note that there cannot be more that CONFIG\_MAX\_TASKS tasks in total. However, the number of child status structures may need to be significantly larger because this number includes the maximum number of tasks that are running PLUS the number of tasks that have exit'ed without having their exit status reaped (via wait(), waitpid() or waitid()).

Obviously, if tasks spawn children indefinitely and never have the exit status reaped, then you may have a memory leak! (See **Warning** below)

**Warning**: If you enable the CONFIG\_SCHED\_CHILD\_STATUS feature, then your application must either (1) take responsibility for reaping the child status with wait(), waitpid() or waitid(), or (2) suppress retention of child status. If you do not reap the child status, then you have a memory leak and your system will eventually fail.

Retention of child status can be suppressed on the parent using logic like:

struct sigaction sa;

sa.sa\_handler = SIG\_IGN;

sa.sa\_flags = SA\_NOCLDWAIT;

int ret = sigaction(SIGCHLD, &sa, NULL);

**2.3.1 sched\_lock**

**Function Prototype:**

#include <sched.h>

int sched\_lock(void);

**Description:** This function disables context switching by Disabling addition of new tasks to the ready-to-run task list. The task that calls this function will be the only task that is allowed to run until it either calls sched\_unlock (the appropriate number of times) or until it blocks itself.

**Input Parameters:** None.

**Returned Value:**

* OK or ERROR.

**Assumptions/Limitations:**

**POSIX Compatibility:** This is a NON-POSIX interface. VxWorks provides the comparable interface:

STATUS taskLock(void);

**2.3.2 sched\_unlock**

**Function Prototype:**

#include <sched.h>

int sched\_unlock(void);

**Description:** This function decrements the preemption lock count. Typically this is paired with sched\_lock() and concludes a critical section of code. Preemption will not be unlocked until sched\_unlock() has been called as many times as sched\_lock(). When the lockCount is decremented to zero, any tasks that were eligible to preempt the current task will execute.

**Input Parameters:** None.

**Returned Value:**

* OK or ERROR.

**Assumptions/Limitations:**

**POSIX Compatibility:** This is a NON-POSIX interface. VxWorks provides the comparable interface:

STATUS taskUnlock(void);

**2.3.3 sched\_lockcount**

**Function Prototype:**

#include <sched.h>

int32\_t sched\_lockcount(void)

**Description:** This function returns the current value of the lockCount. If zero, preemption is enabled; if non-zero, this value indicates the number of times that sched\_lock() has been called on this thread of execution.

**Input Parameters:** None.

**Returned Value:**

* The current value of the lockCount.

**Assumptions/Limitations:**

**POSIX Compatibility:** None.

**2.3.4 waitpid**

**Function Prototype:**

#include <sys/wait.h>

ipid\_t waitpid(pid\_t pid, int \*stat\_loc, int options);

**Description:**

The following discussion is a general description of the waitpid() interface. However, as of this writing, the implementation of waitpid() is incomplete (but usable). If CONFIG\_SCHED\_HAVE\_PARENT is defined, waitpid() will be a little more compliant to specifications. Without CONFIG\_SCHED\_HAVE\_PARENT, waitpid() simply supports waiting for any task to complete execution. With CONFIG\_SCHED\_HAVE\_PARENT, waitpid()will use SIGCHLD and can, therefore, wait for any child of the parent to complete. The implementation is incomplete in either case, however: NuttX does not support any concept of process groups. Nor does NuttX retain the status of exited tasks so if waitpid() is called after a task has exited, then no status will be available. The options argument is currently ignored.

The waitpid() functions will obtain status information pertaining to one of the caller's child processes. The waitpid() function will suspend execution of the calling thread until status information for one of the terminated child processes of the calling process is available, or until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process. If more than one thread is suspended in waitpid() awaiting termination of the same process, exactly one thread will return the process status at the time of the target process termination. If status information is available prior to the call to waitpid(), return will be immediate.

**NOTE**: Because waitpid() is not fully POSIX compliant, it must be specifically enabled by setting CONFIG\_SCHED\_WAITPID in the NuttX configuration file.

**Input Parameters:**

* pid. The task ID of the thread to waid for
* stat\_loc. The location to return the exit status
* options. ignored

The pid argument specifies a set of child processes for which status is requested. The waitpid() function will only return the status of a child process from this set:

* If pid is equal to (pid\_t)-1), status is requested for any child process. In this respect, waitpid() is then equivalent to  wait().
* If pid is greater than 0, it specifies the process ID of a single child process for which status is requested.
* If pid is 0, status is requested for any child process whose process group ID is equal to that of the calling process.
* If pid is less than (pid\_t)-1), status is requested for any child process whose process group ID is equal to the absolute value of pid.

The options argument is constructed from the bitwise-inclusive OR of zero or more of the following flags, defined in the <sys/wait.h>header:

* WCONTINUED. The waitpid() function will report the status of any continued child process specified by pid whose status has not been reported since it continued from a job control stop.
* WNOHANG. The waitpid() function will not suspend execution of the calling thread if status is not immediately available for one of the child processes specified by pid.
* WUNTRACED. The status of any child processes specified by pid that are stopped, and whose status has not yet been reported since they stopped, will also be reported to the requesting process.

If the calling process has SA\_NOCLDWAIT set or has SIGCHLD set to SIG\_IGN, and the process has no unwaited-for children that were transformed into zombie processes, the calling thread will block until all of the children of the process containing the calling thread terminate, and waitpid() will fail and set errno to ECHILD.

If waitpid() returns because the status of a child process is available, these functions will return a value equal to the process ID of the child process. In this case, if the value of the argument stat\_loc is not a null pointer, information will be stored in the location pointed to by stat\_loc. The value stored at the location pointed to by stat\_loc will be 0 if and only if the status returned is from a terminated child process that terminated by one of the following means:

1. The process returned 0 from main().
2. The process called \_exit() or exit() with a status argument of 0.
3. The process was terminated because the last thread in the process terminated.

Regardless of its value, this information may be interpreted using the following macros, which are defined in <sys/wait.h> and evaluate to integral expressions; the stat\_val argument is the integer value pointed to by stat\_loc.

* WIFEXITED(stat\_val). Evaluates to a non-zero value if status was returned for a child process that terminated normally.
* WEXITSTATUS(stat\_val). If the value of WIFEXITED(stat\_val) is non-zero, this macro evaluates to the low-order 8 bits of the status argument that the child process passed to \_exit() or exit(), or the value the child process returned from main().
* WIFSIGNALED(stat\_val). Evaluates to a non-zero value if status was returned for a child process that terminated due to the receipt of a signal that was not caught (see >signal.h<).
* WTERMSIG(stat\_val). If the value of WIFSIGNALED(stat\_val) is non-zero, this macro evaluates to the number of the signal that caused the termination of the child process.
* WIFSTOPPED(stat\_val). Evaluates to a non-zero value if status was returned for a child process that is currently stopped.
* WSTOPSIG(stat\_val). If the value of WIFSTOPPED(stat\_val) is non-zero, this macro evaluates to the number of the signal that caused the child process to stop.
* WIFCONTINUED(stat\_val). Evaluates to a non-zero value if status was returned for a child process that has continued from a job control stop.

**Returned Value:**

If waitpid() returns because the status of a child process is available, it will return a value equal to the process ID of the child process for which status is reported.

If waitpid() returns due to the delivery of a signal to the calling process, -1 will be returned and errno set to EINTR.

If waitpid() was invoked with WNOHANG set in options, it has at least one child process specified by pid for which status is not available, and status is not available for any process specified by pid, 0 is returned.

Otherwise, (pid\_t)-1errno set to indicate the error:

* ECHILD. The process specified by pid does not exist or is not a child of the calling process, or the process group specified by piddoes not exist or does not have any member process that is a child of the calling process.
* EINTR. The function was interrupted by a signal. The value of the location pointed to by stat\_loc is undefined.
* EINVAL. The options argument is not valid.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name, but the implementation is incomplete (as detailed above).

**2.3.5 waitid**

**Function Prototype:**

#include <sys/wait.h>

#ifdef CONFIG\_SCHED\_HAVE\_PARENT

int waitid(idtype\_t idtype, id\_t id, FAR siginfo\_t \*info, int options);

#endif

**Description:**

The following discussion is a general description of the waitid() interface. However, as of this writing, the implementation of waitid() is incomplete (but usable). If CONFIG\_SCHED\_HAVE\_PARENT is defined, waitid() will be a little more compliant to specifications. waitpid() simply supports waiting a specific child task (P\_PID or for any child task P\_ALL to complete execution. SIGCHLD is used. The implementation is incomplete in either case, however: NuttX does not support any concept of process groups. Nor does NuttX retain the status of exited tasks so if waitpid() is called after a task has exited, then no status will be available. The options argument is currently ignored.

The waitid() function suspends the calling thread until one child of the process containing the calling thread changes state. It records the current state of a child in the structure pointed to by info. If a child process changed state prior to the call to waitid(), waitid()returns immediately. If more than one thread is suspended in wait() or waitpid() waiting termination of the same process, exactly one thread will return the process status at the time of the target process termination

The idtype and id arguments are used to specify which children waitid() will wait for.

* If idtype is P\_PID, waitid() will wait for the child with a process ID equal to (pid\_t)id.
* If idtype is P\_PGID, waitid() will wait for any child with a process group ID equal to (pid\_t)id.
* If idtype is P\_ALL, waitid() will wait for any children and id is ignored.

The options argument is used to specify which state changes waitid() will will wait for. It is formed by OR-ing together one or more of the following flags:

* WEXITED: Wait for processes that have exited.
* WSTOPPED: Status will be returned for any child that has stopped upon receipt of a signal.
* WCONTINUES: Status will be returned for any child that was stopped and has been continued.
* WNOHANG: Return immediately if there are no children to wait for.
* WNOWAIT: Keep the process whose status is returned in info in a waitable state. This will not affect the state of the process; the process may be waited for again after this call completes.

The info argument must point to a siginfo\_t structure. If waitid() returns because a child process was found that satisfied the conditions indicated by the arguments idtype and options, then the structure pointed to by info will be filled in by the system with the status of the process. The si\_signo member will always be equal to SIGCHLD.

**Input Parameters:** See the description above.

**Returned Value:** If waitid() returns due to the change of state of one of its children, 0 is returned. Otherwise, -1 is returned and errnois set to indicate the error.

The waitid() function will fail if:

* ECHILD:

The calling process has no existing unwaited-for child processes.

* EINTR:

The waitid() function was interrupted by a signal.

* EINVAL: An invalid value was specified for options, or idtype and id specify an invalid set of processes.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name, but the implementation is incomplete (as detailed in the description above).

**2.3.6 wait**

**Function Prototype:**

#include <sys/wait.h>

#ifdef CONFIG\_SCHED\_HAVE\_PARENT

pid\_t wait(FAR int \*stat\_loc);

#endif

**Description:**

The following discussion is a general description of the wait() interface. However, as of this writing, the implementation of wait() is incomplete (but usable). wait() is based on waitpaid(). See the description of waitpaid() for further information.

The wait() function will suspend execution of the calling thread until status information for one of its terminated child processes is available, or until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process. If more than one thread is suspended in wait() awaiting termination of the same process, exactly one thread will return the process status at the time of the target process termination. If status information is available prior to the call towait(), return will be immediate.

The waitpid() function will behave identically to wait(), if its pid argument is (pid\_t)-1 and the options argument is 0. Otherwise, its behavior will be modified by the values of the pid and options arguments.

**Input Parameters:**

* stat\_loc. The location to return the exit status

**Returned Value:** See the values returned by waitpaid().

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name, but the implementation is incomplete (as detailed in the description waitpaid()).

**2.3.7 atexit**

**Function Prototype:**

#include <stdlib.h>

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

**Description:** Registers a function to be called at program exit. The atexit() function registers the given function to be called at normal process termination, whether via exit() or via return from the program's main().

**NOTE**: CONFIG\_SCHED\_ATEXIT must be defined to enable this function.

**Input Parameters:**

* func. A pointer to the function to be called when the task exits.

**Returned Value:** On success, atexit() returns OK (0). On error, ERROR (-1) is returned, and errno is set to indicate the cause of the failure.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the ISO C interface of the same name. Limitiations in the current implementation:

1. Only a single atexit function can be registered unless CONFIG\_SCHED\_ATEXIT\_MAX defines a larger number.
2. atexit() functions are not inherited when a new task is created.

**2.3.8 on\_exit**

**Function Prototype:**

#include <stdlib.h>

int on\_exit(CODE void (\*func)(int, FAR void \*), FAR void \*arg)

**Description:** Registers a function to be called at program exit. The on\_exit() function registers the given function to be called at normal process termination, whether via exit() or via return from the program's main(). The function is passed the status argument given to the last call to exit() and the arg argument from on\_exit().

**NOTE**: CONFIG\_SCHED\_ONEXIT must be defined to enable this function

**Input Parameters:**

* func. A pointer to the function to be called when the task exits.
* arg. An argument that will be provided to the on\_exit() function when the task exits.

**Returned Value:** On success, on\_exit() returns OK (0). On error, ERROR (-1) is returned, and errno is set to indicate the cause of the failure.

**Assumptions/Limitations:**

**POSIX Compatibility:** This function comes from SunOS 4, but is also present in libc4, libc5 and glibc. It no longer occurs in Solaris (SunOS 5). Avoid this function, and use the standard atexit() instead.

1. Only a single on\_exit function can be registered unless CONFIG\_SCHED\_ONEXIT\_MAX defines a larger number.
2. on\_exit() functions are not inherited when a new task is created.

|  |
| --- |
| **2.4 Named Message Queue Interfaces** |

NuttX supports POSIX named message queues for inter-task communication. Any task may send or receive messages on named message queues. Interrupt handlers may send messages via named message queues.

* 2.4.1 mq\_open
* 2.4.2 mq\_close
* 2.4.3 mq\_unlink
* 2.4.4 mq\_send
* 2.4.5 mq\_timedsend
* 2.4.6 mq\_receive
* 2.4.7 mq\_timedreceive
* 2.4.8 mq\_notify
* 2.4.9 mq\_setattr
* 2.4.10 mq\_getattrdf

**2.4.1 mq\_open**

**Function Prototype:**

#include <mqueue.h>

mqd\_t mq\_open(const char \*mqName, int oflags, ...);

**Description:** This function establish a connection between a named message queue and the calling task. After a successful call of mq\_open(), the task can reference the message queue using the address returned by the call. The message queue remains usable until it is closed by a successful call to mq\_close().

**Input Parameters:**

* mqName. Name of the queue to open
* oflags. Open flags. These may be any combination of:
  + O\_RDONLY. Open for read access.
  + O\_WRONLY. Open for write access.
  + O\_RDWR. Open for both read & write access.
  + O\_CREAT. Create message queue if it does not already exist.
  + O\_EXCL. Name must not exist when opened.
  + O\_NONBLOCK. Don't wait for data.
* *... Optional parameters*. When the O\_CREAT flag is specified, POSIX requires that a third and fourth parameter be supplied:
  + mode. The mode parameter is of type mode\_t. In the POSIX specification, this mode value provides file permission bits for the message queue. This parameter is required but not used in the present implementation.
  + attr. A pointer to an mq\_attr that is provided to initialize. the message queue. If attr is NULL, then the messages queue is created with implementation-defined default message queue attributes. If attr is non-NULL, then the message queue mq\_maxmsg attribute is set to the corresponding value when the queue is created. The mq\_maxmsg attribute determines the maximum number of messages that can be queued before addition attempts to send messages on the message queue fail or cause the sender to block; the mq\_msgsize attribute determines the maximum size of a message that can be sent or received. Other elements of attr are ignored (i.e, set to default message queue attributes).

**Returned Value:**

* A message queue descriptor or -1 (ERROR)

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name. Differences from the full POSIX implementation include:

* The mq\_msgsize attributes determines the maximum size of a message that may be sent or received. In the present implementation, this maximum message size is limited at 22 bytes.

**2.4.2 mq\_close**

**Function Prototype:**

#include <mqueue.h>

int mq\_close(mqd\_t mqdes);

**Description:** This function is used to indicate that the calling task is finished with the specified message queued mqdes. The mq\_close() deallocates any system resources allocated by the system for use by this task for its message queue.

If the calling task has attached a notification request to the message queue via this mqdes (see mq\_notify()), this attachment will be removed and the message queue is available for another task to attach for notification.

**Input Parameters:**

* mqdes. Message queue descriptor.

**Returned Value:**

* 0 (OK) if the message queue is closed successfully, otherwise, -1 (ERROR).

**Assumptions/Limitations:**

* The behavior of a task that is blocked on either a mq\_send() or mq\_receive() is undefined when mq\_close() is called.
* The result of using this message queue descriptor after successful return from mq\_close() is undefined.

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.4.3 mq\_unlink**

**Function Prototype:**

#include <mqueue.h>

int mq\_unlink(const char \*mqName);

**Description:** This function removes the message queue named by "mqName." If one or more tasks have the message queue open when mq\_unlink() is called, removal of the message queue is postponed until all references to the message queue have been closed.

**Input Parameters:**

* mqName. Name of the message queue

**Returned Value:** None.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.4.4 mq\_send**

**Function Prototype:**

#include <mqueue.h>

int mq\_send(mqd\_t mqdes, const void \*msg, size\_t msglen, int prio);

**Description:** This function adds the specified message, msg, to the message queue, mqdes. The msglen parameter specifies the length of the message in bytes pointed to by msg. This length must not exceed the maximum message length from the mq\_getattr().

If the message queue is not full, mq\_send() will place the msg in the message queue at the position indicated by the prio argument. Messages with higher priority will be inserted before lower priority messages The value of prio must not exceed MQ\_PRIO\_MAX.

If the specified message queue is full and O\_NONBLOCK is not set in the message queue, then mq\_send() will block until space becomes available to the queue the message.

If the message queue is full and NON\_BLOCK is set, the message is not queued and ERROR is returned.

**NOTE**: mq\_send() may be called from an interrupt handler. However, it behaves differently when called from the interrupt level:

* It does not check the size of the queue. It will always post the message, even if there is already too many messages in queue. This is because the interrupt handler does not have the option of waiting for the message queue to become non-full.
* It doesn't allocate new memory (because you cannot allocate memory from an interrup handler). Instead, there are are pool of pre-allocated message structures that may be used just for sending messages from interrupt handlers. The number of such pre-allocated messages is a configuration parameter.

**Input Parameters:**

* mqdes. Message queue descriptor.
* msg. Message to send.
* msglen. The length of the message in bytes.
* prio. The priority of the message.

**Returned Value:** On success, mq\_send() returns 0 (OK); on error, -1 (ERROR) is returned, with errno set to indicate the error:

* EAGAIN. The queue was empty, and the O\_NONBLOCK flag was set for the message queue description referred to by mqdes.
* EINVAL. Either msg or mqdes is NULL or the value of prio is invalid.
* EPERM. Message queue opened not opened for writing.
* EMSGSIZE. msglen was greater than the maxmsgsize attribute of the message queue.
* EINTR. The call was interrupted by a signal handler.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**mq\_timedsend**

**Function Prototype:**

#include <mqueue.h>

int mq\_timedsend(mqd\_t mqdes, const char \*msg, size\_t msglen, int prio,

const struct timespec \*abstime);

**Description:** This function adds the specified message, msg, to the message queue, mqdes. The msglen parameter specifies the length of the message in bytes pointed to by msg. This length must not exceed the maximum message length from the mq\_getattr().

If the message queue is not full, mq\_timedsend() will place the msg in the message queue at the position indicated by the prio argument. Messages with higher priority will be inserted before lower priority messages The value of prio must not exceed MQ\_PRIO\_MAX.

If the specified message queue is full and O\_NONBLOCK is not set in the message queue, then mq\_send() will block until space becomes available to the queue the message or until a timeout occurs.

mq\_timedsend() behaves just like mq\_send(), except that if the queue is full and the O\_NONBLOCK flag is not enabled for the message queue description, then abstime points to a structure which specifies a ceiling on the time for which the call will block. This ceiling is an absolute timeout in seconds and nanoseconds since the Epoch (midnight on the morning of 1 January 1970).

If the message queue is full, and the timeout has already expired by the time of the call, mq\_timedsend() returns immediately.

**Input Parameters:**

* mqdes. Message queue descriptor.
* msg. Message to send.
* msglen. The length of the message in bytes.
* prio. The priority of the message.

**Returned Value:** On success, mq\_send() returns 0 (OK); on error, -1 (ERROR) is returned, with errno set to indicate the error:

* EAGAIN. The queue was empty, and the O\_NONBLOCK flag was set for the message queue description referred to by mqdes.
* EINVAL. Either msg or mqdes is NULL or the value of prio is invalid.
* EPERM. Message queue opened not opened for writing.
* EMSGSIZE. msglen was greater than the maxmsgsize attribute of the message queue.
* EINTR. The call was interrupted by a signal handler.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.4.5 mq\_receive**

**Function Prototype:**

#include <mqueue.h>

ssize\_t mq\_receive(mqd\_t mqdes, void \*msg, size\_t msglen, int \*prio);

**Description:** This function receives the oldest of the highest priority messages from the message queue specified by mqdes. If the size of the buffer in bytes, msgLen, is less than the mq\_msgsize attribute of the message queue, mq\_receive() will return an error. Otherwise, the selected message is removed from the queue and copied to msg.

If the message queue is empty and O\_NONBLOCK was not set, mq\_receive() will block until a message is added to the message queue. If more than one task is waiting to receive a message, only the task with the highest priority that has waited the longest will be unblocked.

If the queue is empty and O\_NONBLOCK is set, ERROR will be returned.

**Input Parameters:**

* mqdes. Message Queue Descriptor.
* msg. Buffer to receive the message.
* msglen. Size of the buffer in bytes.
* prio. If not NULL, the location to store message priority.

**Returned Value:**. One success, the length of the selected message in bytes is returned. On failure, -1 (ERROR) is returned and the errno is set appropriately:

* EAGAIN The queue was empty and the O\_NONBLOCK flag was set for the message queue description referred to by mqdes.
* EPERM Message queue opened not opened for reading.
* EMSGSIZE msglen was less than the maxmsgsize attribute of the message queue.
* EINTR The call was interrupted by a signal handler.
* EINVAL Invalid msg or mqdes

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.4.6 mq\_timedreceive**

**Function Prototype:**

#include <mqueue.h>

ssize\_t mq\_timedreceive(mqd\_t mqdes, void \*msg, size\_t msglen,

int \*prio, const struct timespec \*abstime);

**Description:** This function receives the oldest of the highest priority messages from the message queue specified by mqdes. If the size of the buffer in bytes, msgLen, is less than the mq\_msgsize attribute of the message queue, mq\_timedreceive() will return an error. Otherwise, the selected message is removed from the queue and copied to msg.

If the message queue is empty and O\_NONBLOCK was not set, mq\_timedreceive() will block until a message is added to the message queue (or until a timeout occurs). If more than one task is waiting to receive a message, only the task with the highest priority that has waited the longest will be unblocked.

mq\_timedreceive() behaves just like mq\_receive(), except that if the queue is empty and the O\_NONBLOCK flag is not enabled for the message queue description, then abstime points to a structure which specifies a ceiling on the time for which the call will block. This ceiling is an absolute timeout in seconds and nanoseconds since the Epoch (midnight on the morning of 1 January 1970).

If no message is available, and the timeout has already expired by the time of the call, mq\_timedreceive() returns immediately.

**Input Parameters:**

* mqdes. Message Queue Descriptor.
* msg. Buffer to receive the message.
* msglen. Size of the buffer in bytes.
* prio. If not NULL, the location to store message priority.
* abstime. The absolute time to wait until a timeout is declared.

**Returned Value:**. One success, the length of the selected message in bytes is returned. On failure, -1 (ERROR) is returned and the errno is set appropriately:

* EAGAIN: The queue was empty and the O\_NONBLOCK flag was set for the message queue description referred to by mqdes.
* EPERM: Message queue opened not opened for reading.
* EMSGSIZE: msglen was less than the maxmsgsize attribute of the message queue.
* EINTR: The call was interrupted by a signal handler.
* EINVAL: Invalid msg or mqdes or abstime
* ETIMEDOUT: The call timed out before a message could be transferred.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.4.7 mq\_notify**

**Function Prototype:**

#include <mqueue.h>

int mq\_notify(mqd\_t mqdes, const struct sigevent \*notification);

**Description:** If the notification input parameter is not NULL, this function connects the task with the message queue such that the specified signal will be sent to the task whenever the message changes from empty to non-empty. One notification can be attached to a message queue.

If notification; is NULL, the attached notification is detached (if it was held by the calling task) and the queue is available to attach another notification.

When the notification is sent to the registered task, its registration will be removed. The message queue will then be available for registration.

**Input Parameters:**

* mqdes. Message queue descriptor
* notification. Real-time signal structure containing:
  + sigev\_notify. Should be SIGEV\_SIGNAL (but actually ignored)
  + sigev\_signo. The signo to use for the notification
  + sigev\_value. Value associated with the signal

**Returned Value:** On success mq\_notify() returns 0; on error, -1 is returned, with errno set to indicate the error:

* EBADF. The descriptor specified in mqdes is invalid.
* EBUSY. Another process has already registered to receive notification for this message queue.
* EINVAL. sevp->sigev\_notify is not one of the permitted values; or sevp->sigev\_notify is SIGEV\_SIGNAL and sevp->sigev\_signo is not a valid signal number.
* ENOMEM. Insufficient memory.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name. Differences from the full POSIX implementation include:

* The notification signal will be sent to the registered task even if another task is waiting for the message queue to become non-empty. This is inconsistent with the POSIX specification which states, "If a process has registered for notification of message arrival at a message queue and some process is blocked in mq\_receive waiting to receive a message when a message arrives at the queue, the arriving message will satisfy the appropriate mq\_receive() ... The resulting behavior is as if the message queue remains empty, and no notification will be sent."

**2.4.8 mq\_setattr**

**Function Prototype:**

#include <mqueue.h>

int mq\_setattr(mqd\_t mqdes, const struct mq\_attr \*mqStat,

struct mq\_attr \*oldMqStat);

**Description:** This function sets the attributes associated with the specified message queue "mqdes." Only the "O\_NONBLOCK" bit of the "mq\_flags" can be changed.

If "oldMqStat" is non-null, mq\_setattr() will store the previous message queue attributes at that location (just as would have been returned by mq\_getattr()).

**Input Parameters:**

* mqdes. Message queue descriptor
* mqStat. New attributes
* oldMqState. Old attributes

**Returned Value:**

* 0 (OK) if attributes are set successfully, otherwise -1 (ERROR).

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.4.9 mq\_getattr**

**Function Prototype:**

#include <mqueue.h>

int mq\_getattr(mqd\_t mqdes, struct mq\_attr \*mqStat);

**Description:** This functions gets status information and attributes associated with the specified message queue.

**Input Parameters:**

* mqdes. Message queue descriptor
* mqStat. Buffer in which to return attributes. The returned attributes include:
  + mq\_maxmsg. Max number of messages in queue.
  + mq\_msgsize. Max message size.
  + mq\_flags. Queue flags.
  + mq\_curmsgs. Number of messages currently in queue.

**Returned Value:**

* 0 (OK) if attributes provided, -1 (ERROR) otherwise.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

|  |
| --- |
| **2.5 Counting Semaphore Interfaces** |

**Semaphores**. Semaphores are the basis for synchronization and mutual exclusion in NuttX. NuttX supports POSIX semaphores.

Semaphores are the preferred mechanism for gaining exclusive access to a resource. sched\_lock() and sched\_unlock() can also be used for this purpose. However, sched\_lock() and sched\_unlock() have other undesirable side-affects in the operation of the system: sched\_lock() also prevents higher-priority tasks from running that do not depend upon the semaphore-managed resource and, as a result, can adversely affect system response times.

**Priority Inversion**. Proper use of semaphores avoids the issues of sched\_lock(). However, consider the following example:

1. Some low-priority task, *Task C*, acquires a semaphore in order to get exclusive access to a protected resource.
2. *Task C* is suspended to allow some high-priority task,

*Task A*, to execute.

1. *Task A* attempts to acquire the semaphore held by *Task C* and gets blocked until *Task C* relinquishes the semaphore.
2. *Task C* is allowed to execute again, but gets suspended by some medium-priority *Task B*.

At this point, the high-priority *Task A* cannot execute until *Task B* (and possibly other medium-priority tasks) completes and until *Task C*relinquishes the semaphore. In effect, the high-priority task, *Task A* behaves as though it were lower in priority than the low-priority task, *Task C*! This phenomenon is called *priority inversion*.

Some operating systems avoid priority inversion by *automatically* increasing the priority of the low-priority *Task C* (the operable buzz-word for this behavior is *priority inheritance*). NuttX supports this behavior, but only if CONFIG\_PRIORITY\_INHERITANCE is defined in your OS configuration file. If CONFIG\_PRIORITY\_INHERITANCE is not defined, then it is left to the designer to provide implementations that will not suffer from priority inversion. The designer may, as examples:

* Implement all tasks that need the semaphore-managed resources at the same priority level,
* Boost the priority of the low-priority task before the semaphore is acquired, or
* Use sched\_lock() in the low-priority task.

**Priority Inheritance**. As mentioned, NuttX does support *priority inheritance* provided that CONFIG\_PRIORITY\_INHERITANCE is defined in your OS configuration file. However, the implementation and configuration of the priority inheritance feature is sufficiently complex that more needs to be said. How can a feature that can be described by a single, simple sentence require such a complex implementation:

* **CONFIG\_SEM\_PREALLOCHOLDERS.** First of all, in NuttX priority inheritance is implement on POSIX counting semaphores. The reason for this is that these semaphores are the most primitive waiting mechanism in NuttX; Most other waiting facilities are based on semaphores. So if priority inheritance is implemented for POSIX counting semaphores, then most NuttX waiting mechanisms will have this capability.

Complexity arises because counting semaphores can have numerous holders of semaphore counts. Therefore, in order to implement priority inheritance across all holders, then internal data structures must be allocated to manage the various holders associated with a semaphore. The setting CONFIG\_SEM\_PREALLOCHOLDERS defines the maximum number of different threads (minus one per semaphore instance) that can take counts on a semaphore with priority inheritance support. This setting defines the size of a single pool of pre-allocated structures. It may be set to zero if priority inheritance is disabled OR if you are only using semaphores as mutexes (only one holder) OR if no more than two threads participate using a counting semaphore.

The cost associated with setting CONFIG\_SEM\_PREALLOCHOLDERS is slightly increased code size and around 6-12 bytes times the value of CONFIG\_SEM\_PREALLOCHOLDERS.

* **CONFIG\_SEM\_NNESTPRIO:** In addition, there may be multiple threads of various priorities that need to wait for a count from the semaphore. These, the lower priority thread holding the semaphore may have to be boosted numerous time and, to make things more complex, will have to keep track of all of the boost priorities values in in order to correctly restore the priorities after a count has been handed out to the higher priority thread. The CONFIG\_SEM\_NNESTPRIO defines the size of an array, one array per active thread. This setting is the maximum number of higher priority threads (minus 1) than can be waiting for another thread to release a count on a semaphore. This value may be set to zero if no more than one thread is expected to wait for a semaphore.

The cost associated with setting CONFIG\_SEM\_NNESTPRIO is slightly increased code size and (CONFIG\_SEM\_PREALLOCHOLDERS + 1) times the maximum number of active threads.

* **Increased Susceptibility to Bad Thread Behavior**. These various structures tie the semaphore implementation more tightly to the behavior of the implementation. For examples, if a thread executes while holding counts on a semaphore, or if a thread exits without call sem\_destroy() then. Or what if the thread with the boosted priority re-prioritizes itself? The NuttX implement of priority inheritance attempts to handle all of these types of corner cases, but it is very likely that some are missed. The worst case result is that memory could by stranded within the priority inheritance logic.

**Locking versus Signaling Semaphores**. Semaphores (and mutexes) may be used for many different purposes. One typical use of for mutual exclusion and locking of resources: In this usage, the thread that needs exclusive access to a resources takes the semaphore to get access to the resource. The same thread subsequently releases the seamphore count when it no longer needs exclusive access. Priority inheritance is intended just for this usage case.

In a different usage case, a semaphore may to be used to signal an event: One thread A waits on a semaphore for an event to occur. When the event occurs, another thread B will post the semaphore waking the waiting thread A. This is a completely different usage model; notice that in the mutual exclusion case, the same thread takes and posts the semaphore. In the signaling case, one thread takes the seamphore and a different thread posts the samphore. Priority inheritance should *never* be used in this signaling case. Subtle, strange behaviors may result.

When priority inheritance is enabled with CONFIG\_PRIORITY\_INHERITANCE, the default *protocol* for the semaphore will be to use priority inheritance. For signaling semaphores, priority inheritance must be explicitly disabled by calling sem\_setprotocol with SEM\_PRIO\_NONE. For the case of pthread mutexes, pthread\_mutexattr\_setprotocol with PTHREAD\_PRIO\_NONE.

This is discussed in much more detail on this [Wiki page](http://www.nuttx.org/doku.php?id=wiki:howtos:signalling-semaphores).

**POSIX semaphore interfaces:**

* 2.5.1 sem\_init
* 2.5.2 sem\_destroy
* 2.5.3 sem\_open
* 2.5.4 sem\_close
* 2.5.5 sem\_unlink
* 2.5.6 sem\_wait
* 2.5.7 sem\_timedwait
* 2.5.8 sem\_trywait
* 2.5.9 sem\_post
* 2.5.10 sem\_getvalue
* 2.5.11 sem\_getprotocol
* 2.5.12 sem\_setprotocol

**2.5.1 sem\_init**

**Function Prototype:**

#include <semaphore.h>

int sem\_init(sem\_t \*sem, int pshared, unsigned int value);

**Description:** This function initializes the UN-NAMED semaphore sem. Following a successful call to sem\_init(), the semaphore may be used in subsequent calls to sem\_wait(), sem\_post(), and sem\_trywait(). The semaphore remains usable until it is destroyed.

Only sem itself may be used for performing synchronization. The result of referring to copies of sem in calls to sem\_wait(),sem\_trywait(), sem\_post(), and sem\_destroy(), is not defined.

**Input Parameters:**

* sem. Semaphore to be initialized
* pshared. Process sharing (not used)
* value. Semaphore initialization value

**Returned Value:**

* 0 (OK), or -1 (ERROR) if unsuccessful.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name. Differences from the full POSIX implementation include:

* pshared is not used.

**2.5.2 sem\_destroy**

**Function Prototype:**

#include <semaphore.h>

int sem\_destroy(sem\_t \*sem);

**Description:** This function is used to destroy the un-named semaphore indicated by sem. Only a semaphore that was created usingsem\_init() may be destroyed using sem\_destroy(). The effect of calling sem\_destroy() with a named semaphore is undefined. The effect of subsequent use of the semaphore sem is undefined until sem is re-initialized by another call to sem\_init().

The effect of destroying a semaphore upon which other tasks are currently blocked is undefined.

**Input Parameters:**

* sem. Semaphore to be destroyed.

**Returned Value:**

* 0 (OK), or -1 (ERROR) if unsuccessful.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.5.3 sem\_open**

**Function Prototype:**

#include <semaphore.h>

sem\_t \*sem\_open(const char \*name, int oflag, ...);

**Description:** This function establishes a connection between named semaphores and a task. Following a call to sem\_open() with the semaphore name, the task may reference the semaphore associated with name using the address returned by this call. The semaphore may be used in subsequent calls to sem\_wait(), sem\_trywait(), and sem\_post(). The semaphore remains usable until the semaphore is closed by a successful call to sem\_close().

If a task makes multiple calls to sem\_open() with the same name, then the same semaphore address is returned (provided there have been no calls to sem\_unlink()).

**Input Parameters:**

* name. Semaphore name
* oflag. Semaphore creation options. This may one of the following bit settings:
  + oflag = 0: Connect to the semaphore only if it already exists.
  + oflag = O\_CREAT: Connect to the semaphore if it exists, otherwise create the semaphore.
  + oflag = O\_CREAT with O\_EXCL (O\_CREAT|O\_EXCL): Create a new semaphore unless one of this name already exists.
* ... Optional parameters. NOTE: When the O\_CREAT flag is specified, POSIX requires that a third and fourth parameter be supplied:
  + mode. The mode parameter is of type mode\_t. This parameter is required but not used in the present implementation.
  + value. The value parameter is type unsigned int. The semaphore is created with an initial value of value. Valid initial values for semaphores must be less than or equal to SEM\_VALUE\_MAX (defined in include/limits.h).

**Returned Value:**

* A pointer to sem\_t or SEM\_FAILED if unsuccessful.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name. Differences from the full POSIX implementation include:

* Treatment of links/connections is highly simplified. It is just a counting semaphore.

**2.5.4 sem\_close**

**Function Prototype:**

#include <semaphore.h>

int sem\_close(sem\_t \*sem);

**Description:** This function is called to indicate that the calling task is finished with the specified named semaphore, sem. The sem\_close() deallocates any system resources allocated by the system for this named semaphore.

If the semaphore has not been removed with a call to sem\_unlink(), then sem\_close() has no effect on the named semaphore. However, when the named semaphore has been fully unlinked, the semaphore will vanish when the last task closes it.

Care must be taken to avoid risking the deletion of a semaphore that another calling task has already locked.

**Input Parameters:**

* sem. Semaphore descriptor

**Returned Value:**

* 0 (OK), or -1 (ERROR) if unsuccessful.

**Assumptions/Limitations:**

* Care must be taken to avoid deletion of a semaphore that another task has already locked.
* sem\_close() must not be called with an un-named semaphore.

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.5.5 sem\_unlink**

**Function Prototype:**

#include <semaphore.h>

int sem\_unlink(const char \*name);

**Description:** This function will remove the semaphore named by the input name parameter. If one or more tasks have the semaphore named by name open when sem\_unlink() is called, destruction of the semaphore will be postponed until all references have been destroyed by calls to sem\_close().

**Input Parameters:**

* name. Semaphore name

**Returned Value:**

* 0 (OK), or -1 (ERROR) if unsuccessful.

**Assumptions/Limitations:**

* Care must be taken to avoid deletion of a semaphore that another task has already locked.
* sem\_unlink() must not be called with an un-named semaphore.

**POSIX Compatibility:** Comparable to the POSIX interface of the same name. Differences from the full POSIX implementation include:

* Treatment of links/connections is highly simplified. It is just a counting semaphore.
* Calls to sem\_open() to re-create or re-connect to the semaphore may refer to the same semaphore; POSIX specifies that a new semaphore with the same name should be created after sem\_unlink() is called.

**2.5.6 sem\_wait**

**Function Prototype:**

#include <semaphore.h>

int sem\_wait(sem\_t \*sem);

**Description:** This function attempts to lock the semaphore referenced by sem. If the semaphore as already locked by another task, the calling task will not return until it either successfully acquires the lock or the call is interrupted by a signal.

**Input Parameters:**

* sem. Semaphore descriptor.

**Returned Value:**

* 0 (OK), or -1 (ERROR) is unsuccessful

If sem\_wait returns -1 (ERROR) then the cause of the failure will be indicated by the thread-specific errno. The following lists the possible values for errno:

* EINVAL: Indicates that the sem input parameter is not valid.
* EINTR: Indicates that the wait was interrupt by a signal received by this task. In this case, the semaphore has not be acquired.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.5.7 sem\_timedwait**

**Function Prototype:**

#include <semaphore.h>

#include <time.h>

int sem\_wait(sem\_t \*sem, const struct timespec \*abstime);

**Description:** This function will lock the semaphore referenced by sem as in the sem\_wait() function. However, if the semaphore cannot be locked without waiting for another process or thread to unlock the semaphore by performing a sem\_post() function, this wait will be terminated when the specified timeout expires.

The timeout will expire when the absolute time specified by abstime passes, as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds abstime), or if the absolute time specified by abstime has already been passed at the time of the call. This function attempts to lock the semaphore referenced by sem. If the semaphore as already locked by another task, the calling task will not return until it either successfully acquires the lock or the call is interrupted by a signal.

**Input Parameters:**

* sem. Semaphore descriptor.
* abstime. The absolute time to wait until a timeout is declared.

**Returned Value:**

* 0 (OK), or -1 (ERROR) is unsuccessful

If sem\_wait returns -1 (ERROR) then the cause of the failure will be indicated by the thread-specific errno. The following lists the possible values for errno:

 EINVAL: Indicates that the sem input parameter is not valid or the thread would have blocked, and the abstime parameter specified a nanoseconds field value less than zero or greater than or equal to 1000 million.

 ETIMEDOUT: The semaphore could not be locked before the specified timeout expired.

 EDEADLK: A deadlock condition was detected.

 EINTR: Indicates that the wait was interrupt by a signal received by this task. In this case, the semaphore has not be acquired.

**Assumptions/Limitations:**

**POSIX Compatibility:** Derived from IEEE Std 1003.1d-1999.

**2.5.8 sem\_trywait**

**Function Prototype:**

#include <semaphore.h>

int sem\_trywait(sem\_t \*sem);

**Description:** This function locks the specified semaphore only if the semaphore is currently not locked. In any event, the call returns without blocking.

**Input Parameters:**

* sem. The semaphore descriptor

**Returned Value:**

* 0 (OK) or -1 (ERROR) if unsuccessful

If sem\_wait returns -1 (ERROR) then the cause of the failure will be indicated by the thread-specific errno. The following lists the possible values for errno:

* EINVAL: Indicates that the sem input parameter is not valid.
* EAGAIN: Indicates that the semaphore was not acquired.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.5.9 sem\_post**

**Function Prototype:**

#include <semaphore.h>

int sem\_post(sem\_t \*sem);

**Description:** When a task has finished with a semaphore, it will call sem\_post(). This function unlocks the semaphore referenced by semby performing the semaphore unlock operation.

If the semaphore value resulting from this operation is positive, then no tasks were blocked waiting for the semaphore to become unlocked; The semaphore value is simply incremented.

If the value of the semaphore resulting from this operation is zero, then on of the tasks blocked waiting for the semaphore will be allowed to return successfully from its call to sem\_wait().

**NOTE**: sem\_post() may be called from an interrupt handler.

**Input Parameters:**

* sem. Semaphore descriptor

**Returned Value:**

* 0 (OK) or -1 (ERROR) if unsuccessful.

**Assumptions/Limitations:**. When called from an interrupt handler, it will appear as though the interrupt task is the one that is performing the unlock.

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.5.10 sem\_getvalue**

**Function Prototype:**

#include <semaphore.h>

int sem\_getvalue(sem\_t \*sem, int \*sval);

**Description:** This function updates the location referenced by sval argument to have the value of the semaphore referenced by sem without effecting the state of the semaphore. The updated value represents the actual semaphore value that occurred at some unspecified time during the call, but may not reflect the actual value of the semaphore when it is returned to the calling task.

If sem is locked, the value return by sem\_getvalue() will either be zero or a negative number whose absolute value represents the number of tasks waiting for the semaphore.

**Input Parameters:**

* sem. Semaphore descriptor
* sval. Buffer by which the value is returned

**Returned Value:**

* 0 (OK) or -1 (ERROR) if unsuccessful.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.5.11 sem\_getprotocol**

**Function Prototype:**

#include <nuttx/semaphore.h>

int sem\_getprotocol(FAR const pthread\_mutexattr\_t \*attr, FAR int \*protocol);

**Description:** Return the value of the semaphore protocol attribute.

**Input Parameters:**

* attr. A pointer to the semaphore to be queried
* protocol. The user provided location in which to store the protocol value. May be one of SEM\_PRIO\_NONE, or SEM\_PRIO\_INHERIT, SEM\_PRIO\_PROTECT.

**Returned Value:**

If successful, the sem\_getprotocol() function will return zero (OK). Otherwise, an -1 (ERROR) will be returned and the errno value will be set to indicate the nature of the error.

**Assumptions/Limitations:**

**POSIX Compatibility:** Non-standard NuttX interface. Should not be used in portable code. Analogous to pthread\_muxtexattr\_getprotocol().

**2.5.12 sem\_setprotocol**

**Function Prototype:**

#include <nuttx/semaphore.h>

int sem\_setprotocol(FAR pthread\_mutexattr\_t \*attr, int protocol);

**Description:** Set semaphore protocol attribute. See the paragraph Locking versus Signaling Semaphores for some important information about the use of this interface.

**Input Parameters:**

* attr. A pointer to the semaphore to be modified
* protocol. The new protocol to use. One of SEM\_PRIO\_NONE, or SEM\_PRIO\_INHERIT, SEM\_PRIO\_PROTECT.  SEM\_PRIO\_INHERIT is supported only if  CONFIG\_PRIORITY\_INHERITANCE is defined;  SEM\_PRIO\_PROTECT is not currently supported in any configuration.

**Returned Value:**

If successful, the sem\_getprotocol() function will return zero (OK). Otherwise, an -1 (ERROR) will be returned and the errno value will be set to indicate the nature of the error.

**Assumptions/Limitations:**

**POSIX Compatibility:** Non-standard NuttX interface. Should not be used in portable code. Analogous to pthread\_muxtexattr\_setprotocol().

|  |
| --- |
| **2.6 Clocks and Timers** |

* 2.6.1 clock\_settime
* 2.6.2 clock\_gettime
* 2.6.3 clock\_getres
* 2.6.4 mktime
* 2.6.5 gmtime
* 2.6.6 localtime
* 2.6.7 asctime
* 2.6.8 ctime
* 2.6.9 gmtime\_r
* 2.6.10 localtime\_r
* 2.6.11 asctime\_r
* 2.6.12 ctime\_r
* 2.6.13 timer\_create
* 2.6.14 timer\_delete
* 2.6.15 timer\_settime
* 2.6.16 timer\_gettime
* 2.6.17 timer\_getoverrun
* 2.6.18 gettimeofday

**2.6.1 clock\_settime**

**Function Prototype:**

#include <time.h>

int clock\_settime(clockid\_t clockid, const struct timespec \*tp);

**Description:**

**Input Parameters:**

* parm.

**Returned Value:** If successful, the clock\_settime() function will return zero (OK). Otherwise, an non-zero error number will be returned to indicate the error:

**2.6.2 clock\_gettime**

**Function Prototype:**

#include <time.h>

int clock\_gettime(clockid\_t clockid, struct timespec \*tp);

**Description:**

**Input Parameters:**

* parm.

**Returned Value:** If successful, the clock\_gettime() function will return zero (OK). Otherwise, an non-zero error number will be returned to indicate the error:

**2.6.3 clock\_getres**

**Function Prototype:**

#include <time.h>

int clock\_getres(clockid\_t clockid, struct timespec \*res);

**Description:**

**Input Parameters:**

* parm.

**Returned Value:** If successful, the clock\_getres() function will return zero (OK). Otherwise, an non-zero error number will be returned to indicate the error:

**2.6.4 mktime**

**Function Prototype:**

#include <time.h>

time\_t mktime(struct tm \*tp);

**Description:**

**Input Parameters:**

* parm.

**Returned Value:** If successful, the mktime() function will return zero (OK). Otherwise, an non-zero error number will be returned to indicate the error:

**2.6.5 gmtime**

**Function Prototype:**

#include <time.h>

FAR struct tm \*gmtime(FAR const time\_t \*timep);

**Description:** Represents GMT date/time in a type struct tm. This function is not re-entrant.

**Input Parameters:**

* timep. Represents GMT calendar time. This is an absolute time value representing the number of seconds elapsed since 00:00:00 on January 1, 1970, Coordinated Universal Time (UTC).

**Returned Value:** If successful, the gmtime() function will return the pointer to a statically defined instance of struct tm. Otherwise, a NULL will be returned to indicate the error:

**2.6.6 localtime**

#include <time.h>

#ifdef CONFIG\_LIBC\_LOCALTIME

# define localtime(c) gmtime(c)

#else

FAR struct tm \*localtime(FAR const time\_t \*timep);

#endif

**Description:** Represents local date/time in a type struct tm. This function is not re-entrant.

**Input Parameters:**

* timep. Represents GMT calendar time. This is an absolute time value representing the number of seconds elapsed since 00:00:00 on January 1, 1970, Coordinated Universal Time (UTC).

**Returned Value:** **Returned Value:** If successful, the localtime() function will return the pointer to a statically defined instance of struct tm. Otherwise, a NULL will be returned to indicate the error:

**2.6.7 asctime**

**Function Prototype:**

#include <time.h>

#if defined(CONFIG\_LIBC\_LOCALTIME) || defined(CONFIG\_TIME\_EXTENDED)

FAR char \*asctime(FAR const struct tm \*tp);

#endif

**Description:** asctime() convert the time provided in a struct tm to a string representation. asctime() is not re-entrant.

**Input Parameters:**

* tp. Pointer to the time to be converted.

**Returned Value:** If successful, the asctime() function will return a pointer to a statically defined string holding the converted time. Otherwise, a NULL will be returned to indicate the error.

**2.6.8 ctime**

**Function Prototype:**

#include <time.h>

#if defined(CONFIG\_LIBC\_LOCALTIME) || defined(CONFIG\_TIME\_EXTENDED)

FAR char \*ctime(FAR const time\_t \*timep);

#endif

**Description:** ctime() converts the time provided in seconds since the epoch to a string representation. ctime() is not re-entrant.

**Input Parameters:**

* timep. The current time represented as seconds since the epoch.

**Returned Value:** If successful, the ctime() function will return the pointer to the converted string. Otherwise, a NULL will be returned to indicate the error.

**2.6.9 gmtime\_r**

**Function Prototype:**

#include <time.h>

struct tm \*gmtime\_r(const time\_t \*timep, struct tm \*result);

**Description:** Represents GMT date/time in a type struct tm. This function is re-entrant.

**Input Parameters:**

* timep. Represents GMT calendar time. This is an absolute time value representing the number of seconds elapsed since 00:00:00 on January 1, 1970, Coordinated Universal Time (UTC).
* result. A user-provided buffer to receive the converted time structure.

**Returned Value:** If successful, the gmtime\_r() function will return the pointer, result, provided by the caller. Otherwise, a NULL will be returned to indicate the error:

**2.6.10 localtime\_r**

#include <time.h>

#ifdef CONFIG\_LIBC\_LOCALTIME

# define localtime\_r(c,r) gmtime\_r(c,r)

#else

FAR struct tm \*localtime\_r(FAR const time\_t \*timep, FAR struct tm \*result);

#endif

**Description:** Represents local date/time in a type struct tm. This function is re-entrant.

**Input Parameters:**

* timep. Represents GMT calendar time. This is an absolute time value representing the number of seconds elapsed since 00:00:00 on January 1, 1970, Coordinated Universal Time (UTC).
* result. A user-provided buffer to receive the converted time structure.

**Returned Value:** **Returned Value:** If successful, the localtime\_r() function will return the pointer, result, provided by the caller. Otherwise, a NULL will be returned to indicate the error:

**2.6.11 asctime\_r**

**Function Prototype:**

#include <time.h>

#if defined(CONFIG\_LIBC\_LOCALTIME) || defined(CONFIG\_TIME\_EXTENDED)

FAR char \*asctime\_r(FAR const struct tm \*tp, FAR char \*buf);

#endif

**Description:** asctime\_r() converts the time provided in a struct tm to a string representation. asctime-r() is re-entrant.

**Input Parameters:**

* tp. Pointer to the time to be converted.
* buf. The user provider buffer. of size >= 26 characters, to receive the converted time.

**Returned Value:** If successful, the asctime\_r() function will return the pointer, buf, provided by the caller. Otherwise, a NULL will be returned to indicate the error.

**2.6.12 ctime\_r**

**Function Prototype:**

#include <time.h>

#if defined(CONFIG\_LIBC\_LOCALTIME) || defined(CONFIG\_TIME\_EXTENDED)

FAR char \*ctime\_r(FAR const time\_t \*timep, FAR char \*buf);

#endif

**Description:** ctime\_r() converts the time provided in seconds since the epoch to a string representation. ctime() is re-entrant.

**Input Parameters:**

* timep. The current time represented as seconds since the epoch.
* buf. The user provider buffer. of size >= 26 characters, to receive the converted time.

**Returned Value:** If successful, the ctime\_r() function will return the pointer, buf, provided by the caller. Otherwise, a NULL will be returned to indicate the error.

**2.6.13 timer\_create**

**Function Prototype:**

#include <time.h>

int timer\_create(clockid\_t clockid, struct sigevent \*evp, timer\_t \*timerid);

**Description:** The  timer\_create() function creates per-thread timer using the specified clock, clock\_id, as the timing base. The timer\_create() function returns, in the location referenced by timerid, a timer ID of type timer\_t used to identify the timer in timer requests. This timer ID is unique until the timer is deleted. The particular clock, clock\_id, is defined in <time.h>. The timer whose ID is returned will be in a disarmed state upon return from timer\_create().

The evp argument, if non-NULL, points to a sigevent structure. This structure is allocated by the called and defines the asynchronous notification to occur. If the evp argument is NULL, the effect is as if the evp argument pointed to a sigevent structure with the sigev\_notify member having the value SIGEV\_SIGNAL, the sigev\_signo having a default signal number, and the sigev\_value member having the value of the timer ID.

Each implementation defines a set of clocks that can be used as timing bases for per-thread timers. All implementations will support a clock\_id of CLOCK\_REALTIME.

**Input Parameters:**

* clockid. Specifies the clock to use as the timing base. Must be CLOCK\_REALTIME.
* evp. Refers to a user allocated sigevent structure that defines the asynchronous notification. evp may be NULL (see above).
* timerid. The pre-thread timer created by the call to timer\_create().

**Returned Value:** If the call succeeds, timer\_create() will return 0 (OK) and update the location referenced by timerid to a timer\_t, which can be passed to the other per-thread timer calls. If an error occurs, the function will return a value of -1 (ERROR) and set [errno](http://nuttx.org/Documentation/NuttxUserGuide.html#ErrnoAccess) to indicate the error.

* EAGAIN. The system lacks sufficient signal queuing resources to honor the request.
* EAGAIN. The calling process has already created all of the timers it is allowed by this implementation.
* EINVAL. The specified clock ID is not defined.
* ENOTSUP. The implementation does not support the creation of a timer attached to the CPU-time clock that is specified by clock\_id and associated with a thread different thread invoking timer\_create().

**POSIX Compatibility:** Comparable to the POSIX interface of the same name. Differences from the full POSIX implementation include:

* Only CLOCK\_REALTIME is supported for the clockid argument.

**2.6.14 timer\_delete**

**Function Prototype:**

#include <time.h>

int timer\_delete(timer\_t timerid);

**Description:** The timer\_delete() function deletes the specified timer, timerid, previously created by the timer\_create() function. If the timer is armed when timer\_delete() is called, the timer will be automatically disarmed before removal. The disposition of pending signals for the deleted timer is unspecified.

**Input Parameters:**

* timerid. The pre-thread timer, previously created by the call to timer\_create(), to be deleted.

**Returned Value:** If successful, the timer\_delete() function will return zero (OK). Otherwise, the function will return a value of -1 (ERROR) and set errno to indicate the error:

* EINVAL. The timer specified timerid is not valid.

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.6.15 timer\_settime**

**Function Prototype:**

#include <time.h>

int timer\_settime(timer\_t timerid, int flags, const struct itimerspec \*value,

struct itimerspec \*ovalue);

**Description:** The timer\_settime() function sets the time until the next expiration of the timer specified by timerid from the it\_valuemember of the value argument and arm the timer if the it\_value member of value is non-zero. If the specified timer was already armed when timer\_settime() is called, this call will reset the time until next expiration to the value specified. If the it\_value member of value is zero, the timer will be disarmed. The effect of disarming or resetting a timer with pending expiration notifications is unspecified.

If the flag TIMER\_ABSTIME is not set in the argument flags, timer\_settime() will behave as if the time until next expiration is set to be equal to the interval specified by the it\_value member of value. That is, the timer will expire in it\_value nanoseconds from when the call is made. If the flag TIMER\_ABSTIME is set in the argument flags, timer\_settime() will behave as if the time until next expiration is set to be equal to the difference between the absolute time specified by the it\_value member of value and the current value of the clock associated with timerid. That is, the timer will expire when the clock reaches the value specified by the it\_value member of value. If the specified time has already passed, the function will succeed and the expiration notification will be made.

The reload value of the timer will be set to the value specified by the it\_interval member of value. When a timer is armed with a non-zero it\_interval, a periodic (or repetitive) timer is specified.

Time values that are between two consecutive non-negative integer multiples of the resolution of the specified timer will be rounded up to the larger multiple of the resolution. Quantization error will not cause the timer to expire earlier than the rounded time value.

If the argument ovalue is not NULL, the timer\_settime() function will store, in the location referenced by ovalue, a value representing the previous amount of time before the timer would have expired, or zero if the timer was disarmed, together with the previous timer reload value. Timers will not expire before their scheduled time.

**NOTE:**At present, the ovalue argument is ignored.

**Input Parameters:**

* timerid. The pre-thread timer, previously created by the call to timer\_create(), to be be set.
* flags. Specify characteristics of the timer (see above)
* value. Specifies the timer value to set
* ovalue. A location in which to return the time remaining from the previous timer setting (ignored).

**Returned Value:** If the timer\_gettime() succeeds, a value of 0 (OK) will be returned. If an error occurs, the value -1 (ERROR) will be returned, and errno set to indicate the error.

* EINVAL. The timerid argument does not correspond to an ID returned by timer\_create() but not yet deleted by timer\_delete().
* EINVAL. A value structure specified a nanosecond value less than zero or greater than or equal to 1000 million, and the it\_value member of that structure did not specify zero seconds and nanoseconds.

**POSIX Compatibility:** Comparable to the POSIX interface of the same name. Differences from the full POSIX implementation include:

* The ovalue argument is ignored.

**2.6.16 timer\_gettime**

**Function Prototype:**

#include <time.h>

int timer\_gettime(timer\_t timerid, struct itimerspec \*value);

**Description:** The timer\_gettime() function will store the amount of time until the specified timer, timerid, expires and the reload value of the timer into the space pointed to by the value argument. The it\_value member of this structure will contain the amount of time before the timer expires, or zero if the timer is disarmed. This value is returned as the interval until timer expiration, even if the timer was armed with absolute time. The it\_interval member of value will contain the reload value last set by timer\_settime().

Due to the asynchronous operation of this function, the time reported by this function could be significantly more than that actual time remaining on the timer at any time.

**Input Parameters:**

* timerid. Specifies pre-thread timer, previously created by the call to timer\_create(), whose remaining count will be returned.

**Returned Value:** If successful, the timer\_gettime() function will return zero (OK). Otherwise, an non-zero error number will be returned to indicate the error:

* EINVAL. The timerid argument does not correspond to an ID returned by timer\_create() but not yet deleted by timer\_delete().

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.6.17 timer\_getoverrun**

**Function Prototype:**

#include <time.h>

int timer\_getoverrun(timer\_t timerid);

**Description:** Only a single signal will be queued to the process for a given timer at any point in time. When a timer for which a signal is still pending expires, no signal will be queued, and a timer overrun will occur. When a timer expiration signal is delivered to or accepted by a process, if the implementation supports the  *Realtime Signals Extension*, the timer\_getoverrun() function will return the timer expiration overrun count for the specified timer. The overrun count returned contains the number of extra timer expirations that occurred between the time the signal was generated (queued) and when it was delivered or accepted, up to but not including an implementation-defined maximum of DELAYTIMER\_MAX. If the number of such extra expirations is greater than or equal to DELAYTIMER\_MAX, then the overrun count will be set to DELAYTIMER\_MAX. The value returned by timer\_getoverrun() will apply to the most recent expiration signal delivery or acceptance for the timer. If no expiration signal has been delivered for the timer, or if the *Realtime Signals Extension* is not supported, the return value of timer\_getoverrun() is unspecified.

**NOTE:** This interface is not currently implemented in NuttX.

**Input Parameters:**

* timerid. Specifies pre-thread timer, previously created by the call to timer\_create(), whose overrun count will be returned.

**Returned Value:** If the timer\_getoverrun() function succeeds, it will return the timer expiration overrun count as explained above. timer\_getoverrun() will fail if:

* EINVAL. The timerid argument does not correspond to an ID returned by timer\_create() but not yet deleted by timer\_delete().

**POSIX Compatibility:** Comparable to the POSIX interface of the same name. Differences from the full POSIX implementation include:

* This interface is not currently implemented by NuttX.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.6.18 gettimeofday**

**Function Prototype:**

#include <sys/time.h>

int gettimeofday(struct timeval \*tp, void \*tzp);

**Description:** This implementation of gettimeofday() is simply a thin wrapper around clock\_gettime(). It simply calls clock\_gettime() using the CLOCK\_REALTIME timer and converts the result to the required struct timeval.

**Input Parameters:**

* tp. The current time will be returned to this user provided location.
* tzp. A reference to the timezone -- *IGNORED*.

**Returned Value:** See clock\_gettime().

|  |
| --- |
| **2.7 Signal Interfaces** |

**Tasks and Signals**. NuttX provides signal interfaces for tasks and pthreads. Signals are used toalter the flow control of tasks by communicating asynchronous events within or between task contexts. Any task or interrupt handler can post (or send) a signal to a particular task using its task ID. The task being signaled will execute task-specified signal handler function the next time that the task has priority. The signal handler is a user-supplied function that is bound to a specific signal and performs whatever actions are necessary whenever the signal is received.

There are no predefined actions for any signal. The default action for all signals (i.e., when no signal handler has been supplied by the user) is to ignore the signal. In this sense, all NuttX are *real time* signals.

Tasks may also suspend themselves and wait until a signal is received.

**Tasks Groups**. NuttX supports both tasks and pthreads. The primary difference between tasks and pthreads is the tasks are much more independent. Tasks can create pthreads and those pthreads will share the resources of the task. The main task and its children pthreads together are referred as a *task group*. A task group is used in NuttX to emulate a POSIX *process*.

**NOTE:** Behavior of features related to *task group*s depend of NuttX configuration settings. See the [NuttX Threading Wiki](http://www.nuttx.org/doku.php?id=wiki:nxinternal:nxtasking) page and the [Tasks vs. Threads FAQ](http://www.nuttx.org/doku.php?id=wiki:nxinternal:tasksnthreads) for additional information on tasks and threads in NuttX.

**Signaling Multi-threaded Task Groups**. The behavior of signals in the multi-thread task group is complex. NuttX emulates a process model with task groups and follows the POSIX rules for signaling behavior. Normally when you signal the task group you would signal using the task ID of the main task that created the group (in practice, a different task should not know the IDs of the internal threads created within the task group); that ID is remembered by the task group (even if the main task thread exits).

Here are some of the things that should happen when you signal a multi-threaded task group:

* If a task group receives a signal then one and only one indeterminate thread in the task group which is not blocking the signal will receive the signal.
* If a task group receives a signal and more than one thread is waiting on that signal, then one and only one indeterminate thread out of that waiting group will receive the signal.

You can mask out that signal using ''sigprocmask()'' (or ''pthread\_sigmask()''). That signal will then be effectively disabled and will never be received in those threads that have the signal masked. On creation of a new thread, the new thread will inherit the signal mask of the parent thread that created it. So you if block signal signals on one thread then create new threads, those signals will also be blocked in the new threads as well.

You can control which thread receives the signal by controlling the signal mask. You can, for example, create a single thread whose sole purpose it is to catch a particular signal and respond to it: Simply block the signal in the main task; then the signal will be blocked in all of the pthreads in the group too. In the one "signal processing" pthread, enable the blocked signal. This thread will then be only thread that will receive the signal.

**Signal Interfaces**. The following signal handling interfaces are provided by NuttX:

* 2.7.1 sigemptyset
* 2.7.2 sigfillset
* 2.7.3 sigaddset
* 2.7.4 sigdelset
* 2.7.5 sigismember
* 2.7.6 sigaction
* 2.7.7 sigignore
* 2.7.8 sigset
* 2.7.9 sigprocmask
* 2.7.10 sighold
* 2.7.11 sigrelse
* 2.7.12 sigpending
* 2.7.13 sigsuspend
* 2.7.14 sigpause
* 2.7.15 sigwaitinfo
* 2.7.16 sigtimedwait
* 2.7.17 sigqueue
* 2.7.18 kill
* 2.7.19 pause

**2.7.1 sigemptyset**

**Function Prototype:**

#include <signal.h>

int sigemptyset(sigset\_t \*set);

**Description:** This function initializes the signal set specified by set such that all signals are excluded.

**Input Parameters:**

* set. Signal set to initialize.

**Returned Value:**

* 0 (OK), or -1 (ERROR) if the signal set cannot be initialized.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.7.2 sigfillset**

**Function Prototype:**

#include <signal.h>

int sigfillset(sigset\_t \*set);

**Description:** This function initializes the signal set specified by set such that all signals are included.

**Input Parameters:**

* set. Signal set to initialize

**Returned Value:**

* 0 (OK), or -1 (ERROR) if the signal set cannot be initialized.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.7.3 sigaddset**

**Function Prototype:**

#include <signal.h>

int sigaddset(sigset\_t \*set, int signo);

**Description:** This function adds the signal specified by signo to the signal set specified by set.

**Input Parameters:**

* set. Signal set to add signal to
* signo. Signal to add

**Returned Value:**

* 0 (OK), or -1 (ERROR) if the signal number is invalid.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.7.4 sigdelset**

**Function Prototype:**

#include <signal.h>

int sigdelset(sigset\_t \*set, int signo);

**Description:** This function deletes the signal specified by signo from the signal set specified by set.

**Input Parameters:**

* set. Signal set to delete the signal from
* signo. Signal to delete

**Returned Value:**

* 0 (OK), or -1 (ERROR) if the signal number is invalid.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.7.5 sigismember**

**Function Prototype:**

#include <signal.h>

int sigismember(const sigset\_t \*set, int signo);

**Description:** This function tests whether the signal specified by signo is a member of the set specified by set.

**Input Parameters:**

* set. Signal set to test
* signo. Signal to test for

**Returned Value:**

* 1 (TRUE), if the specified signal is a member of the set,
* 0 (OK or FALSE), if it is not, or
* -1 (ERROR) if the signal number is invalid.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.7.6 sigaction**

**Function Prototype:**

#include <signal.h>

int sigaction(int signo, const struct sigaction \*act,

struct sigaction \*oact);

**Description:** This function allows the calling task to examine and/or specify the action to be associated with a specific signal.

The structure sigaction, used to describe an action to be taken, is defined to include the following members:

* sa\_u.sa\_handler. A pointer to a signal-catching function.
* sa\_u.sa\_sigaction. An alternative form for the signal catching function.
* sa\_mask. Additional set of signals to be blocked during execution of the signal-catching function.
* sa\_flags: Special flags to affect behavior of a signal.

If the argument act is not NULL, it points to a structure specifying the action to be associated with the specified signal. If the argument oact is not NULL, the action previously associated with the signal is stored in the location pointed to by the argument oact. If the argument act is NULL, signal handling is unchanged by this function call; thus, the call can be used to inquire about the current handling of a given signal.

When a signal is caught by a signal-catching function installed by the sigaction() function, a new signal mask is calculated and installed for the duration of the signal-catching function. This mask is formed by taking the union of the current signal mask and the value of the sa\_mask for the signal being delivered, and then including the signal being delivered. If and when the signal handler returns, the original signal mask is restored.

Signal catching functions execute in the same address environment as the task that called sigaction() to install the signal-catching function.

Once an action is installed for a specific signal, it remains installed until another action is explicitly requested by another call to sigaction().

**Input Parameters:**

* sig. Signal of interest
* act. Location of new handler
* oact. Location to store old handler

**Returned Value:**

* 0 (OK), or -1 (ERROR) if the signal number is invalid.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name. Differences from the POSIX implementation include:

* There are no default actions so the special value SIG\_DFL is treated like SIG\_IGN.
* All sa\_flags in struct sigaction of act input are ignored (all treated like SA\_SIGINFO). The one exception is if CONFIG\_SCHED\_CHILD\_STATUS is defined; then SA\_NOCLDWAIT is supported but only for SIGCHLD.

**2.7.7 sigignore**

**Function Prototype:**

#include <signal.h>

int sigignore(int signo);

**Description:** The sigignore() function will set the disposition of signo to SIG\_IGN.

**Input Parameters:**

* signo. The signal number to act on

**Returned Value:**

* Zero is returned upon successful completion, otherwise -1 (ERROR) is returned with the <code<>errno set appropriately. The errno value of EINVAL, for example, would indicate that signo argument is not a valid signal number.</code<>

**2.7.8 sigset**

**Function Prototype:**

#include <signal.h>

void (\*sigset(int signo, void (\*disp)(int)))(int);

**Description:** The sigset() function will modify signal dispositions. The signo argument specifies the signal. The disp argument specifies the signal's disposition, which may be SIG\_DFL, SIG\_IGN, or the address of a signal handler. If disp is the address of a signal handler, the system will add signo to the calling process' signal mask before executing the signal handler; when the signal handler returns, the system will restore the calling process' signal mask to its state prior to the delivery of the signal. signo will be removed from the calling process' signal mask.

NOTE: The value SIG\_HOLD for disp is not currently supported.

**Input Parameters:**

* signo. The signal number to operate on
* disp. The new disposition of the signal

**Returned Value:**

* Upon successful completion, sigset() will the previous disposition of the signal. Otherwise, SIG\_ERR will be returned and errnoset to indicate the error.

**2.7.9 sigprocmask**

**Function Prototype:**

#include <signal.h>

int sigprocmask(int how, const sigset\_t \*set, sigset\_t \*oset);

**Description:** This function allows the calling task to examine and/or change its signal mask. If the set is not NULL, then it points to a set of signals to be used to change the currently blocked set. The value of how indicates the manner in which the set is changed.

If there are any pending unblocked signals after the call to sigprocmask(), those signals will be delivered before sigprocmask() returns.

If sigprocmask() fails, the signal mask of the task is not changed.

**Input Parameters:**

* how. How the signal mast will be changed:
  + SIG\_BLOCK. The resulting set is the union of the current set and the signal set pointed to by the set input parameter.
  + SIG\_UNBLOCK. The resulting set is the intersection of the current set and the complement of the signal set pointed to by the set input parameter.
  + SIG\_SETMASK. The resulting set is the signal set pointed to by the set input parameter.
* set. Location of the new signal mask
* oset. Location to store the old signal mask

**Returned Value:**

* 0 (OK), or -1 (ERROR) if how is invalid.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.7.10 sighold**

**Function Prototype:**

#include <signal.h>

int sighold(int signo);

**Description:** The sighold() function will add signo to the calling process' signal mask

**Input Parameters:**

* signo. Identifies the signal to be blocked.

**Returned Value:**

* Zero is returned upon successful completion, otherwise -1 (ERROR) is returned with the <code<>errno set appropriately. The errno value of EINVAL, for example, would indicate that signo argument is not a valid signal number.</code<>

**2.7.11 sigrelse**

**Function Prototype:**

#include <signal.h>

int sigrelse(int signo);

**Description:** The sighold() function will remove signo from the calling process' signal mask

**Input Parameters:**

* signo. Identifies the signal to be unblocked.

**Returned Value:**

* Zero is returned upon successful completion, otherwise -1 (ERROR) is returned with the <code<>errno set appropriately. The errno value of EINVAL, for example, would indicate that signo argument is not a valid signal number.</code<>

**2.7.12 sigpending**

**Function Prototype:**

#include <signal.h>

int sigpending(sigset\_t \*set);

**Description:** This function stores the returns the set of signals that are blocked for delivery and that are pending for the calling task in the space pointed to by set.

If the task receiving a signal has the signal blocked via its sigprocmask, the signal will pend until it is unmasked. Only one pending signal (for a given signo) is retained by the system. This is consistent with POSIX which states: "If a subsequent occurrence of a pending signal is generated, it is implementation defined as to whether the signal is delivered more than once."

**Input Parameters:**

* set. The location to return the pending signal set.

**Returned Value:**

* 0 (OK) or -1 (ERROR)

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.7.13 sigsuspend**

**Function Prototype:**

#include <signal.h>

int sigsuspend(const sigset\_t \*set);

**Description:** The sigsuspend() function replaces the signal mask with the set of signals pointed to by the argument set and then suspends the task until delivery of a signal to the task.

If the effect of the set argument is to unblock a pending signal, then no wait is performed.

The original signal mask is restored when sigsuspend() returns.

Waiting for an empty signal set stops a task without freeing any resources (a very bad idea).

**Input Parameters:**

* set. The value of the signal **mask** to use while suspended.

**Returned Value:**

* -1 (ERROR) always

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name. Differences from the POSIX specification include:

* POSIX does not indicate that the original signal mask is restored.
* POSIX states that sigsuspend() "suspends the task until delivery of a signal whose action is either to execute a signal-catching function or to terminate the task." Only delivery of the signal is required in the present implementation (even if the signal is ignored).

**2.7.14 sigpause**

**Function Prototype:**

#include <signal.h>

int sigpause(int signo);

**Description:** The sigpause()) function will remove signo) from the calling process' signal mask and suspend the calling process until a signal is received. The sigpause()) function will restore the process' signal mask to its original state before returning.

**Input Parameters:**

* set. Identifies the signal to be unblocked while waiting.

**Returned Value:**

* sigpause always returns -1 (ERROR). On a successful wait for a signal, the errno will be set to EINTR.

**2.7.15 sigwaitinfo**

**Function Prototype:**

#include <signal.h>

int sigwaitinfo(const sigset\_t \*set, struct siginfo \*info);

**Description:** This function is equivalent to sigtimedwait() with a NULL timeout parameter. (see below).

**Input Parameters:**

* set. The set of pending signals to wait for.
* info. The returned signal values

**Returned Value:**

* Signal number that cause the wait to be terminated, otherwise -1 (ERROR) is returned.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.7.16 sigtimedwait**

**Function Prototype:**

#include <signal.h>

int sigtimedwait(const sigset\_t \*set, struct siginfo \*info,

const struct timespec \*timeout);

**Description:** This function selects the pending signal set specified by the argument set. If multiple signals are pending in set, it will remove and return the lowest numbered one. If no signals in set are pending at the time of the call, the calling task will be suspended until one of the signals in set becomes pending OR until the task interrupted by an unblocked signal OR until the time interval specified by timeout (if any), has expired. If timeout is NULL, then the timeout interval is forever.

If the info argument is non-NULL, the selected signal number is stored in the si\_signo member and the cause of the signal is store in the si\_code member. The content of si\_value is only meaningful if the signal was generated by sigqueue(). The following values for si\_code are defined in signal.h:

* SI\_USER. Signal sent from kill, raise, or abort
* SI\_QUEUE. Signal sent from sigqueue
* SI\_TIMER. Signal is result of timer expiration
* SI\_ASYNCIO. Signal is the result of asynchronous IO completion
* SI\_MESGQ. Signal generated by arrival of a message on an empty message queue.

**Input Parameters:**

* set. The set of pending signals to wait for.
* info. The returned signal values
* timeout. The amount of time to wait

**Returned Value:**

* Signal number that cause the wait to be terminated, otherwise -1 (ERROR) is returned.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name. Differences from the POSIX interface include:

* Values for si\_codes differ
* No mechanism to return cause of ERROR. (It can be inferred from si\_code in a non-standard way).
* POSIX states that "If no signal is pending at the time of the call, the calling task will be suspended until one or more signals in set become pending or until it is interrupted by an unblocked, *caught* signal." The present implementation does not require that the unblocked signal be caught; the task will be resumed even if the unblocked signal is ignored.

**2.7.17 sigqueue**

**Function Prototype:**

#include <signal.h>

int sigqueue (int tid, int signo, union sigval value);

**Description:** This function sends the signal specified by signo with the signal parameter value to the task specified by tid.

If the receiving task has the signal blocked via its sigprocmask, the signal will pend until it is unmasked. Only one pending signal (for a given signo) is retained by the system. This is consistent with POSIX which states: "If a subsequent occurrence of a pending signal is generated, it is implementation defined as to whether the signal is delivered more than once."

**Input Parameters:**

* tid. ID of the task to receive signal
* signo. Signal number
* value. Value to pass to task with signal

**Returned Value:**

* On success (at least one signal was sent), zero (OK) is returned. On error, -1 (ERROR) is returned, and errno is set appropriately.
  + EGAIN. The limit of signals which may be queued has been reached.
  + EINVAL. signo was invalid.
  + EPERM. The task does not have permission to send the signal to the receiving process.
  + ESRCH. No process has a PID matching pid.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name. Differences from the POSIX interface include:

* Default action is to ignore signals.
* Signals are processed one at a time in order
* POSIX states that, "If signo is zero (the null signal), error checking will be performed but no signal is actually sent." There is no null signal in the present implementation; a zero signal will be sent.

**2.7.18 kill**

**Function Prototype:**

#include <sys/types.h>

#include <signal.h>

int kill(pid\_t pid, int sig);

**Description:** The kill() system call can be used to send any signal to any task.

If the receiving task has the signal blocked via its sigprocmask, the signal will pend until it is unmasked. Only one pending signal (for a given signo) is retained by the system. This is consistent with POSIX which states: "If a subsequent occurrence of a pending signal is generated, it is implementation defined as to whether the signal is delivered more than once."

**Input Parameters:**

* pid. The id of the task to receive the signal. The POSIX kill() specification encodes process group information as zero and negative pid values. Only positive, non-zero values of pid are supported by this implementation. ID of the task to receive signal
* signo. The signal number to send. If signo is zero, no signal is sent, but all error checking is performed.

**Returned Value:**

* OK or ERROR

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name. Differences from the POSIX interface include:

* Default action is to ignore signals.
* Signals are processed one at a time in order
* Sending of signals to 'process groups' is not supported in NuttX.

**2.7.19 pause**

**Function Prototype:**

#include <unistd.h>

int pause(void);

**Description:** The pause() function will suspend the calling thread until delivery of a non-blocked signal.

**Input Parameters:**

* *None*

**Returned Value:** Since pause() suspends thread execution indefinitely unless interrupted a signal, there is no successful completion return value. A value of -1 (ERROR will always be returned and errno set to indicate the error (EINTR).

**Assumptions/Limitations:**

**POSIX Compatibility:** In the POSIX description of this function is the pause() function will suspend the calling thread until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process. This implementation only waits for any non-blocked signal to be received.

|  |
| --- |
| **2.8 Pthread Interfaces** |

NuttX does not support *processes* in the way that, say, Linux does. NuttX only supports simple threads or tasks running within the same address space. However, NuttX does support the concept of a *task group*. A task group is the functional analog of a process: It is a group that consists of the main task thread and of all of the pthreads created by the main thread or any of the other pthreads within the task group. Members of a task group share certain resources such as environment variables, file descriptors, FILE streams, sockets, pthread keys and open message queues.

**NOTE:** Behavior of features related to *task group*s depend of NuttX configuration settings. See the [NuttX Threading Wiki](http://www.nuttx.org/doku.php?id=wiki:nxinternal:nxtasking) page and the [Tasks vs. Threads FAQ](http://www.nuttx.org/doku.php?id=wiki:nxinternal:tasksnthreads) for additional information on tasks and threads in NuttX.

The following pthread interfaces are supported in some form by NuttX:

**pthread control interfaces**. Interfaces that allow you to create and manage pthreads.

* + 2.8.1 pthread\_attr\_init
  + 2.8.2 pthread\_attr\_destroy
  + 2.8.3 pthread\_attr\_setschedpolicy
  + 2.8.4 pthread\_attr\_getschedpolicy
  + 2.8.5 pthread\_attr\_setschedparam
  + 2.8.6 pthread\_attr\_getschedparam
  + 2.8.7 pthread\_attr\_setinheritsched
  + 2.8.8 pthread\_attr\_getinheritsched
  + 2.8.9 pthread\_attr\_setstacksize
  + 2.8.10 pthread\_attr\_getstacksize
  + 2.8.11 pthread\_create
  + 2.8.12 pthread\_detach
  + 2.8.13 pthread\_exit
  + 2.8.14 pthread\_cancel
  + 2.8.15 pthread\_setcancelstate
  + 2.8.16 pthread\_setcanceltype
  + 2.8.17 pthread\_testcancel
  + 2.8.18 pthread\_cleanup\_pop
  + 2.8.19 pthread\_cleanup\_push
  + 2.8.20 pthread\_join
  + 2.8.21 pthread\_yield
  + 2.8.22 pthread\_self
  + 2.8.23 pthread\_getschedparam
  + 2.8.24 pthread\_setschedparam

**Thread Specific Data**. These interfaces can be used to create pthread *keys* and then to access thread-specific data using these keys. Each *task group* has its own set of pthread keys. NOTES: (1) pthread keys create in one *task group* are not accessible in other task groups. (2) The main task thread does not had thread-specific data.

* + 2.8.25 pthread\_key\_create
  + 2.8.26 pthread\_setspecific
  + 2.8.27 pthread\_getspecific
  + 2.8.28 pthread\_key\_delete

**pthread Mutexes**.

* + 2.8.29 pthread\_mutexattr\_init
  + 2.8.30 pthread\_mutexattr\_destroy
  + 2.8.31 pthread\_mutexattr\_getpshared
  + 2.8.32 pthread\_mutexattr\_setpshared
  + 2.8.33 pthread\_mutexattr\_gettype
  + 2.8.34 pthread\_mutexattr\_settype
  + 2.8.35 pthread\_mutexattr\_getprotocol
  + 2.8.36 pthread\_mutexattr\_setprotocol
  + 2.8.37 pthread\_mutex\_init
  + 2.8.38 pthread\_mutex\_destroy
  + 2.8.39 pthread\_mutex\_lock
  + 2.8.40 pthread\_mutex\_trylock
  + 2.8.41 pthread\_mutex\_unlock

**Condition Variables**.

* + 2.8.42 pthread\_condattr\_init
  + 2.8.43 pthread\_condattr\_destroy
  + 2.8.44 pthread\_cond\_init
  + 2.8.45 pthread\_cond\_destroy
  + 2.8.46 pthread\_cond\_broadcast
  + 2.8.47 pthread\_cond\_signal
  + 2.8.48 pthread\_cond\_wait
  + 2.8.49 pthread\_cond\_timedwait

**Barriers**.

* + 2.8.50 pthread\_barrierattr\_init
  + 2.8.51 pthread\_barrierattr\_destroy
  + 2.8.52 pthread\_barrierattr\_setpshared
  + 2.8.53 pthread\_barrierattr\_getpshared
  + 2.8.54 pthread\_barrier\_init
  + 2.8.55 pthread\_barrier\_destroy
  + 2.8.56 pthread\_barrier\_wait

**Initialization**.

* + 2.8.57 pthread\_once

**Signals**.

* + 2.8.58 pthread\_kill
  + 2.8.59 pthread\_sigmask

No support for the following pthread interfaces is provided by NuttX:

* pthread\_atfork. register fork handlers.
* pthread\_attr\_getdetachstate. get and set the detachstate attribute.
* pthread\_attr\_getguardsize. get and set the thread guardsize attribute.
* pthread\_attr\_getinheritsched. get and set the inheritsched attribute.
* pthread\_attr\_getscope. get and set the contentionscope attribute.
* pthread\_attr\_getstack. get and set stack attributes.
* pthread\_attr\_getstackaddr. get and set the stackaddr attribute.
* pthread\_attr\_setdetachstate. get and set the detachstate attribute.
* pthread\_attr\_setguardsize. get and set the thread guardsize attribute.
* pthread\_attr\_setscope. get and set the contentionscope attribute.
* pthread\_attr\_setstack. get and set stack attributes.
* pthread\_attr\_setstackaddr. get and set the stackaddr attribute.
* pthread\_condattr\_getclock. set the clock selection condition variable attribute.
* pthread\_condattr\_getpshared. get the process-shared condition variable attribute.
* pthread\_condattr\_setclock. set the clock selection condition variable attribute.
* pthread\_condattr\_setpshared. set the process-shared condition variable attribute.
* pthread\_getconcurrency. get and set the level of concurrency.
* pthread\_getcpuclockid. access a thread CPU-time clock.
* pthread\_mutex\_getprioceiling. get and set the priority ceiling of a mutex.
* pthread\_mutex\_setprioceiling. get and set the priority ceiling of a mutex.
* pthread\_mutex\_timedlock. lock a mutex.
* pthread\_mutexattr\_getprioceiling. get and set the prioceiling attribute of the mutex attributes object.
* pthread\_mutexattr\_setprioceiling. get and set the prioceiling attribute of the mutex attributes object.
* pthread\_rwlockattr\_destroy. destroy and initialize the read-write lock attributes object.
* pthread\_rwlockattr\_getpshared. get and set the process-shared attribute of the read-write lock attributes object.
* pthread\_rwlockattr\_init. destroy and initialize the read-write lock attributes object.
* pthread\_rwlockattr\_setpshared. get and set the process-shared attribute of the read-write lock attributes object.
* pthread\_setconcurrency. get and set the level of concurrency.
* pthread\_spin\_destroy. destroy or initialize a spin lock object.
* pthread\_spin\_init. destroy or initialize a spin lock object.
* pthread\_spin\_lock. lock a spin lock object.
* pthread\_spin\_trylock. lock a spin lock object.
* pthread\_spin\_unlock. unlock a spin lock object.

**2.8.1 pthread\_attr\_init**

**Function Prototype:**

#include <pthread.h>

int pthread\_attr\_init(pthread\_attr\_t \*attr);

**Description:** Initializes a thread attributes object (attr) with default values for all of the individual attributes used by the implementation.

**Input Parameters:**

* To be provided.

**Returned Value:**

If successful, the pthread\_attr\_init() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* To be provided.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.2 pthread\_attr\_destroy**

**Function Prototype:**

#include <pthread.h>

int pthread\_attr\_destroy(pthread\_attr\_t \*attr);

**Description:** An attributes object can be deleted when it is no longer needed.

**Input Parameters:**

* To be provided.

**Returned Value:**

If successful, the pthread\_attr\_destroy() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* To be provided.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.3 pthread\_attr\_setschedpolicy**

**Function Prototype:**

#include <pthread.h>

int pthread\_attr\_setschedpolicy(pthread\_attr\_t \*attr, int policy);

**Description:**

**Input Parameters:**

* To be provided.

**Returned Value:**

If successful, the pthread\_attr\_setschedpolicy() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* To be provided.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.4 pthread\_attr\_getschedpolicy**

**Function Prototype:**

#include <pthread.h>

int pthread\_attr\_getschedpolicy(pthread\_attr\_t \*attr, int \*policy);

**Description:**

**Input Parameters:**

* To be provided.

**Returned Value:**

If successful, the pthread\_attr\_getschedpolicy() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* To be provided.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.5 pthread\_attr\_getschedpolicy**

**Function Prototype:**

#include <pthread.h>

int pthread\_attr\_setschedparam(pthread\_attr\_t \*attr,

const struct sched\_param \*param);

**Description:**

**Input Parameters:**

* To be provided.

**Returned Value:**

If successful, the pthread\_attr\_getschedpolicy() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* To be provided.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.6 pthread\_attr\_getschedparam**

**Function Prototype:**

#include <pthread.h>

int pthread\_attr\_getschedparam(pthread\_attr\_t \*attr,

struct sched\_param \*param);

**Description:**

**Input Parameters:**

* To be provided.

**Returned Value:**

If successful, the pthread\_attr\_getschedparam() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* To be provided.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.7 pthread\_attr\_setinheritsched**

**Function Prototype:**

#include <pthread.h>

int pthread\_attr\_setinheritsched(pthread\_attr\_t \*attr,

int inheritsched);

**Description:**

**Input Parameters:**

* To be provided.

**Returned Value:**

If successful, the pthread\_attr\_setinheritsched() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* To be provided.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.8 pthread\_attr\_getinheritsched**

**Function Prototype:**

#include <pthread.h>

int pthread\_attr\_getinheritsched(const pthread\_attr\_t \*attr,

int \*inheritsched);

**Description:**

**Input Parameters:**

* To be provided.

**Returned Value:**

If successful, the pthread\_attr\_getinheritsched() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* To be provided.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.9 pthread\_attr\_setstacksize**

**Function Prototype:**

#include <pthread.h>

int pthread\_attr\_setstacksize(pthread\_attr\_t \*attr, long stacksize);

**Description:**

**Input Parameters:**

* To be provided.

**Returned Value:**

If successful, the pthread\_attr\_setstacksize() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* To be provided.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.10 pthread\_attr\_getstacksize**

**Function Prototype:**

#include <pthread.h>

int pthread\_attr\_getstacksize(pthread\_attr\_t \*attr, long \*stackaddr);

**Description:**

**Input Parameters:**

* To be provided.

**Returned Value:**

If successful, the pthread\_attr\_getstacksize() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* To be provided.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.11 pthread\_create**

**Function Prototype:**

#include <pthread.h>

int pthread\_create(pthread\_t \*thread, pthread\_attr\_t \*attr,

pthread\_startroutine\_t startRoutine,

pthread\_addr\_t arg);

**Description:** To create a thread object and runnable thread, a routine must be specified as the new thread's start routine. An argument may be passed to this routine, as an untyped address; an untyped address may also be returned as the routine's value. An attributes object may be used to specify details about the kind of thread being created.

**Input Parameters:**

* To be provided.

**Returned Value:**

If successful, the pthread\_create() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* To be provided.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.12 pthread\_detach**

**Function Prototype:**

#include <pthread.h>

int pthread\_detach(pthread\_t thread);

**Description:** A thread object may be "detached" to specify that the return value and completion status will not be requested.

**Input Parameters:**

* To be provided.

**Returned Value:**

If successful, the pthread\_detach() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* To be provided.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.13 pthread\_exit**

**Function Prototype:**

#include <pthread.h>

void pthread\_exit(pthread\_addr\_t pvValue);

**Description:** A thread may terminate it's own execution.

**Input Parameters:**

* To be provided.

**Returned Value:**

If successful, the pthread\_exit() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* To be provided.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.14 pthread\_cancel**

**Function Prototype:**

#include <pthread.h>

int pthread\_cancel(pthread\_t thread);

**Description:**

The pthread\_cancel() function will request that thread be canceled. The target thread's cancelability state, enabled, or disabled, determines when the cancellation takes effect: When the cancellation is acted on, thread will be terminated. When cancelability is disabled, all cancellations are held pending in the target thread until the thread re-enables cancelability.

The target thread's cancelability state determines how the cancellation is acted on: Either asychronrously or deferred. Asynchronous cancellations we be acted upon immediately (when enabled), interrupting the thread with its processing in an abritray state.

When cancelability is deferred, all cancellations are held pending in the target thread until the thread changes the cancelability type or a [Cancellation Point](http://www.nuttx.org/doku.php?id=wiki:nxinternal:cancellation-points) function such as [pthread\_testcancel()](http://nuttx.org/Documentation/NuttxUserGuide.html#pthreadtestcancel) is entered.

**Input Parameters:**

* thread. Identifies the thread to be canceled.

**Returned Value:**

If successful, the pthread\_cancel() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* ESRCH. No thread could be found corresponding to that specified by the given thread ID.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name. Except:

* The thread-specific data destructor functions will be not called for thread These destructors are not currently supported.

**2.8.15 pthread\_setcancelstate**

**Function Prototype:**

#include <pthread.h>

int pthread\_setcancelstate(int state, int \*oldstate);

**Description:**

The pthread\_setcancelstate() function atomically sets both the calling thread's cancelability state to the indicated state and returns the previous cancelability state at the location referenced by oldstate. Legal values for state are PTHREAD\_CANCEL\_ENABLE and PTHREAD\_CANCEL\_DISABLE.

Any pending thread cancellation may occur at the time that the cancellation state is set to PTHREAD\_CANCEL\_ENABLE.

**Input Parameters:**

* state New cancellation state. One of PTHREAD\_CANCEL\_ENABLE or PTHREAD\_CANCEL\_DISABLE.
* oldstate. Location to return the previous cancellation state.

**Returned Value:**

If successful, the pthread\_setcancelstate() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* ESRCH. No thread could be found corresponding to that specified by the given thread ID.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.16 pthread\_setcanceltype**

**Function Prototype:**

#include <pthread.h>

int pthread\_setcanceltype(int type, FAR int \*oldtype);

**Description:** The pthread\_setcanceltype() function atomically both sets the calling thread's cancelability type to the indicated type and returns the previous cancelability type at the location referenced by oldtype. Legal values for type are PTHREAD\_CANCEL\_DEFERREDand PTHREAD\_CANCEL\_ASYNCHRONOUS.

The cancelability state and type of any newly created threads are PTHREAD\_CANCEL\_ENABLE and PTHREAD\_CANCEL\_DEFERRED respectively.

**Input Parameters:**

* type New cancellation state. One of PTHREAD\_CANCEL\_DEFERRED or PTHREAD\_CANCEL\_ASYNCHRONOUS.
* oldtype. Location to return the previous cancellation type.

**Returned Value:**

If successful, the pthread\_setcancelstate() function will return zero (OK). Otherwise, an error number will be returned to indicate the error.

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.17 pthread\_testcancel**

**Function Prototype:**

#include <pthread.h>

void pthread\_testcancel(void);

**Description:**

The pthread\_testcancel() function creates a [Cancellation Point](http://www.nuttx.org/doku.php?id=wiki:nxinternal:cancellation-points) in the calling thread. The pthread\_testcancel() function has no effect if cancelability is disabled.

**Input Parameters:** None

**Returned Value:** None

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.18 pthread\_cleanup\_pop**

**Function Prototype:**

#include <pthread.h>

void pthread\_cleanup\_pop(int execute);

**Description:**

The pthread\_cleanup\_pop() function will remove the routine at the top of the calling thread's cancellation cleanup stack and optionally invoke it (if execute is non-zero).

**Input Parameters:**

* execute. Execute the popped cleanup function immediately.

**Returned Value:**

If successful, the pthread\_setcancelstate() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.19 pthread\_cleanup\_push**

**Function Prototype:**

#include <pthread.h>

void pthread\_cleanup\_push(CODE void (\*routine)(FAR void \*), FAR void \*arg);

**Description:**

The pthread\_cleanup\_push() function will push the specified cancellation cleanup handler routine onto the calling thread's cancellation cleanup stack.

The cancellation cleanup handler will be popped from the cancellation cleanup stack and invoked with the argument arg when:

* The thread exits (that is, calls pthread\_exit()).
* The thread acts upon a cancellation request.
* The thread calls pthread\_cleanup\_pop() with a non-zero execute argument.

**Input Parameters:**

* routine. The cleanup routine to be pushed on the cleanup stack.
* arg. An argument that will accompany the callback.

**Returned Value:**

If successful, the pthread\_setcancelstate() function will return zero (OK). Otherwise, an error number will be returned to indicate the error.

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.20 pthread\_join**

**Function Prototype:**

#include <pthread.h>

int pthread\_join(pthread\_t thread, pthread\_addr\_t \*ppvValue);

**Description:** A thread can await termination of another thread and retrieve the return value of the thread.

**Input Parameters:**

* To be provided.

**Returned Value:**

If successful, the pthread\_join() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* To be provided.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.21 pthread\_yield**

**Function Prototype:**

#include <pthread.h>

void pthread\_yield(void);

**Description:** A thread may tell the scheduler that its processor can be made available.

**Input Parameters:**

* To be provided.

**Returned Value:**

If successful, the pthread\_yield() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* To be provided.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.22 pthread\_self**

**Function Prototype:**

#include <pthread.h>

pthread\_t pthread\_self(void);

**Description:** A thread may obtain a copy of its own thread handle.

**Input Parameters:**

* To be provided.

**Returned Value:**

If successful, the pthread\_self() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* To be provided.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.23 pthread\_getschedparam**

**Function Prototype:**

#include <pthread.h>

int pthread\_getschedparam(pthread\_t thread, int \*policy,

struct sched\_param \*param);

**Description:** The pthread\_getschedparam() functions will get the scheduling policy and parameters of threads. For SCHED\_FIFO and SCHED\_RR, the only required member of the sched\_param structure is the priority sched\_priority.

The pthread\_getschedparam() function will retrieve the scheduling policy and scheduling parameters for the thread whose thread ID is given by thread and will store those values in policy and param, respectively. The priority value returned from pthread\_getschedparam() will be the value specified by the most recent pthread\_setschedparam(), pthread\_setschedprio(), or pthread\_create() call affecting the target thread. It will not reflect any temporary adjustments to its priority (such as might result of any priority inheritance, for example).

The policy parameter may have the value SCHED\_FIFO, SCHED\_RR, or SCHED\_SPORADIC. SCHED\_RR requires the configuration setting CONFIG\_RR\_INTERVAL > 0; SCHED\_SPORADIC requires the configuration setting CONFIG\_SCHED\_SPORADIC=y. (SCHED\_OTHER and non-standard scheduler policies, in particular, are not supported). The SCHED\_FIFO and SCHED\_RR policies will have a single scheduling parameter:

* sched\_priority The thread priority.

The SCHED\_SPORADIC policy has four additional scheduling parameters:

* sched\_ss\_low\_priority Low scheduling priority for sporadic server.
* sched\_ss\_repl\_period Replenishment period for sporadic server.
* sched\_ss\_init\_budget Initial budget for sporadic server.
* sched\_ss\_max\_repl Maximum pending replenishments for sporadic server.

**Input Parameters:**

* thread. The ID of thread whose scheduling parameters will be queried.
* policy. The location to store the thread's scheduling policy.
* param. The location to store the thread's priority.

**Returned Value:** 0 (OK) if successful. Otherwise, the error code ESRCH if the value specified by thread does not refer to an existing thread.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.24 pthread\_setschedparam**

**Function Prototype:**

#include <pthread.h>

int pthread\_setschedparam(pthread\_t thread, int policy,

const struct sched\_param \*param);

**Description:** The pthread\_setschedparam() functions will set the scheduling policy and parameters of threads. For SCHED\_FIFO and SCHED\_RR, the only required member of the sched\_param structure is the priority sched\_priority.

The pthread\_setschedparam() function will set the scheduling policy and associated scheduling parameters for the thread whose thread ID is given by thread to the policy and associated parameters provided in policy and param, respectively.

The policy parameter may have the value SCHED\_FIFO or SCHED\_RR. (SCHED\_OTHER and SCHED\_SPORADIC, in particular, are not supported). The SCHED\_FIFO and SCHED\_RR policies will have a single scheduling parameter, sched\_priority.

If the pthread\_setschedparam() function fails, the scheduling parameters will not be changed for the target thread.

**Input Parameters:**

* thread. The ID of thread whose scheduling parameters will be modified.
* policy. The new scheduling policy of the thread. Either SCHED\_FIFO or SCHED\_RR. SCHED\_OTHER and SCHED\_SPORADIC are not supported.
* param. The location to store the thread's priority.

**Returned Value:**

If successful, the pthread\_setschedparam() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* EINVAL. The value specified by policy or one of the scheduling parameters associated with the scheduling policy policy is invalid.
* ENOTSUP. An attempt was made to set the policy or scheduling parameters to an unsupported value (SCHED\_OTHER and SCHED\_SPORADIC in particular are not supported)
* EPERM. The caller does not have the appropriate permission to set either the scheduling parameters or the scheduling policy of the specified thread. Or, the implementation does not allow the application to modify one of the parameters to the value specified.
* ESRCH. The value specified by thread does not refer to a existing thread.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.25 pthread\_key\_create**

**Function Prototype:**

#include <pthread.h>

int pthread\_key\_create(pthread\_key\_t \*key, void (\*destructor)(void\*))

**Description:**

This function creates a thread-specific data key visible to all threads in the system. Although the same key value may be used by different threads, the values bound to the key by pthread\_setspecific() are maintained on a per-thread basis and persist for the life of the calling thread.

Upon key creation, the value NULL will be associated with the new key in all active threads. Upon thread creation, the value NULL will be associated with all defined keys in the new thread.

**Input Parameters:**

* key is a pointer to the key to create.
* destructor is an optional destructor() function that may be associated with each key that is invoked when a thread exits. However, this argument is ignored in the current implementation.

**Returned Value:**

If successful, the pthread\_key\_create() function will store the newly created key value at \*key and return zero (OK). Otherwise, an error number will be returned to indicate the error:

* EAGAIN. The system lacked sufficient resources to create another thread-specific data key, or the system-imposed limit on the total number of keys per task {PTHREAD\_KEYS\_MAX} has been exceeded
* ENONMEM Insufficient memory exists to create the key.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

* The present implementation ignores the destructor argument.

**2.8.26 pthread\_setspecific**

**Function Prototype:**

#include <pthread.h>

int pthread\_setspecific(pthread\_key\_t key, void \*value)

**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 effect of calling pthread\_setspecific() with a key value not obtained from pthread\_key\_create() or after a key has been deleted with pthread\_key\_delete() is undefined.

**Input Parameters:**

* key. The data key to set the binding for.
* value. The value to bind to the key.

**Returned Value:**

If successful, pthread\_setspecific() will return zero (OK). Otherwise, an error number will be returned:

* ENOMEM. Insufficient memory exists to associate the value with the key.
* EINVAL. The key value is invalid.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

* pthread\_setspecific() may be called from a thread-specific data destructor function.

**2.8.27 pthread\_getspecific**

**Function Prototype:**

#include <pthread.h>

void \*pthread\_getspecific(pthread\_key\_t key)

**Description:**

The pthread\_getspecific() function returns the value currently bound to the specified key on behalf of the calling thread.

The effect of calling pthread\_getspecific() with a key value not obtained from pthread\_key\_create() or after a key has been deleted with pthread\_key\_delete() is undefined.

**Input Parameters:**

* key. The data key to get the binding for.

**Returned Value:**

The function pthread\_getspecific() returns the thread- specific data associated with the given key. If no thread specific data is associated with the key, then the value NULL is returned.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

* pthread\_getspecific() may be called from a thread-specific data destructor function.

**2.8.28 pthread\_key\_delete**

**Function Prototype:**

#include <pthread.h>

int pthread\_key\_delete(pthread\_key\_t key)

**Description:**

This POSIX function should delete a thread-specific data key previously returned by pthread\_key\_create(). However, this function does nothing in the present implementation.

**Input Parameters:**

* key. The key to delete

**Returned Value:**

* Always returns EINVAL.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.29 pthread\_mutexattr\_init**

**Function Prototype:**

#include <pthread.h>

int pthread\_mutexattr\_init(pthread\_mutexattr\_t \*attr);

**Description:**

**Input Parameters:**

* To be provided.

**Returned Value:**

If successful, the pthread\_mutexattr\_init() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* To be provided.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.30 pthread\_mutexattr\_destroy**

**Function Prototype:**

#include <pthread.h>

int pthread\_mutexattr\_destroy(pthread\_mutexattr\_t \*attr);

**Description:**

**Input Parameters:**

* To be provided.

**Returned Value:**

If successful, the pthread\_mutexattr\_destroy() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* To be provided.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.31 pthread\_mutexattr\_getpshared**

**Function Prototype:**

#include <pthread.h>

int pthread\_mutexattr\_getpshared(pthread\_mutexattr\_t \*attr,

int \*pshared);

**Description:**

**Input Parameters:**

* To be provided.

**Returned Value:**

If successful, the pthread\_mutexattr\_getpshared() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* To be provided.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.32 pthread\_mutexattr\_setpshared**

**Function Prototype:**

#include <pthread.h>

int pthread\_mutexattr\_setpshared(pthread\_mutexattr\_t \*attr,

int pshared);

**Description:**

**Input Parameters:**

* To be provided.

**Returned Value:**

If successful, the pthread\_mutexattr\_setpshared() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* To be provided.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.33 pthread\_mutexattr\_gettype**

**Function Prototype:**

#include <pthread.h>

#ifdef CONFIG\_PTHREAD\_MUTEX\_TYPES

int pthread\_mutexattr\_gettype(const pthread\_mutexattr\_t \*attr, int \*type);

#endif

**Description:** Return the mutex type from the mutex attributes.

**Input Parameters:**

* attr. The mutex attributes to query
* type. Location to return the mutex type. See [pthread\_mutexattr\_setttyp()](http://nuttx.org/Documentation/NuttxUserGuide.html#pthreadmutexattrsettype) for a description of possible mutex types that may be returned.

**Returned Value:**

If successful, the pthread\_mutexattr\_settype() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* EINVAL. Parameters attr and/or attr are invalid.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.34 pthread\_mutexattr\_settype**

**Function Prototype:**

#include <pthread.h>

#ifdef CONFIG\_PTHREAD\_MUTEX\_TYPES

int pthread\_mutexattr\_settype(pthread\_mutexattr\_t \*attr, int type);

#endif

**Description:** Set the mutex type in the mutex attributes.

**Input Parameters:**

* attr. The mutex attributes in which to set the mutex type.
* type. The mutex type value to set. The following values are supported:
  + PTHREAD\_MUTEX\_NORMAL. This type of mutex does not detect deadlock. A thread attempting to re-lock this mutex without first unlocking it will deadlock. Attempting to unlock a mutex locked by a different thread results in undefined behavior. Attempting to unlock an unlocked mutex results in undefined behavior.
  + PTHREAD\_MUTEX\_ERRORCHECK. This type of mutex provides error checking. A thread attempting to re-lock this mutex without first unlocking it will return with an error. A thread attempting to unlock a mutex which another thread has locked will return with an error. A thread attempting to unlock an unlocked mutex will return with an error.
  + PTHREAD\_MUTEX\_RECURSIVE. A thread attempting to re-lock this mutex without first unlocking it will succeed in locking the mutex. The re-locking deadlock which can occur with mutexes of type PTHREAD\_MUTEX\_NORMAL cannot occur with this type of mutex. Multiple locks of this mutex require the same number of unlocks to release the mutex before another thread can acquire the mutex. A thread attempting to unlock a mutex which another thread has locked will return with an error. A thread attempting to unlock an unlocked mutex will return with an error.
  + PTHREAD\_MUTEX\_DEFAULT. The default mutex type (PTHREAD\_MUTEX\_NORMAL).

In NuttX, PTHREAD\_MUTEX\_NORMAL is not implemented. Rather, the behavior described for PTHREAD\_MUTEX\_ERRORCHECK is the *normal* behavior.

**Returned Value:**

If successful, the pthread\_mutexattr\_settype() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* EINVAL. Parameters attr and/or attr are invalid.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.35 pthread\_mutexattr\_getprotocol**

**Function Prototype:**

#include <pthread.h>

int pthread\_mutexattr\_getprotocol(FAR const pthread\_mutexattr\_t \*attr,

FAR int \*protocol);

**Description:** Return the value of the mutex protocol attribute..

**Input Parameters:**

* attr. A pointer to the mutex attributes to be queried
* protocol. The user provided location in which to store the protocol value. May be one of PTHREAD\_PRIO\_NONE, or PTHREAD\_PRIO\_INHERIT, PTHREAD\_PRIO\_PROTECT.

**Returned Value:**

If successful, the pthread\_mutexattr\_getprotocol() function will return zero (OK). Otherwise, an error number will be returned to indicate the error.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.36 pthread\_mutexattr\_setprotocol**

**Function Prototype:**

#include <pthread.h>

int pthread\_mutexattr\_setprotocol(FAR pthread\_mutexattr\_t \*attr,

int protocol);

**Description:** Set mutex protocol attribute. See the paragraph Locking versus Signaling Semaphores for some important information about the use of this interface.

**Input Parameters:**

* attr. A pointer to the mutex attributes to be modified
* protocol. The new protocol to use. One of PTHREAD\_PRIO\_NONE, or PTHREAD\_PRIO\_INHERIT, PTHREAD\_PRIO\_PROTECT. PTHREAD\_PRIO\_INHERIT is supported only if  CONFIG\_PRIORITY\_INHERITANCE is defined;  PTHREAD\_PRIO\_PROTECT is not currently supported in any configuration.

**Returned Value:**

If successful, the pthread\_mutexattr\_setprotocol() function will return zero (OK). Otherwise, an error number will be returned to indicate the error.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.37 pthread\_mutex\_init**

**Function Prototype:**

#include <pthread.h>

int pthread\_mutex\_init(pthread\_mutex\_t \*mutex,

pthread\_mutexattr\_t \*attr);

**Description:**

**Input Parameters:**

* To be provided.

**Returned Value:**

If successful, the pthread\_mutex\_init() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* To be provided.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.38 pthread\_mutex\_destroy**

**Function Prototype:**

#include <pthread.h>

int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);

**Description:**

**Input Parameters:**

* To be provided.

**Returned Value:**

If successful, the pthread\_mutex\_destroy() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* To be provided.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.39 pthread\_mutex\_lock**

**Function Prototype:**

#include <pthread.h>

int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);

**Description:** The mutex object referenced by mutex is locked by calling pthread\_mutex\_lock(). If the mutex is already locked, the calling thread blocks until the mutex becomes available. This operation returns with the mutex object referenced by mutex in the locked state with the calling thread as its owner.

If the mutex type is PTHREAD\_MUTEX\_NORMAL, deadlock detection is not provided. Attempting to re-lock the mutex causes deadlock. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, undefined behavior results.

In NuttX, PTHREAD\_MUTEX\_NORMAL is not implemented. Rather, the behavior described for PTHREAD\_MUTEX\_ERRORCHECK is the *normal*behavior.

If the mutex type is PTHREAD\_MUTEX\_ERRORCHECK, then error checking is provided. If a thread attempts to re-lock a mutex that it has already locked, an error will be returned. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error will be returned.

If the mutex type is PTHREAD\_MUTEX\_RECURSIVE, then the mutex maintains the concept of a lock count. When a thread successfully acquires a mutex for the first time, the lock count is set to one. Every time a thread re-locks this mutex, the lock count is incremented by one. Each time the thread unlocks the mutex, the lock count is decremented by one. When the lock count reaches zero, the mutex becomes available for other threads to acquire. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error will be returned.

If a signal is delivered to a thread waiting for a mutex, upon return from the signal handler the thread resumes waiting for the mutex as if it was not interrupted.

**Input Parameters:**

* mutex. A reference to the mutex to be locked.

**Returned Value:**

If successful, the pthread\_mutex\_lock() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* To be provided.

Note that this function will never return the error EINTR.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.40 pthread\_mutex\_trylock**

**Function Prototype:**

#include <pthread.h>

int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex);

**Description:** The function pthread\_mutex\_trylock() is identical to pthread\_mutex\_lock() except that if the mutex object referenced by mutex is currently locked (by any thread, including the current thread), the call returns immediately with the errno EBUSY.

If a signal is delivered to a thread waiting for a mutex, upon return from the signal handler the thread resumes waiting for the mutex as if it was not interrupted.

**Input Parameters:**

* mutex. A reference to the mutex to be locked.

**Returned Value:**

If successful, the pthread\_mutex\_trylock() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* To be provided.

Note that this function will never return the error EINTR.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.41 pthread\_mutex\_unlock**

**Function Prototype:**

#include <pthread.h>

int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);

**Description:**

The pthread\_mutex\_unlock() function releases the mutex object referenced by mutex. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by mutex when pthread\_mutex\_unlock() is called, resulting in the mutex becoming available, the scheduling policy is used to determine which thread will acquire the mutex. (In the case of PTHREAD\_MUTEX\_RECURSIVE mutexes, the mutex becomes available when the count reaches zero and the calling thread no longer has any locks on this mutex).

If a signal is delivered to a thread waiting for a mutex, upon return from the signal handler the thread resumes waiting for the mutex as if it was not interrupted.

**Input Parameters:**

* mutex.

**Returned Value:**

If successful, the pthread\_mutex\_unlock() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* To be provided.

Note that this function will never return the error EINTR.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.42 pthread\_condattr\_init**

**Function Prototype:**

#include <pthread.h>

int pthread\_condattr\_init(pthread\_condattr\_t \*attr);

**Description:**

**Input Parameters:**

* To be provided.

**Returned Value:**

If successful, the pthread\_condattr\_init() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* To be provided.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.43 pthread\_condattr\_destroy**

**Function Prototype:**

#include <pthread.h>

int pthread\_condattr\_destroy(pthread\_condattr\_t \*attr);

**Description:**

**Input Parameters:**

* To be provided.

**Returned Value:**

If successful, the pthread\_condattr\_destroy() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* To be provided.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.44 pthread\_cond\_init**

**Function Prototype:**

#include <pthread.h>

int pthread\_cond\_init(pthread\_cond\_t \*cond, pthread\_condattr\_t \*attr);

**Description:**

**Input Parameters:**

* To be provided.

**Returned Value:**

If successful, the pthread\_cond\_init() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* To be provided.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.45 pthread\_cond\_destroy**

**Function Prototype:**

#include <pthread.h>

int pthread\_cond\_destroy(pthread\_cond\_t \*cond);

**Description:**

**Input Parameters:**

* To be provided.

**Returned Value:**

If successful, the pthread\_cond\_destroy() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* To be provided.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.46 pthread\_cond\_broadcast**

**Function Prototype:**

#include <pthread.h>

int pthread\_cond\_broadcast(pthread\_cond\_t \*cond);

**Description:**

**Input Parameters:**

* To be provided.

**Returned Value:**

If successful, the pthread\_cond\_broadcast() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* To be provided.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.47 pthread\_cond\_signal**

**Function Prototype:**

#include <pthread.h>

int pthread\_cond\_signal(pthread\_cond\_t \*dond);

**Description:**

**Input Parameters:**

* To be provided.

**Returned Value:**

If successful, the pthread\_cond\_signal() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* To be provided.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.48 pthread\_cond\_wait**

**Function Prototype:**

#include <pthread.h>

int pthread\_cond\_wait(pthread\_cond\_t \*cond, pthread\_mutex\_t \*mutex);

**Description:**

**Input Parameters:**

* To be provided.

**Returned Value:**

If successful, the pthread\_cond\_wait() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* To be provided.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.49 pthread\_cond\_timedwait**

**Function Prototype:**

#include <pthread.h>

int pthread\_cond\_timedwait(pthread\_cond\_t \*cond, pthread\_mutex\_t \*mutex,

const struct timespec \*abstime);

**Description:**

**Input Parameters:**

* To be provided.

**Returned Value:**

If successful, the pthread\_cond\_timedwait() function will return zero (OK). Otherwise, an error number will be returned to indicate the error:

* To be provided.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.50 pthread\_barrierattr\_init**

**Function Prototype:**

#include <pthread.h>

int pthread\_barrierattr\_init(FAR pthread\_barrierattr\_t \*attr);

**Description:** The pthread\_barrierattr\_init() function will initialize a barrier attribute object attr with the default value for all of the attributes defined by the implementation.

**Input Parameters:**

* attr. Barrier attributes to be initialized.

**Returned Value:** 0 (OK) on success or EINVAL if attr is invalid.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.51 pthread\_barrierattr\_destroy**

**Function Prototype:**

#include <pthread.h>

int pthread\_barrierattr\_destroy(FAR pthread\_barrierattr\_t \*attr);

**Description:** The pthread\_barrierattr\_destroy() function will destroy a barrier attributes object. A destroyed attributes object can be reinitialized using pthread\_barrierattr\_init(); the results of otherwise referencing the object after it has been destroyed are undefined.

**Input Parameters:**

* attr. Barrier attributes to be destroyed.

**Returned Value:** 0 (OK) on success or EINVAL if attr is invalid.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.52 pthread\_barrierattr\_setpshared**

**Function Prototype:**

#include <pthread.h>

int pthread\_barrierattr\_setpshared(FAR pthread\_barrierattr\_t \*attr, int pshared);

**Description:** The process-shared attribute is set to PTHREAD\_PROCESS\_SHARED to permit a barrier to be operated upon by any thread that has access to the memory where the barrier is allocated. If the process-shared attribute is PTHREAD\_PROCESS\_PRIVATE, the barrier can only be operated upon by threads created within the same process as the thread that initialized the barrier. If threads of different processes attempt to operate on such a barrier, the behavior is undefined. The default value of the attribute is PTHREAD\_PROCESS\_PRIVATE.

**Input Parameters:**

* attr. Barrier attributes to be modified.
* pshared. The new value of the pshared attribute.

**Returned Value:** 0 (OK) on success or EINVAL if either attr is invalid or pshared is not one of PTHREAD\_PROCESS\_SHARED or PTHREAD\_PROCESS\_PRIVATE.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.53 pthread\_barrierattr\_getpshared**

**Function Prototype:**

#include <pthread.h>

int pthread\_barrierattr\_getpshared(FAR const pthread\_barrierattr\_t \*attr, FAR int \*pshared);

**Description:** The pthread\_barrierattr\_getpshared() function will obtain the value of the process-shared attribute from the attributes object referenced by attr.

**Input Parameters:**

* attr. Barrier attributes to be queried.
* pshared. The location to stored the current value of the pshared attribute.

**Returned Value:** 0 (OK) on success or EINVAL if either attr or pshared is invalid.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.54 pthread\_barrier\_init**

**Function Prototype:**

#include <pthread.h>

int pthread\_barrier\_init(FAR pthread\_barrier\_t \*barrier,

FAR const pthread\_barrierattr\_t \*attr, unsigned int count);

**Description:** The pthread\_barrier\_init() function allocates any resources required to use the barrier referenced by barrier and initialized the barrier with the attributes referenced by attr. If attr is NULL, the default barrier attributes will be used. The results are undefined if pthread\_barrier\_init() is called when any thread is blocked on the barrier. The results are undefined if a barrier is used without first being initialized. The results are undefined if pthread\_barrier\_init() is called specifying an already initialized barrier.

**Input Parameters:**

* barrier. The barrier to be initialized.
* attr. Barrier attributes to be used in the initialization.
* count. The count to be associated with the barrier. The count argument specifies the number of threads that must callpthread\_barrier\_wait() before any of them successfully return from the call. The value specified by count must be greater than zero.

**Returned Value:**0 (OK) on success or on of the following error numbers:

* EAGAIN. The system lacks the necessary resources to initialize another barrier.
* EINVAL. The barrier reference is invalid, or the values specified by attr are invalid, or the value specified by count is equal to zero.
* ENOMEM. Insufficient memory exists to initialize the barrier.
* EBUSY. The implementation has detected an attempt to reinitialize a barrier while it is in use.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.55 pthread\_barrier\_destroy**

**Function Prototype:**

#include <pthread.h>

int pthread\_barrier\_destroy(FAR pthread\_barrier\_t \*barrier);

**Description:** The pthread\_barrier\_destroy() function destroys the barrier referenced by barrie and releases any resources used by the barrier. The effect of subsequent use of the barrier is undefined until the barrier is reinitialized by another call to pthread\_barrier\_init(). The results are undefined if pthread\_barrier\_destroy() is called when any thread is blocked on the barrier, or if this function is called with an uninitialized barrier.

**Input Parameters:**

* barrier. The barrier to be destroyed.

**Returned Value:** 0 (OK) on success or on of the following error numbers:

* EBUSY. The implementation has detected an attempt to destroy a barrier while it is in use.
* EINVAL. The value specified by barrier is invalid.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.56 pthread\_barrier\_wait**

**Function Prototype:**

#include <pthread.h>

int pthread\_barrier\_wait(FAR pthread\_barrier\_t \*barrier);

**Description:** The pthread\_barrier\_wait() function synchronizes participating threads at the barrier referenced by barrier. The calling thread is blocked until the required number of threads have called pthread\_barrier\_wait() specifying the same barrier. When the required number of threads have called pthread\_barrier\_wait() specifying the barrier, the constant PTHREAD\_BARRIER\_SERIAL\_THREAD will be returned to one unspecified thread and zero will be returned to each of the remaining threads. At this point, the barrier will be reset to the state it had as a result of the most recent pthread\_barrier\_init() function that referenced it.

The constant PTHREAD\_BARRIER\_SERIAL\_THREAD is defined in pthread.h and its value must be distinct from any other value returned by pthread\_barrier\_wait().

The results are undefined if this function is called with an uninitialized barrier.

If a signal is delivered to a thread blocked on a barrier, upon return from the signal handler the thread will resume waiting at the barrier if the barrier wait has not completed. Otherwise, the thread will continue as normal from the completed barrier wait. Until the thread in the signal handler returns from it, it is unspecified whether other threads may proceed past the barrier once they have all reached it.

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

**Input Parameters:**

* barrier. The barrier on which to wait.

**Returned Value:** 0 (OK) on success or EINVAL if the barrier is not valid.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.57 pthread\_once**

**Function Prototype:**

#include <pthread.h>

int pthread\_once(FAR pthread\_once\_t \*once\_control, CODE void (\*init\_routine)(void));

**Description:** The first call to pthread\_once() by any thread with a given once\_control, will call the init\_routine() with no arguments. Subsequent calls to pthread\_once() with the same once\_control will have no effect. On return from pthread\_once(), init\_routine() will have completed.

**Input Parameters:**

* once\_control. Determines if init\_routine() should be called. once\_control should be declared and initialized as follows:
* pthread\_once\_t once\_control = PTHREAD\_ONCE\_INIT;

PTHREAD\_ONCE\_INIT is defined in pthread.h.

* init\_routine. The initialization routine that will be called once.

**Returned Value:** 0 (OK) on success or EINVAL if either once\_control or init\_routine are invalid.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.58 pthread\_kill**

**Function Prototype:**

#include <signal.h>

#include <pthread.h>

int pthread\_kill(pthread\_t thread, int signo)

**Description:** The pthread\_kill() system call can be used to send any signal to a thread. See kill() for further information as this is just a simple wrapper around the kill() function.

**Input Parameters:**

* thread. The id of the thread to receive the signal. Only positive, non-zero values of tthreadt are supported.
* signo. The signal number to send. If signo is zero, no signal is sent, but all error checking is performed.

**Returned Value:**

On success, the signal was sent and zero is returned. On error one of the following error numbers is returned.

* EINVAL. An invalid signal was specified.
* EPERM. The thread does not have permission to send the signal to the target thread.
* ESRCH. No thread could be found corresponding to that specified by the given thread ID.
* ENOSYS. Do not support sending signals to process groups.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

**2.8.59 pthread\_sigmask**

**Function Prototype:**

#include <signal.h>

#include <pthread.h>

int pthread\_sigmask(int how, FAR const sigset\_t \*set, FAR sigset\_t \*oset);

**Description:** This function is a simple wrapper around sigprocmask(). See the sigprocmask() function description for further information.

**Input Parameters:**

* how. How the signal mast will be changed:
  + SIG\_BLOCK: The resulting set is the union of the current set and the signal set pointed to by set.
  + SIG\_UNBLOCK: The resulting set is the intersection of the current set and the complement of the signal set pointed to by set.
  + SIG\_SETMASK: The resulting set is the signal set pointed to by set.
* set. Location of the new signal mask.
* oset. Location to store the old signal mask.

**Returned Value:**

0 (OK) on success or EINVAL if how is invalid.

**Assumptions/Limitations:**

**POSIX Compatibility:** Comparable to the POSIX interface of the same name.

|  |
| --- |
| **2.9 Environment Variables** |

**Overview**. NuttX supports environment variables that can be used to control the behavior of programs. In the spirit of NuttX the environment variable behavior attempts to emulate the behavior of environment variables in the multi-processing OS:

* **Task environments**. When a new task is created using task\_create, the environment of the child task is an inherited, exact copy of the environment of the parent. However, after child task has been created, subsequent operations by the child task on its environment does not alter the environment of the parent. No do operations by the parent effect the child's environment. The environments start identical but are independent and may diverge.
* **Thread environments**. When a pthread is created using pthread\_create, the child thread also inherits that environment of the parent. However, the child does not receive a copy of the environment but, rather, shares the same environment. Changes to the environment are visible to all threads with the same parentage.

**Programming Interfaces**. The following environment variable programming interfaces are provided by Nuttx and are described in detail in the following paragraphs.

* 2.9.1 getenv
* 2.9.2 putenv
* 2.9.3 clearenv
* 2.9.4 setenv
* 2.9.5 unsetenv

**Disabling Environment Variable Support**. All support for environment variables can be disabled by setting CONFIG\_DISABLE\_ENVIRONin the board configuration file.

**2.9.1 getenv**

**Function Prototype:**

#include <stdlib.h>

FAR char \*getenv(const char \*name);

**Description:** The getenv() function searches the environment list for a string that matches the string pointed to by name.

**Input Parameters:**

* name. The name of the variable to find.

**Returned Value:** The value of the variable (read-only) or NULL on failure.

**2.9.2 putenv**

**Function Prototype:**

#include <stdlib.h>

int putenv(char \*string);

**Description:** The putenv() function adds or changes the value of environment variables. The argument string is of the form *name=value*. If name does not already exist in the environment, then string is added to the environment. If name does exist, then the value of name in the environment is changed to value.

**Input Parameters:**

* string *name=value* string describing the environment setting to add/modify.

**Returned Value:** Zero on success.

**2.9.3 clearenv**

**Function Prototype:**

#include <stdlib.h>

int clearenv(void);

**Description:** The clearenv() function clears the environment of all name-value pairs and sets the value of the external variable environ to NULL.

**Input Parameters:** None

**Returned Value:** Zero on success.

**2.9.4 setenv**

**Function Prototype:**

#include <stdlib.h>

int setenv(const char \*name, const char \*value, int overwrite);

**Description:** The setenv() function adds the variable name to the environment with the specified value if the variable name does not exist. If the name does exist in the environment, then its value is changed to value if overwrite is non-zero; if overwrite is zero, then the value of name is unaltered.

**Input Parameters:**

* name The name of the variable to change.
* value The new value of the variable.
* value Replace any existing value if non-zero.

**Returned Value:** Zero on success.

**2.9.5 unsetenv**

**Function Prototype:**

#include <stdlib.h>

int unsetenv(const char \*name);

**Description:** The unsetenv() function deletes the variable name from the environment.

**Input Parameters:**

* name The name of the variable to delete.

**Returned Value:** Zero on success.

|  |
| --- |
| **2.10 File System Interfaces** |

* 2.10.1 NuttX File System Overview
* 2.10.2 Driver Operations
* 2.10.3 Directory Operations
* 2.10.4 UNIX Standard Operations
* 2.10.5 Standard I/O
* 2.10.6 Standard Library
* 2.10.7 Asynchronous I/O
* 2.10.8 Standard String Operations
* 2.10.9 Pipes and FIFOs
* 2.10.10 mmap() and eXecute In Place (XIP)

**2.10.1 NuttX File System Overview**

**Overview**. NuttX includes an optional, scalable file system. This file-system may be omitted altogether; NuttX does not depend on the presence of any file system.

**Pseudo Root File System**. Or, a simple *in-memory*, *pseudo* file system can be enabled. This simple file system can be enabled setting the CONFIG\_NFILE\_DESCRIPTORS option to a non-zero value. This is an *in-memory* file system because it does not require any storage medium or block driver support. Rather, file system contents are generated on-the-fly as referenced via standard file system operations (open, close, read, write, etc.). In this sense, the file system is *pseudo* file system (in the same sense that the Linux /proc file system is also referred to as a pseudo file system).

Any user supplied data or logic can be accessed via the pseudo-file system. Built in support is provided for character and block [driver](http://nuttx.org/Documentation/NuttxPortingGuide.html#DeviceDrivers)*nodes* in the any pseudo file system directory. (By convention, however, all driver nodes should be in the /dev pseudo file system directory).

**Mounted File Systems** The simple in-memory file system can be extended my mounting block devices that provide access to true file systems backed up via some mass storage device. NuttX supports the standard mount() command that allows a block driver to be bound to a mount-point within the pseudo file system and to a a file system. At present, NuttX supports only the VFAT file system.

**Comparison to Linux** From a programming perspective, the NuttX file system appears very similar to a Linux file system. However, there is a fundamental difference: The NuttX root file system is a pseudo file system and true file systems may be mounted in the pseudo file system. In the typical Linux installation by comparison, the Linux root file system is a true file system and pseudo file systems may be mounted in the true, root file system. The approach selected by NuttX is intended to support greater scalability from the very tiny platform to the moderate platform.

**File System Interfaces**. The NuttX file system simply supports a set of standard, file system APIs (open(), close(), read(), write, etc.) and a registration mechanism that allows devices drivers to a associated with *nodes* in a file-system-like name space.

**2.10.2 Driver Operations**

* 2.10.2.1 fcntl.h
* 2.10.2.2 unistd.h
* 2.10.2.3 sys/ioctl.h
* 2.10.2.4 poll.h
* 2.10.2.5 sys/select.h

**2.10.2.1 fcntl.h**

#include <fcntl.h>

int creat(const char \*path, mode\_t mode);

int open(const char \*path, int oflag, ...);

int fcntl(int fd, int cmd, ...);

**2.10.2.2 unistd.h**

#include <unistd.h>

int close(int fd);

int dup(int fd);

int dup2(int fd1, int fd2);

off\_t lseek(int fd, off\_t offset, int whence);

ssize\_t pread(int fd, void \*buf, size\_t nbytes, off\_t offset);

ssize\_t pwrite(int fd, const void \*buf, size\_t nbytes, off\_t offset);

ssize\_t read(int fd, void \*buf, size\_t nbytes);

int unlink(const char \*path);

ssize\_t write(int fd, const void \*buf, size\_t nbytes);

**2.10.2.3 sys/ioctl.h**

#include <sys/ioctl.h>

#ifdef CONFIG\_LIBC\_IOCTL\_VARIADIC

int ioctl(int fd, int req, ...);

#else

int ioctl(int fd, int req, unsigned long arg);

#endif

**2.10.2.4 poll.h**

**2.10.2.4.1 poll**

**Function Prototype:**

#include <poll.h>

int poll(struct pollfd \*fds, nfds\_t nfds, int timeout);

**Description:** poll() waits for one of a set of file descriptors to become ready to perform I/O. If none of the events requested (and no error) has occurred for any of the file descriptors, then  poll() blocks until one of the events occurs.

**Configuration Settings**. In order to use the poll() API, the following must be defined in your NuttX configuration file:

* CONFIG\_NFILE\_DESCRIPTORS Defined to be greater than 0
* CONFIG\_DISABLE\_POLL NOT defined

In order to use the select with TCP/IP sockets test, you must also have the following additional things selected in your NuttX configuration file:

* CONFIG\_NET Defined for general network support
* CONFIG\_NET\_TCP Defined for TCP/IP support
* CONFIG\_NSOCKET\_DESCRIPTORS Defined to be greater than 0
* CONFIG\_NET\_TCP\_READAHEAD Define to enable read-ahead buffering
* CONFIG\_NET\_NTCP\_READAHEAD\_BUFFERS Defined to be greater than zero

In order to for select to work with incoming connections, you must also select:

* CONFIG\_NET\_TCPBACKLOG Incoming connections pend in a backlog until accept() is called. The size of the backlog is selected when listen() is called.

**Input Parameters:**

* fds. List of structures describing file descriptors to be monitored.
* nfds. The number of entries in the list.
* timeout. Specifies an upper limit on the time for which poll() will block in milliseconds. A negative value of timeout means an infinite timeout.

**Returned Value:**

On success, the number of structures that have nonzero revents fields. A value of 0 indicates that the call timed out and no file descriptors were ready. On error, -1 is returned, and errno is set appropriately:

* EBADF. An invalid file descriptor was given in one of the sets.
* EFAULT. The fds address is invalid
* EINTR. A signal occurred before any requested event.
* EINVAL. The nfds value exceeds a system limit.
* ENOMEM. There was no space to allocate internal data structures.
* ENOSYS. One or more of the drivers supporting the file descriptor does not support the poll method.

**2.10.2.5 sys/select.h**

**2.10.2.5.1 select**

**Function Prototype:**

#include <sys/select.h>

int select(int nfds, FAR fd\_set \*readfds, FAR fd\_set \*writefds,

FAR fd\_set \*exceptfds, FAR struct timeval \*timeout);

**Description:** select() allows a program to monitor multiple file descriptors, waiting until one or more of the file descriptors become "ready" for some class of I/O operation (e.g., input possible). A file descriptor is considered ready if it is possible to perform the corresponding I/O operation (e.g., read(2)) without blocking.

**NOTE:** poll() is the fundamental API for performing such monitoring operation under NuttX.  select() is provided for compatibility and is simply a layer of added logic on top of poll(). As such, select() is more wasteful of resources and poll() is the recommended API to be used.

**Input Parameters:**

* nfds. the maximum file descriptor number (+1) of any descriptor in any of the three sets.
* readfds. the set of descriptions to monitor for read-ready events
* writefds. the set of descriptions to monitor for write-ready events
* exceptfds. the set of descriptions to monitor for error events
* timeout. Return at this time if none of these events of interest occur.

**Returned Value:**

* 0: Timer expired
* >0: The number of bits set in the three sets of descriptors
* -1: An error occurred (errno will be set appropriately, see poll()).

**2.10.3 Directory Operations**

#include <dirent.h>

int closedir(DIR \*dirp);

FAR DIR \*opendir(const char \*path);

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

int readdir\_r(FAR DIR \*dirp, FAR struct dirent \*entry, FAR struct dirent \*\*result);

void rewinddir(FAR DIR \*dirp);

void seekdir(FAR DIR \*dirp, int loc);

int telldir(FAR DIR \*dirp);

**2.10.4 UNIX Standard Operations**

#include <unistd.h>

/\* Task Control Interfaces \*/

pid\_t vfork(void);

pid\_t getpid(void);

void \_exit(int status) noreturn\_function;

unsigned int sleep(unsigned int seconds);

void usleep(unsigned long usec);

int pause(void);

/\* File descriptor operations \*/

int close(int fd);

int dup(int fd);

int dup2(int fd1, int fd2);

int fsync(int fd);

off\_t lseek(int fd, off\_t offset, int whence);

ssize\_t read(int fd, FAR void \*buf, size\_t nbytes);

ssize\_t write(int fd, FAR const void \*buf, size\_t nbytes);

ssize\_t pread(int fd, FAR void \*buf, size\_t nbytes, off\_t offset);

ssize\_t pwrite(int fd, FAR const void \*buf, size\_t nbytes, off\_t offset);

/\* Check if a file descriptor corresponds to a terminal I/O file \*/

int isatty(int fd);

/\* Memory management \*/

#if defined(CONFIG\_ARCH\_ADDRENV) && defined(CONFIG\_MM\_PGALLOC) && \

defined(CONFIG\_ARCH\_USE\_MMU)

FAR void \*sbrk(intptr\_t incr);

#endif

/\* Special devices \*/

int pipe(int fd[2]);

/\* Working directory operations \*/

int chdir(FAR const char \*path);

FAR char \*getcwd(FAR char \*buf, size\_t size);

/\* File path operations \*/

int access(FAR const char \*path, int amode);

int rmdir(FAR const char \*pathname);

int unlink(FAR const char \*pathname);

#ifdef CONFIG\_PSEUDOFS\_SOFTLINKS

int link(FAR const char \*path1, FAR const char \*path2);

ssize\_t readlink(FAR const char \*path, FAR char \*buf, size\_t bufsize);

#endif

/\* Execution of programs from files \*/

#ifdef CONFIG\_LIBC\_EXECFUNCS

int execl(FAR const char \*path, ...);

int execv(FAR const char \*path, FAR char \*const argv[]);

#endif

/\* Networking \*/

#ifdef CONFIG\_NET

int gethostname(FAR char \*name, size\_t size);

int sethostname(FAR const char \*name, size\_t size);

#endif

/\* Other \*/

int getopt(int argc, FAR char \*const argv[], FAR const char \*optstring);

**2.10.5 Standard I/O**

#include <stdio.h>

/\* Operations on streams (FILE) \*/

void clearerr(register FILE \*stream);

int fclose(FILE \*stream);

int fflush(FILE \*stream);

int feof(FILE \*stream);

int ferror(FILE \*stream);

int fileno(FAR FILE \*stream);

int fgetc(FILE \*stream);

int fgetpos(FILE \*stream, fpos\_t \*pos);

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

FILE \*fopen(const char \*path, const char \*type);

int fprintf(FILE \*stream, const char \*format, ...);

int fputc(int c, FILE \*stream);

int fputs(const char \*s, FILE \*stream);

size\_t fread(void \*ptr, size\_t size, size\_t n\_items, FILE \*stream);

FAR FILE \*freopen(FAR const char \*path, FAR const char \*mode,

FAR FILE \*stream);

int fseek(FILE \*stream, long int offset, int whence);

int fsetpos(FILE \*stream, fpos\_t \*pos);

long ftell(FILE \*stream);

size\_t fwrite(const void \*ptr, size\_t size, size\_t n\_items, FILE \*stream);

FAR char \*gets(char \*s);

FAR char \*gets\_s(FAR char \*s, rsize\_t n);

void setbuf(FAR FILE \*stream, FAR char \*buf);

int setvbuf(FAR FILE \*stream, FAR char \*buffer, int mode, size\_t size);

int ungetc(int c, FAR FILE \*stream);

/\* Operations on the stdout stream, buffers, paths, and the whole printf-family \* /

int printf(const char \*format, ...);

int puts(const char \*s);

int rename(const char \*source, const char \*target);

int sprintf(char \*dest, const char \*format, ...);

int asprintf (FAR char \*\*ptr, FAR const char \*fmt, ...);

int snprintf(FAR char \*buf, size\_t size, const char \*format, ...);

int sscanf(const char \*buf, const char \*fmt, ...);

void perror(FAR const char \*s);

int vprintf(const char \*s, va\_list ap);

int vfprintf(FILE \*stream, const char \*s, va\_list ap);

int vsprintf(char \*buf, const char \*s, va\_list ap);

int vasprintf(FAR char \*\*ptr, const char \*fmt, va\_list ap);

int vsnprintf(FAR char \*buf, size\_t size, const char \*format, va\_list ap);

int vsscanf(char \*buf, const char \*s, va\_list ap);

/\* Operations on file descriptors including:

\*

\* POSIX-like File System Interfaces (fdopen), and

\* Extensions from the Open Group Technical Standard, 2006, Extended API Set

\* Part 1 (dprintf and vdprintf)

\*/

FAR FILE \*fdopen(int fd, FAR const char \*type);

int dprintf(int fd, FAR const char \*fmt, ...);

int vdprintf(int fd, FAR const char \*fmt, va\_list ap);

/\* Operations on paths \*/

FAR char \*tmpnam(FAR char \*s);

FAR char \*tempnam(FAR const char \*dir, FAR const char \*pfx);

int remove(FAR const char \*path);

#include <sys/stat.h>

int mkdir(FAR const char \*pathname, mode\_t mode);

int mkfifo(FAR const char \*pathname, mode\_t mode);

int stat(const char \*path, FAR struct stat \*buf);

int fstat(int fd, FAR struct stat \*buf);

#include <sys/statfs.h>

int statfs(FAR const char \*path, FAR struct statfs \*buf);

int fstatfs(int fd, FAR struct statfs \*buf);

**2.10.6 Standard Library**

stdlib.h generally addresses other operating system interfaces. However, the following may also be considered as file system interfaces:

#include <stdlib.h>

int mktemp(FAR char \*template);

int mkstemp(FAR char \*template);

**2.10.7 Asynchronous I/O**

#include <aio.h>

int aio\_cancel(int, FAR struct aiocb \*aiocbp);

int aio\_error(FAR const struct aiocb \*aiocbp);

int aio\_fsync(int, FAR struct aiocb \*aiocbp);

int aio\_read(FAR struct aiocb \*aiocbp);

ssize\_t aio\_return(FAR struct aiocb \*aiocbp);

int aio\_suspend(FAR const struct aiocb \*const list[], int nent,

FAR const struct timespec \*timeout);

int aio\_write(FAR struct aiocb \*aiocbp);

int lio\_listio(int mode, FAR struct aiocb \*const list[], int nent,

FAR struct sigevent \*sig);

**2.10.8 Standard String Operations**

#include <string.h>

char \*strchr(const char \*s, int c);

FAR char \*strdup(const char \*s);

const char \*strerror(int);

size\_t strlen(const char \*);

size\_t strnlen(const char \*, size\_t);

char \*strcat(char \*, const char \*);

char \*strncat(char \*, const char \*, size\_t);

int strcmp(const char \*, const char \*);

int strncmp(const char \*, const char \*, size\_t);

int strcasecmp(const char \*, const char \*);

int strncasecmp(const char \*, const char \*, size\_t);

char \*strcpy(char \*dest, const char \*src);

char \*strncpy(char \*, const char \*, size\_t);

char \*strpbrk(const char \*, const char \*);

char \*strchr(const char \*, int);

char \*strrchr(const char \*, int);

size\_t strspn(const char \*, const char \*);

size\_t strcspn(const char \*, const char \*);

char \*strstr(const char \*, const char \*);

char \*strtok(char \*, const char \*);

char \*strtok\_r(char \*, const char \*, char \*\*);

void \*memset(void \*s, int c, size\_t n);

void \*memcpy(void \*dest, const void \*src, size\_t n);

int memcmp(const void \*s1, const void \*s2, size\_t n);

void \*memmove(void \*dest, const void \*src, size\_t count);

#include <strings.h>

#define bcmp(b1,b2,len) memcmp(b1,b2,(size\_t)len)

#define bcopy(b1,b2,len) (void)memmove(b2,b1,len)

#define bzero(s,n) (void)memset(s,0,n)

#define index(s,c) strchr(s,c)

#define rindex(s,c) strrchr(s,c)

int ffs(int j);

int strcasecmp(const char \*, const char \*);

int strncasecmp(const char \*, const char \*, size\_t);

**2.10.9 Pipes and FIFOs**

**2.10.9.1****pipe**

**Function Prototype:**

#include <unistd.h>

int pipe(int fd[2]);

**Description:**

pipe() creates a pair of file descriptors, pointing to a pipe inode, and places them in the array pointed to by fd. fd[0] is for reading, fd[1] is for writing.

**Input Parameters:**

* fd[2]. The user provided array in which to catch the pipe file descriptors.

**Returned Value:**

0 is returned on success; otherwise, -1 is returned with errno set appropriately.

**2.10.9.2****mkfifo**

**Function Prototype:**

#include <sys/stat.h>

int mkfifo(FAR const char \*pathname, mode\_t mode);

**Description:**

mkfifo() makes a FIFO device driver file with name pathname. Unlike Linux, a NuttX FIFO is not a special file type but simply a device driver instance. mode specifies the FIFO's permissions (but is ignored in the current implementation).

Once the FIFO has been created by mkfifo(), any thread can open it for reading or writing, in the same way as an ordinary file. However, it must have been opened from both reading and writing before input or output can be performed. This FIFO implementation will block all attempts to open a FIFO read-only until at least one thread has opened the FIFO for writing.

If all threads that write to the FIFO have closed, subsequent calls to read() on the FIFO will return 0 (end-of-file).

**Input Parameters:**

* pathname. The full path to the FIFO instance to attach to or to create (if not already created).
* mode. Ignored for now

**Returned Value:**

0 is returned on success; otherwise, -1 is returned with errno set appropriately.

**2.10.10 mmap() and eXecute In Place (XIP)**

NuttX operates in a flat open address space and is focused on MCUs that do support Memory Management Units (MMUs). Therefore, NuttX generally does not require mmap() functionality and the MCUs generally cannot support true memory-mapped files.

However, memory mapping of files is the mechanism used by NXFLAT, the NuttX tiny binary format, to get files into memory in order to execute them. mmap() support is therefore required to support NXFLAT. There are two conditions where mmap() can be supported:

1. mmap() can be used to support *eXecute In Place* (XIP) on random access media under the following very restrictive conditions:
   1. The file-system supports the FIOC\_MMAP ioctl command. Any file system that maps files contiguously on the media should support this ioctl command. By comparison, most file system scatter files over the media in non-contiguous sectors. As of this writing, ROMFS is the only file system that meets this requirement.
   2. The underlying block driver supports the BIOC\_XIPBASE ioctl command that maps the underlying media to a randomly accessible address. At present, only the RAM/ROM disk driver does this.

Some limitations of this approach are as follows:

* 1. Since no real mapping occurs, all of the file contents are "mapped" into memory.
  2. All mapped files are read-only.
  3. There are no access privileges.

1. If CONFIG\_FS\_RAMMAP is defined in the configuration, then mmap() will support simulation of memory mapped files by copying files whole into RAM. These copied files have some of the properties of standard memory mapped files. There are many, many exceptions exceptions, however. Some of these include:
   1. The goal is to have a single region of memory that represents a single file and can be shared by many threads. That is, given a filename a thread should be able to open the file, get a file descriptor, and call mmap() to get a memory region. Different file descriptors opened with the same file path should get the same memory region when mapped.

The limitation in the current design is that there is insufficient knowledge to know that these different file descriptors correspond to the same file. So, for the time being, a new memory region is created each time that rammmap() is called. Not very useful!

* 1. The entire mapped portion of the file must be present in memory. Since it is assumed that the MCU does not have an MMU, on-demanding paging in of file blocks cannot be supported. Since the while mapped portion of the file must be present in memory, there are limitations in the size of files that may be memory mapped (especially on MCUs with no significant RAM resources).
  2. All mapped files are read-only. You can write to the in-memory image, but the file contents will not change.
  3. There are no access privileges.
  4. Since there are no processes in NuttX, all mmap() and munmap() operations have immediate, global effects. Under Linux, for example, munmap() would eliminate only the mapping with a process; the mappings to the same file in other processes would not be effected.
  5. Like true mapped file, the region will persist after closing the file descriptor. However, at present, these ram copied file regions are *not* automatically "unmapped" (i.e., freed) when a thread is terminated. This is primarily because it is not possible to know how many users of the mapped region there are and, therefore, when would be the appropriate time to free the region (other than when munmap is called).

NOTE: Note, if the design limitation of a) were solved, then it would be easy to solve exception d) as well.

**2.10.11.1 mmap**

**Function Prototype:**

#include <sys/mman.h>

FAR void \*mmap(FAR void \*start, size\_t length, int prot, int flags, int fd, off\_t offset);

**Description:**

Provides minimal mmap() as needed to support eXecute In Place (XIP) operation (as described above).

**Input Parameters:**

* start A hint at where to map the memory -- ignored. The address of the underlying media is fixed and cannot be re-mapped without MMU support.
* length The length of the mapping -- ignored. The entire underlying media is always accessible.
* prot See the PROT\_\* definitions in sys/mman.h.
  + PROT\_NONE - Will cause an error.
  + PROT\_READ - PROT\_WRITE and PROT\_EXEC also assumed.
  + PROT\_WRITE - PROT\_READ and PROT\_EXEC also assumed.
  + PROT\_EXEC - PROT\_READ and PROT\_WRITE also assumed.
* flags See the MAP\_\* definitions in sys/mman.h.
  + MAP\_SHARED - Required
  + MAP\_PRIVATE - Will cause an error
  + MAP\_FIXED - Will cause an error
  + MAP\_FILE - Ignored
  + MAP\_ANONYMOUS - Will cause an error
  + MAP\_ANON - Will cause an error
  + MAP\_GROWSDOWN - Ignored
  + MAP\_DENYWRITE - Will cause an error
  + MAP\_EXECUTABLE - Ignored
  + MAP\_LOCKED - Ignored
  + MAP\_NORESERVE - Ignored
  + MAP\_POPULATE - Ignored
  + AP\_NONBLOCK - Ignored
* fd file descriptor of the backing file -- required.
* offset The offset into the file to map.

**Returned Value:**

On success, mmap() returns a pointer to the mapped area. On error, the value MAP\_FAILED is returned, and errno is set appropriately.

* + ENOSYS - Returned if any of the unsupported mmap() features are attempted.
  + EBADF - fd is not a valid file descriptor.
  + EINVAL - Length is 0. flags contained neither MAP\_PRIVATE or MAP\_SHARED, or contained both of these values.
  + ENODEV - The underlying file-system of the specified file does not support memory mapping.

|  |
| --- |
| **3.0 OS Data Structures** |

|  |
| --- |
| **3.1 Scalar Types** |

Many of the types used to communicate with NuttX are simple scalar types. These types are used to provide architecture independence of the OS from the application. The scalar types used at the NuttX interface include:

* pid\_t
* size\_t
* sigset\_t
* time\_t

|  |
| --- |
| **3.2 Hidden Interface Structures** |

Several of the types used to interface with NuttX are structures that are intended to be hidden from the application. From the standpoint of the application, these structures (and structure pointers) should be treated as simple handles to reference OS resources. These hidden structures include:

* struct tcb\_s
* mqd\_t
* sem\_t
* WDOG\_ID
* pthread\_key\_t

In order to maintain portability, applications should not reference specific elements within these hidden structures. These hidden structures will not be described further in this user's manual.

|  |
| --- |
| **3.3 Access to the errno Variable** |

A pointer to the thread-specific errno value is available through a function call:

**Function Prototype:**

#include <errno.h>

#define errno \*get\_errno\_ptr()

int \*get\_errno\_ptr(void);

**Description**: get\_errno\_ptr() returns a pointer to the thread-specific errno value. Note that the symbol errno is defined to be get\_errno\_ptr() so that the usual access by referencing the symbol errno will work as expected.

There is a unique, private errno value for each NuttX task. However, the implementation of errno differs somewhat from the use oferrno in most multi-threaded process environments: In NuttX, each pthread will also have its own private copy of errno and the errnowill not be shared between pthreads. This is, perhaps, non-standard but promotes better thread independence.

**Input Parameters**: None

**Returned Values**:

* A pointer to the thread-specific errno value.

|  |
| --- |
| **3.4 User Interface Structures** |

**3.4.1 main\_t**

main\_t defines the type of a task entry point. main\_t is declared in sys/types.h as:

typedef int (\*main\_t)(int argc, char \*argv[]);

**3.4.2 struct sched\_param**

This structure is used to pass scheduling priorities to and from NuttX;

struct sched\_param

{

int sched\_priority;

};

**3.4.3 struct timespec**

This structure is used to pass timing information between the NuttX and a user application:

struct timespec

{

time\_t tv\_sec; /\* Seconds \*/

long tv\_nsec; /\* Nanoseconds \*/

};

**3.4.4 struct mq\_attr**

This structure is used to communicate message queue attributes between NuttX and a MoBY application:

struct mq\_attr {

size\_t mq\_maxmsg; /\* Max number of messages in queue \*/

size\_t mq\_msgsize; /\* Max message size \*/

unsigned mq\_flags; /\* Queue flags \*/

size\_t mq\_curmsgs; /\* Number of messages currently in queue \*/

};

**3.4.5 struct sigaction**

The following structure defines the action to take for given signal:

struct sigaction

{

union

{

void (\*\_sa\_handler)(int);

void (\*\_sa\_sigaction)(int, siginfo\_t \*, void \*);

} sa\_u;

sigset\_t sa\_mask;

int sa\_flags;

};

#define sa\_handler sa\_u.\_sa\_handler

#define sa\_sigaction sa\_u.\_sa\_sigaction

**3.4.6 struct siginfo/siginfo\_t**

The following types is used to pass parameters to/from signal handlers:

typedef struct siginfo

{

int si\_signo;

int si\_code;

union sigval si\_value;

} siginfo\_t;

**3.4.7 union sigval**

This defines the type of the struct siginfo si\_value field and is used to pass parameters with signals.

union sigval

{

int sival\_int;

void \*sival\_ptr;

};

**3.4.8 struct sigevent**

The following is used to attach a signal to a message queue to notify a task when a message is available on a queue.

struct sigevent

{

int sigev\_signo;

union sigval sigev\_value;

int sigev\_notify;

};