RTEMS POSIX API Guide

Release 6.1-rc4 (5th December 2024)

CONTENTS

The RTEMS Project is hosted at https://www.rtems.org. Any inquiries concerning RTEMS, its related support components, or its documentation should be directed to the RTEMS Project community.

RTEMS Online Resources

Home	https://www.rtems.org
Documentation	https://docs.rtems.org
Mailing Lists	https://lists.rtems.org
Bug Reporting	https://gitlab.rtems.org
Git Repositories	https://gitlab.rtems.org
Developers	https://gitlab.rtems.org

CHAPTER

ONE

PREFACE

This is the User's Guide for the POSIX API support provided in RTEMS.

The functionality described in this document is based on the following standards:

- POSIX 1003.1b-1993.
- POSIX 1003.1h/D3.
- Open Group Single UNIX Specification.

Much of the POSIX API standard is actually implemented in the Cygnus Newlib ANSI C Library. Please refer to documentation on Newlib for more information on the functionality it supplies.

This manual is still under construction and improvements are welcomed from users.

1.1 Acknowledgements

The RTEMS Project has been granted permission from The Open Group IEEE to excerpt and use portions of the POSIX standards documents in the RTEMS POSIX API User's Guide and RTEMS Shell User's Guide. We have to include a specific acknowledgement paragraph in these documents (e.g. preface or copyright page) and another slightly different paragraph for each manual page that excerpts and uses text from the standards.

This file should help ensure that the paragraphs are consistent and not duplicated

> The Institute of Electrical and Electronics Engineers, Inc and The Open Group, have given us permission to reprint portions of their documentation. Portions of this text are reprinted and reproduced in electronic form from IEEE Std 1003.1, 2004 Edition, Standard for Information Technology Operating System Interface (POSIX), The Open Group Base Specifications Issue 6, Copyright (c) 2001-2004 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group. In the event of any discrepancy between this version and the original IEEE and The Open Group Standard, the original IEEE and The Open Group Standard is the referee document. The original Standard can be obtained online at http://www.opengroup.org/unix/online.html. This notice shall appear on any product containing this material.

CHAPTER

TWO

PROCESS CREATION AND EXECUTION MANAGER

2.1 Introduction

The process creation and execution manager provides the functionality associated with the creation and termination of processes.

The directives provided by the process creation and execution manager are:

2.2 Background

POSIX process functionality can not be completely supported by RTEMS. This is because RTEMS provides no memory protection and implements a *single process, multi-threaded execution model*. In this light, RTEMS provides none of the routines that are associated with the creation of new processes. However, since the entire RTEMS application (e.g. executable) is logically a single POSIX process, RTEMS is able to provide implementations of many operations on processes. The rule of thumb is that those routines provide a meaningful result. For example, `getpid()` returns the node number.

2.3 Operations

The only functionality method defined by this manager which is supported by RTEMS is the `_exit` service. The implementation of `_exit` shuts the application down and is equivalent to invoking either `exit` or `rtems_shutdown_executive`.

2.4 Directives

This section details the process creation and execution manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

2.4.1 fork - Create a Process

CALLING SEQUENCE:

```
#include <sys/types.h>
int fork( void );
```

STATUS CODES:

ENOSYS	This routine is not supported by RTEMS.

DESCRIPTION:

This routine is not supported by RTEMS.

NOTES:

NONE

2.4.2 execl - Execute a File

CALLING SEQUENCE:

```
int execl(
    const char *path,
    const char *arg,
    ...
);
```

STATUS CODES:

ENOSYS	This routine is not supported by RTEMS.

DESCRIPTION:

This routine is not supported by RTEMS.

NOTES:

NONE

2.4.3 execv - Execute a File

CALLING SEQUENCE:

```
int execv(
    const char *path,
```

(continues on next page)

(continued from previous page)

```
    char const *argv[],
    ...
);
```

STATUS CODES:

ENOSYS	This routine is not supported by RTEMS.

DESCRIPTION:

This routine is not supported by RTEMS.

NOTES:

NONE

2.4.4 execle - Execute a File

CALLING SEQUENCE:

```
int execle(
    const char *path,
    const char *arg,
    ...
);
```

STATUS CODES:

ENOSYS	This routine is not supported by RTEMS.

DESCRIPTION:

This routine is not supported by RTEMS.

NOTES:

NONE

2.4.5 execve - Execute a File

CALLING SEQUENCE:

```
int execve(
    const char *path,
    char *const argv[],
    char *const envp[]
);
```

STATUS CODES:

ENOSYS	This routine is not supported by RTEMS.

DESCRIPTION:

This routine is not supported by RTEMS.

NOTES:

NONE

2.4.6 execlp - Execute a File

CALLING SEQUENCE:

```
int execlp(
    const char *file,
    const char *arg,
    ...
);
```

STATUS CODES:

ENOSYS	This routine is not supported by RTEMS.

DESCRIPTION:

This routine is not supported by RTEMS.

NOTES:

NONE

2.4.7 execvp - Execute a File

CALLING SEQUENCE:

```
int execvp(
    const char *file,
    char *const argv[],
    ...
);
```

STATUS CODES:

ENOSYS	This routine is not supported by RTEMS.

DESCRIPTION:

This routine is not supported by RTEMS.

NOTES:

NONE

2.4.8 pthread_atfork - Register Fork Handlers

CALLING SEQUENCE:

```
#include <sys/types.h>
int pthread_atfork(
    void (*prepare)(void),
    void (*parent)(void),
    void (*child)(void)
);
```

STATUS CODES:

`0`	This routine is a non-functional stub.

DESCRIPTION:

This routine is non-functional stub.

NOTES:

The POSIX specification for `pthread_atfork()` does not address the behavior when in a single process environment. Originally, the RTEMS implementation returned -1 and set errno to `ENOSYS`. This was an arbitrary decision part with no basis from the wider POSIX community. The FACE Technical Standard includes profiles without multiple process support and defined the behavior in a single process environment to return 0. Logically, the application can register atfork handlers but they will never be invoked.

2.4.9 wait - Wait for Process Termination

CALLING SEQUENCE:

```
#include <sys/types.h>
#include <sys/wait.h>
int wait(
    int *stat_loc
);
```

STATUS CODES:

`ENOSYS`	This routine is not supported by RTEMS.

DESCRIPTION:

This routine is not supported by RTEMS.

NOTES:

NONE

2.4.10 waitpid - Wait for Process Termination

CALLING SEQUENCE:

```
int wait(
    pid_t  pid,
    int   *stat_loc,
    int    options
);
```

STATUS CODES:

ENOSYS	This routine is not supported by RTEMS.

DESCRIPTION:

This routine is not supported by RTEMS.

NOTES:

NONE

2.4.11 _exit - Terminate a Process

CALLING SEQUENCE:

```
void _exit(
    int status
);
```

STATUS CODES:

NONE

DESCRIPTION:

The `_exit()` function terminates the calling process.

NOTES:

In RTEMS, a process is equivalent to the entire application on a single processor. Invoking this service terminates the application.

CHAPTER

THREE

SIGNAL MANAGER

3.1 Introduction

The signal manager provides the functionality associated with the generation, delivery, and management of process-oriented signals.

The directives provided by the signal manager are:

3.2 Background

3.2.1 Signals

POSIX signals are an asynchronous event mechanism. Each process and thread has a set of signals associated with it. Individual signals may be enabled (e.g. unmasked) or blocked (e.g. ignored) on both a per-thread and process level. Signals which are enabled have a signal handler associated with them. When the signal is generated and conditions are met, then the signal handler is invoked in the proper process or thread context asynchronous relative to the logical thread of execution.

If a signal has been blocked when it is generated, then it is queued and kept pending until the thread or process unblocks the signal or explicitly checks for it. Traditional, non-real-time POSIX signals do not queue. Thus if a process or thread has blocked a particular signal, then multiple occurrences of that signal are recorded as a single occurrence of that signal.

One can check for the set of outstanding signals that have been blocked. Services are provided to check for outstanding process or thread directed signals.

3.2.2 Signal Delivery

Signals which are directed at a thread are delivered to the specified thread.

Signals which are directed at a process are delivered to a thread which is selected based on the following algorithm:

1. If the action for this signal is currently `SIG_IGN`, then the signal is simply ignored.
2. If the currently executing thread has the signal unblocked, then the signal is delivered to it.
3. If any threads are currently blocked waiting for this signal (`sigwait()`), then the signal is delivered to the highest priority thread waiting for this signal.
4. If any other threads are willing to accept delivery of the signal, then the signal is delivered to the highest priority thread of this set. In the event, multiple threads of the same priority are willing to accept this signal, then priority is given first to ready threads, then to threads blocked on calls which may be interrupted, and finally to threads blocked on non-interruptible calls.
5. In the event the signal still can not be delivered, then it is left pending. The first thread to unblock the signal (`sigprocmask()` or `pthread_sigprocmask()`) or to wait for this signal (`sigwait()`) will be the recipient of the signal.

3.3 Operations

3.3.1 Signal Set Management

Each process and each thread within that process has a set of individual signals and handlers associated with it. Services are provided to construct signal sets for the purposes of building signal sets - type `sigset_t` - that are used to provide arguments to the services that mask, unmask, and check on pending signals.

3.3.2 Blocking Until Signal Generation

A thread may block until receipt of a signal. The “sigwait” and “pause” families of functions block until the requested signal is received or if using `sigtimedwait()` until the specified timeout period has elapsed.

3.3.3 Sending a Signal

This is accomplished via one of a number of services that sends a signal to either a process or thread. Signals may be directed at a process by the service `kill()` or at a thread by the service `pthread_kill()`

3.4 Directives

This section details the signal manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

3.4.1 sigaddset - Add a Signal to a Signal Set

CALLING SEQUENCE:

```
#include <signal.h>
int sigaddset(
    sigset_t *set,
    int       signo
);
```

STATUS CODES:

The function returns 0 on success, otherwise it returns -1 and sets `errno` to indicate the error. `errno` may be set to:

`EINVAL`	Invalid argument passed.

DESCRIPTION:

This function adds the signal `signo` to the specified signal `set`.

NOTES:

The set must be initialized using either `sigemptyset` or `sigfillset` before using this function.

3.4.2 sigdelset - Delete a Signal from a Signal Set

CALLING SEQUENCE:

```
#include <signal.h>
int sigdelset(
    sigset_t *set,
    int       signo
);
```

STATUS CODES:

The function returns 0 on success, otherwise it returns -1 and sets `errno` to indicate the error. `errno` may be set to:

`EINVAL`	Invalid argument passed.

DESCRIPTION:

This function deletes the signal specified by `signo` from the specified signal `set`.

NOTES:

The set must be initialized using either `sigemptyset` or `sigfillset` before using this function.

3.4.3 sigfillset - Fill a Signal Set

CALLING SEQUENCE:

```
#include <signal.h>
int sigfillset(
    sigset_t *set
);
```

STATUS CODES:

The function returns 0 on success, otherwise it returns -1 and sets `errno` to indicate the error. `errno` may be set to:

`EINVAL`	Invalid argument passed.

DESCRIPTION:

This function fills the specified signal `set` such that all signals are set.

3.4.4 sigismember - Is Signal a Member of a Signal Set

CALLING SEQUENCE:

```
#include <signal.h>
int sigismember(
    const sigset_t *set,
    int             signo
);
```

STATUS CODES:

The function returns either 1 or 0 if completed successfully, otherwise it returns -1 and sets `errno` to indicate the error. `errno` may be set to:

`EINVAL`	Invalid argument passed.

DESCRIPTION:

This function returns returns 1 if `signo` is a member of `set` and 0 otherwise.

NOTES:

The set must be initialized using either `sigemptyset` or `sigfillset` before using this function.

3.4.5 sigemptyset - Empty a Signal Set

CALLING SEQUENCE:

```
#include <signal.h>
int sigemptyset(
    sigset_t *set
);
```

STATUS CODES:

The function returns 0 on success, otherwise it returns -1 and sets `errno` to indicate the error. `errno` may be set to:

`EINVAL`	Invalid argument passed.

DESCRIPTION:

This function initializes an empty signal set pointed to by `set`.

3.4.6 sigaction - Examine and Change Signal Action

CALLING SEQUENCE:

```
#include <signal.h>
int sigaction(
    int                      sig,
    const struct sigaction *act,
    struct sigaction       *oact
);
```

STATUS CODES:

The function returns 0 on success, otherwise it returns -1 and sets `errno` to indicate the error. `errno` may be set to:

`EINVAL`	Invalid argument passed.
`ENOTSUP`	Realtime Signals Extension option not supported.

DESCRIPTION:

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

The structure `sigaction` has the following members:

`void(*)(int) sa_handler`	Pointer to a signal-catching function or one of the macros SIG_IGN or SIG_DFL.
`sigset_t sa_mask`	Additional set of signals to be blocked during execution of signal-catching function.
`int sa_flags`	Special flags to affect behavior of signal.
`void(*)(int, siginfo_t*, void*) sa_sigaction`	Alternative pointer to a signal-catching function.

`sa_handler` and `sa_sigaction` should never be used at the same time as their storage may overlap.

If the SA_SIGINFO flag (see below) is set in sa_flags, the sa_sigaction field specifies a signal-catching function, otherwise``sa_handler`` specifies the action to be associated with the signal, which may be a signal-catching function or one of the macros SIG_IGN or SIG_DFN.

The following flags can be set in the sa_flags field:

SA_SIGI	If not set, the signal-catching function should be declared as void func(int signo) and the address of the function should be set in``sa_handler``. If set, the signal-catching function should be declared as void func(int signo, siginfo_t* info, void* context) and the address of the function should be set in sa_sigaction.

The prototype of the siginfo_t structure is the following:

```
typedef struct
{
    int si_signo;        /* Signal number */
    int si_code;         /* Cause of the signal */
    union sigval
    {
        int sival_int;   /* Integer signal value */
        void* sival_ptr; /* Pointer signal value */
    } si_value;          /* Signal value */
} siginfo_t;
```

NOTES:

The signal number cannot be SIGKILL.

3.4.7 pthread_kill - Send a Signal to a Thread

CALLING SEQUENCE:

```
#include <signal.h>
int pthread_kill(
    pthread_t thread,
    int       sig
);
```

STATUS CODES:

The function returns 0 on success, otherwise it returns -1 and sets errno to indicate the error. errno may be set to:

ESRCH	The thread indicated by the parameter thread is invalid.
EINVAL	Invalid argument passed.

DESCRIPTION:

This functions sends the specified signal sig to a thread referenced to by thread.

If the signal code is 0, arguments are validated and no signal is sent.

3.4.8 sigprocmask - Examine and Change Process Blocked Signals

CALLING SEQUENCE:

```
#include <signal.h>
int sigprocmask(
    int             how,
    const sigset_t *set,
    sigset_t       *oset
);
```

STATUS CODES:

The function returns 0 on success, otherwise it returns -1 and sets `errno` to indicate the error. `errno` may be set to:

`EINVAL`	Invalid argument passed.

DESCRIPTION:

This function is used to alter the set of currently blocked signals on a process wide basis. A blocked signal will not be received by the process. The behavior of this function is dependent on the value of `how` which may be one of the following:

`SIG_BLOCK`	The set of blocked signals is set to the union of `set` and those signals currently blocked.
`SIG_UNBLOCK`	The signals specific in `set` are removed from the currently blocked set.
`SIG_SETMASK`	The set of currently blocked signals is set to `set`.

If `oset` is not `NULL`, then the set of blocked signals prior to this call is returned in `oset`. If `set` is `NULL`, no change is done, allowing to examine the set of currently blocked signals.

NOTES:

It is not an error to unblock a signal which is not blocked.

In the current implementation of RTEMS POSIX API `sigprocmask()` is technically mapped to `pthread_sigmask()`.

3.4.9 pthread_sigmask - Examine and Change Thread Blocked Signals

CALLING SEQUENCE:

```
#include <signal.h>
int pthread_sigmask(
int              how,
const sigset_t *set,
sigset_t       *oset
);
```

STATUS CODES:

The function returns 0 on success, otherwise it returns -1 and sets `errno` to indicate the error. `errno` may be set to:

EINVAL
Invalid argument passed.

DESCRIPTION:

This function is used to alter the set of currently blocked signals for the calling thread. A blocked signal will not be received by the process. The behavior of this function is dependent on the value of `how` which may be one of the following:

`SIG_BLOCK`	The set of blocked signals is set to the union of `set` and those signals currently blocked.
`SIG_UNBLOCK`	The signals specific in `set` are removed from the currently blocked set.
`SIG_SETMASK`	The set of currently blocked signals is set to `set`.

If `oset` is not `NULL`, then the set of blocked signals prior to this call is returned in `oset`. If `set` is `NULL`, no change is done, allowing to examine the set of currently blocked signals.

NOTES:

It is not an error to unblock a signal which is not blocked.

3.4.10 kill - Send a Signal to a Process

CALLING SEQUENCE:

```
#include <sys/types.h>
#include <signal.h>
int kill(
    pid_t pid,
    int   sig
);
```

STATUS CODES:

The function returns 0 on success, otherwise it returns -1 and sets `errno` to indicate the error. `errno` may be set to:

`EINVAL`	Invalid argument passed.
`EPERM`	Process does not have permission to send the signal to any receiving process.
`ESRCH`	The process indicated by the parameter pid is invalid.

DESCRIPTION:

This function sends the signal `sig` to the process `pid`.

NOTES:

Since RTEMS is a single-process system, a signal can only be sent to the calling process (i.e. the current node).

3.4.11 sigpending - Examine Pending Signals

CALLING SEQUENCE:

```
#include <signal.h>
    int sigpending(
    const sigset_t *set
);
```

STATUS CODES:

The function returns 0 on success, otherwise it returns -1 and sets `errno` to indicate the error. `errno` may be set to:

`EFAULT`	Invalid address for set.

DESCRIPTION:

This function allows the caller to examine the set of currently pending signals. A pending signal is one which has been raised but is currently blocked. The set of pending signals is returned in `set`.

3.4.12 sigsuspend - Wait for a Signal

CALLING SEQUENCE:

```
#include <signal.h>
   int sigsuspend(
   const sigset_t *sigmask
);
```

STATUS CODES:

The function returns 0 on success, otherwise it returns -1 and sets `errno` to indicate the error. `errno` may be set to:

`EINTR`	Signal interrupted this function.

DESCRIPTION:

This function temporarily replaces the signal mask for the process with that specified by `sigmask` and blocks the calling thread until a signal is raised.

3.4.13 pause - Suspend Process Execution

CALLING SEQUENCE:

```
#include <signal.h>
int pause( void );
```

STATUS CODES:

The function returns 0 on success, otherwise it returns -1 and sets `errno` to indicate the error. `errno` may be set to:

EINTR	Signal interrupted this function.

DESCRIPTION:

This function causes the calling thread to be blocked until an unblocked signal is received.

3.4.14 sigwait - Synchronously Accept a Signal

CALLING SEQUENCE:

```
#include <signal.h>
int sigwait(
    const sigset_t *set,
    int            *sig
);
```

STATUS CODES:

The function returns 0 on success, otherwise it returns -1 and sets `errno` to indicate the error. `errno` may be set to:

EINVAL	Invalid argument passed.
EINTR	Signal interrupted this function.

DESCRIPTION:

This function selects a pending signal based on the set specified in `set`, atomically clears it from the set of pending signals, and returns the signal number for that signal in `sig`.

3.4.15 sigwaitinfo - Synchronously Accept a Signal

CALLING SEQUENCE:

```
#include <signal.h>
int sigwaitinfo(
    const sigset_t *set,
    siginfo_t      *info
);
```

STATUS CODES:

The function returns 0 on success, otherwise it returns -1 and sets `errno` to indicate the error. `errno` may be set to:

EINTR
Signal interrupted this function.

DESCRIPTION:

This function selects a pending signal based on the set specified in `set`, atomically clears it from the set of pending signals, and returns information about that signal in `info`.

The prototype of the `siginfo_t` structure is the following:

```
typedef struct
{
    int si_signo;        /* Signal number */
    int si_code;         /* Cause of the signal */
    union sigval
    {
        int sival_int;   /* Integer signal value */
        void* sival_ptr; /* Pointer signal value */
    } si_value;          /* Signal value */
} siginfo_t;
```

3.4.16 sigtimedwait - Synchronously Accept a Signal with Timeout

CALLING SEQUENCE:

```
#include <signal.h>
int sigtimedwait(
    const sigset_t        *set,
    siginfo_t             *info,
    const struct timespec *timeout
);
```

STATUS CODES:

The function returns 0 on success, otherwise it returns -1 and sets `errno` to indicate the error. `errno` may be set to:

EAGAIN	Timed out while waiting for the specified signal set.
EINVAL	Nanoseconds field of the timeout argument is invalid.
EINTR	Signal interrupted this function.

DESCRIPTION:

This function selects a pending signal based on the set specified in `set`, atomically clears it from the set of pending signals, and returns information about that signal in `info`. The calling thread will block up to `timeout` waiting for the signal to arrive.

The `timespec` structure is defined as follows:

```
struct timespec
{
    time_t tv_sec; /* Seconds */
    long tv_nsec;  /* Nanoseconds */
};
```

NOTES:

If `timeout` is NULL, then the calling thread will wait forever for the specified signal set.

3.4.17 sigqueue - Queue a Signal to a Process

CALLING SEQUENCE:

```
#include <signal.h>
int sigqueue(
    pid_t              pid,
    int                signo,
    const union sigval value
);
```

STATUS CODES:

The function returns 0 on success, otherwise it returns -1 and sets `errno` to indicate the error. `errno` may be set to:

`EAGA`	No resources available to queue the signal. The process has already queued `SIGQUEUE_MAX` signals that are still pending at the receiver or the systemwide resource limit has been exceeded.
`EINV`	The value of the signo argument is an invalid or unsupported signal number.
`EPER`	The process does not have the appropriate privilege to send the signal to the receiving process.
`ESRC`	The process pid does not exist.

DESCRIPTION:

This function sends the signal specified by `signo` to the process `pid`

The `sigval` union is specified as:

```
union sigval
{
    int sival_int; /* Integer signal value */
    void* sival_ptr; /* Pointer signal value */
};
```

NOTES:

Since RTEMS is a single-process system, a signal can only be sent to the calling process (i.e. the current node).

3.4.18 alarm - Schedule Alarm

CALLING SEQUENCE:

```
#include <unistd.h>
unsigned int alarm(
    unsigned int seconds
);
```

STATUS CODES:

This call always succeeds.

If there was a previous `alarm()` request with time remaining, then this routine returns the number of seconds until that outstanding alarm would have fired. If no previous `alarm()` request was outstanding, then zero is returned.

DESCRIPTION:

The `alarm()` service causes the `SIGALRM` signal to be generated after the number of seconds specified by `seconds` has elapsed.

NOTES:

Alarm requests do not queue. If `alarm` is called while a previous request is outstanding, the call will result in rescheduling the time at which the `SIGALRM` signal will be generated.

If the notification signal, `SIGALRM`, is not caught or ignored, the calling process is terminated.

3.4.19 ualarm - Schedule Alarm in Microseconds

CALLING SEQUENCE:

```
#include <unistd.h>
useconds_t ualarm(
    useconds_t useconds,
    useconds_t interval
);
```

STATUS CODES:

This call always succeeds.

If there was a previous `ualarm()` request with time remaining, then this routine returns the number of seconds until that outstanding alarm would have fired. If no previous `alarm()` request was outstanding, then zero is returned.

DESCRIPTION:

The `ualarm()` service causes the `SIGALRM` signal to be generated after the number of microseconds specified by `useconds` has elapsed.

When `interval` is non-zero, repeated timeout notification occurs with a period in microseconds specified by `interval`.

NOTES:

Alarm requests do not queue. If `alarm` is called while a previous request is outstanding, the call will result in rescheduling the time at which the `SIGALRM` signal will be generated.

If the notification signal, `SIGALRM`, is not caught or ignored, the calling process is terminated.

CHAPTER

FOUR

PROCESS ENVIRONMENT MANAGER

4.1 Introduction

The process environment manager is responsible for providing the functions related to user and group Id management.

The directives provided by the process environment manager are:

4.2 Background

4.2.1 Users and Groups

RTEMS provides a single process, multi-threaded execution environment. In this light, the notion of user and group is somewhat without meaning. But RTEMS does provide services to provide a synthetic version of user and group. By default, a single user and group is associated with the application. Thus unless special actions are taken, every thread in the application shares the same user and group Id. The initial rationale for providing user and group Id functionality in RTEMS was for the filesystem infrastructure to implement file permission checks. The effective user/group Id capability has since been used to implement permissions checking by the `ftpd` server.

In addition to the "real" user and group Ids, a process may have an effective user/group Id. This allows a process to function using a more limited permission set for certain operations.

4.2.2 User and Group Names

POSIX considers user and group Ids to be a unique integer that may be associated with a name. This is usually accomplished via a file named `/etc/passwd` for user Id mapping and `/etc/groups` for group Id mapping. Again, although RTEMS is effectively a single process and thus single user system, it provides limited support for user and group names. When configured with an appropriate filesystem, RTEMS will access the appropriate files to map user and group Ids to names.

If these files do not exist, then RTEMS will synthesize a minimal version so this family of services return without error. It is important to remember that a design goal of the RTEMS POSIX services is to provide useable and meaningful results even though a full process model is not available.

4.2.3 Environment Variables

POSIX allows for variables in the run-time environment. These are name/value pairs that make be dynamically set and obtained by programs. In a full POSIX environment with command line shell and multiple processes, environment variables may be set in one process - such as the shell - and inherited by child processes. In RTEMS, there is only one process and thus only one set of environment variables across all processes.

4.3 Operations

4.3.1 Accessing User and Group Ids

The user Id associated with the current thread may be obtain using the `getuid()` service. Similarly, the group Id may be obtained using the `getgid()` service.

4.3.2 Accessing Environment Variables

The value associated with an environment variable may be obtained using the `getenv()` service and set using the `putenv()` service.

4.4 Directives

This section details the process environment manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

4.4.1 getpid - Get Process ID

CALLING SEQUENCE:

```
int getpid( void );
```

STATUS CODES:

The process Id is returned.

DESCRIPTION:

This service returns the process Id.

NOTES:

NONE

4.4.2 getppid - Get Parent Process ID

CALLING SEQUENCE:

```
int getppid( void );
```

STATUS CODES:

The parent process Id is returned.

DESCRIPTION:

This service returns the parent process Id.

NOTES:

NONE

4.4.3 getuid - Get User ID

CALLING SEQUENCE:

```
int getuid( void );
```

STATUS CODES:

The effective user Id is returned.

DESCRIPTION:

This service returns the effective user Id.

NOTES:

NONE

4.4.4 geteuid - Get Effective User ID

CALLING SEQUENCE:

```
int geteuid( void );
```

STATUS CODES:

The effective group Id is returned.

DESCRIPTION:

This service returns the effective group Id.

NOTES:

NONE

4.4.5 getgid - Get Real Group ID

CALLING SEQUENCE:

```
int getgid( void );
```

STATUS CODES:

The group Id is returned.

DESCRIPTION:

This service returns the group Id.

NOTES:

NONE

4.4.6 getegid - Get Effective Group ID

CALLING SEQUENCE:

```
int getegid( void );
```

STATUS CODES:

The effective group Id is returned.

DESCRIPTION:

This service returns the effective group Id.

NOTES:

NONE

4.4.7 setuid - Set User ID

CALLING SEQUENCE:

```
int setuid(
    uid_t uid
);
```

STATUS CODES:

This service returns 0.

DESCRIPTION:

This service sets the user Id to `uid`.

NOTES:

NONE

4.4.8 setgid - Set Group ID

CALLING SEQUENCE:

```
int setgid(
    gid_t  gid
);
```

STATUS CODES:

This service returns 0.

DESCRIPTION:

This service sets the group Id to `gid`.

NOTES:

NONE

4.4.9 getgroups - Get Supplementary Group IDs

CALLING SEQUENCE:

```
int getgroups(
    int    gidsetsize,
    gid_t  grouplist[]
);
```

STATUS CODES:

NA

DESCRIPTION:

This service is not implemented as RTEMS has no notion of supplemental groups.

NOTES:

If supported, this routine would only be allowed for the super-user.

4.4.10 getlogin - Get User Name

CALLING SEQUENCE:

```
char *getlogin( void );
```

STATUS CODES:

Returns a pointer to a string containing the name of the current user.

DESCRIPTION:

This routine returns the name of the current user.

NOTES:

This routine is not reentrant and subsequent calls to `getlogin()` will overwrite the same buffer.

4.4.11 getlogin_r - Reentrant Get User Name

CALLING SEQUENCE:

```
int getlogin_r(
    char   *name,
    size_t  namesize
);
```

STATUS CODES:

`EINVAL`	The arguments were invalid.

DESCRIPTION:

This is a reentrant version of the `getlogin()` service. The caller specified their own buffer, `name`, as well as the length of this buffer, `namesize`.

NOTES:

NONE

4.4.12 getpgrp - Get Process Group ID

CALLING SEQUENCE:

```
pid_t getpgrp( void );
```

STATUS CODES:

The procress group Id is returned.

DESCRIPTION:

This service returns the current progress group Id.

NOTES:

This routine is implemented in a somewhat meaningful way for RTEMS but is truly not functional.

4.4.13 getrusage - Get Resource Utilization

CALLING SEQUENCE:

```
int getrusage( int who, struct rusage *rusage );
```

STATUS CODES:

Returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

DESCRIPTION:

This function provides a measures of the resources being used by RTEMS. RTEMS is a single process environment so child process requests result in an error being returned.

A `who` value of `RUSAGE_SELF` results in the `struct rusage` field `ru_utime` returning the total active time of all threads that exist when the call is made and the field `ru_stime` returning the total idle time.

A `who` value of `RUSAGE_THREAD` results in the `struct rusage` field `ru_utime` returning the total active time of the current thread and the field `ru_stime` is set to 0.

NOTES:

The time returned can be more than the system up time if there is more than one CPU.

This routine is implemented using the internal thread iterator and accounts for time spent by the currently active threads. RTEMS does not account for the execution time of threads that are no longer running.

The idle time is the total time spent in `IDLE` since RTEMS started. The time for each CPU is summed.

The ratio of the difference between samples made by `getrusage` of the user and system idle time can be used to compute the current load.

4.4.14 setsid - Create Session and Set Process Group ID

CALLING SEQUENCE:

```
pid_t setsid( void );
```

STATUS CODES:

`EPERM`	The application does not have permission to create a process group.

DESCRIPTION:

This routine always returns `EPERM` as RTEMS has no way to create new processes and thus no way to create a new process group.

NOTES:

NONE

4.4.15 setpgid - Set Process Group ID for Job Control

CALLING SEQUENCE:

```
int setpgid(
    pid_t pid,
    pid_t pgid
);
```

STATUS CODES:

ENOSYS	The routine is not implemented.

DESCRIPTION:

This service is not implemented for RTEMS as process groups are not supported.

NOTES:

NONE

4.4.16 uname - Get System Name

CALLING SEQUENCE:

```
int uname(
    struct utsname *name
);
```

STATUS CODES:

EPERM	The provided structure pointer is invalid.

DESCRIPTION:

This service returns system information to the caller. It does this by filling in the `struct utsname` format structure for the caller.

NOTES:

The information provided includes the operating system (RTEMS in all configurations), the node number, the release as the RTEMS version, and the CPU family and model. The CPU model name will indicate the multilib executive variant being used.

4.4.17 times - Get process times

CALLING SEQUENCE:

```
#include <sys/time.h>
clock_t times(
    struct tms *ptms
);
```

STATUS CODES:

This routine returns the number of clock ticks that have elapsed since the system was initialized (e.g. the application was started).

DESCRIPTION:

`times` stores the current process times in `ptms`. The format of `struct tms` is as defined in `<sys/times.h>`. RTEMS fills in the field `tms_utime` with the number of ticks that the calling thread has executed and the field `tms_stime` with the number of clock ticks since system boot (also returned). All other fields in the `ptms` are left zero.

NOTES:

RTEMS has no way to distinguish between user and system time so this routine returns the most meaningful information possible.

4.4.18 getenv - Get Environment Variables

CALLING SEQUENCE:

```
char *getenv(
    const char *name
);
```

STATUS CODES:

`NULL`	when no match
pointer to value	when successful

DESCRIPTION:

This service searches the set of environment variables for a string that matches the specified `name`. If found, it returns the associated value.

NOTES:

The environment list consists of name value pairs that are of the form `name = value`.

4.4.19 setenv - Set Environment Variables

CALLING SEQUENCE:

```
int setenv(
    const char *name,
    const char *value,
    int overwrite
);
```

STATUS CODES:

Returns 0 if successful and -1 otherwise.

DESCRIPTION:

This service adds the variable `name` to the environment with `value`. If `name` is not already exist, then it is created. If `name` exists and `overwrite` is zero, then the previous value is not overwritten.

NOTES:

NONE

4.4.20 ctermid - Generate Terminal Pathname

CALLING SEQUENCE:

```
char *ctermid(
    char *s
);
```

STATUS CODES:

Returns a pointer to a string indicating the pathname for the controlling terminal.

DESCRIPTION:

This service returns the name of the terminal device associated with this process. If `s` is NULL, then a pointer to a static buffer is returned. Otherwise, `s` is assumed to have a buffer of sufficient size to contain the name of the controlling terminal.

NOTES:

By default on RTEMS systems, the controlling terminal is `/dev/console`. Again this implementation is of limited meaning, but it provides true and useful results which should be sufficient to ease porting applications from a full POSIX implementation to the reduced profile supported by RTEMS.

4.4.21 ttyname - Determine Terminal Device Name

CALLING SEQUENCE:

```
char *ttyname(
    int fd
);
```

STATUS CODES:

Pointer to a string containing the terminal device name or `NULL` is returned on any error.

DESCRIPTION:

This service returns a pointer to the pathname of the terminal device that is open on the file descriptor `fd`. If `fd` is not a valid descriptor for a terminal device, then NULL is returned.

NOTES:

This routine uses a static buffer.

4.4.22 ttyname_r - Reentrant Determine Terminal Device Name

CALLING SEQUENCE:

```
int ttyname_r(
    int   fd,
    char *name,
```

(continues on next page)

(continued from previous page)

```
    int   namesize
);
```

STATUS CODES:

This routine returns -1 and sets `errno` as follows:

`EBADF`	If not a valid descriptor for a terminal device.
`EINVAL`	If `name` is `NULL` or `namesize` are insufficient.

DESCRIPTION:

This service the pathname of the terminal device that is open on the file descriptor `fd`.

NOTES:

NONE

4.4.23 isatty - Determine if File Descriptor is Terminal

CALLING SEQUENCE:

```
int isatty(
    int fd
);
```

STATUS CODES:

Returns 1 if `fd` is a terminal device and 0 otherwise.

DESCRIPTION:

This service returns 1 if `fd` is an open file descriptor connected to a terminal and 0 otherwise.

NOTES:

4.4.24 sysconf - Get Configurable System Variables

CALLING SEQUENCE:

```
long sysconf(
    int name
);
```

STATUS CODES:

The value returned is the actual value of the system resource. If the requested configuration name is a feature flag, then 1 is returned if the available and 0 if it is not. On any other error condition, -1 is returned.

DESCRIPTION:

This service is the mechanism by which an application determines values for system limits or options at runtime.

NOTES:

Much of the information that may be obtained via `sysconf` has equivalent macros in `unistd.h`. However, those macros reflect conservative limits which may have been altered by application configuration.

CHAPTER

FIVE

FILES AND DIRECTORIES MANAGER

5.1 Introduction

The files and directories manager is . . .

The directives provided by the files and directories manager are:

- *opendir* (page 50) - Open a Directory
- *readdir* (page 50) - Reads a directory
- *rewinddir* (page 51) - Resets the `readdir()` pointer
- *scandir* (page 51) - Scan a directory for matching entries
- *telldir* (page 52) - Return current location in directory stream
- *closedir* (page 52) - Ends directory read operation
- *getdents* (page 66) - Get directory entries
- *chdir* (page 53) - Changes the current working directory
- *fchdir* (page 53) - Changes the current working directory
- *getcwd* (page 54) - Gets current working directory
- *open* (page 54) - Opens a file
- *creat* (page 56) - Create a new file or rewrite an existing one
- *umask* (page 56) - Sets a file creation mask
- *link* (page 57) - Creates a link to a file
- *symlink* (page 58) - Creates a symbolic link to a file
- *readlink* (page 58) - Obtain the name of the link destination
- *mkdir* (page 59) - Makes a directory
- *mkfifo* (page 60) - Makes a FIFO special file
- *unlink* (page 60) - Removes a directory entry
- *rmdir* (page 61) - Delete a directory
- *rename* (page 61) - Renames a file
- *stat* (page 62) - Gets information about a file.
- *fstat* (page 63) - Gets file status
- *lstat* (page 63) - Gets file status
- *access* (page 64) - Check permissions for a file.
- *chmod* (page 64) - Changes file mode
- *fchmod* (page 65) - Changes permissions of a file
- *chown* (page 66) - Changes the owner and/ or group of a file
- *utime* (page 67) - Change access and/or modification times of an inode
- *ftruncate* (page 68) - Truncate a file to a specified length
- *truncate* (page 68) - Truncate a file to a specified length

- *pathconf* (page 69) - Gets configuration values for files
- *fpathconf* (page 70) - Get configuration values for files
- *mknod* (page 71) - Create a directory

5.2 Background

5.2.1 Path Name Evaluation

A pathname is a string that consists of no more than PATH_MAX bytes, including the terminating null character. A pathname has an optional beginning slash, followed by zero or more filenames separated by slashes. If the pathname refers to a directory, it may also have one or more trailing slashes. Multiple successive slahes are considered to be the same as one slash.

POSIX allows a pathname that begins with precisely two successive slashes to be interpreted in an implementation-defined manner. RTEMS does not currently recognize this as a special condition. Any number of successive slashes is treated the same as a single slash. POSIX requires that an implementation treat more than two leading slashes as a single slash.

5.3 Operations

There is currently no text in this section.

5.4 Directives

This section details the files and directories manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

5.4.1 opendir - Open a Directory

CALLING SEQUENCE:

```
#include <sys/types.h>
#include <dirent.h>
int opendir(
    const char *dirname
);
```

STATUS CODES:

EACCES	Search permission was denied on a component of the path prefix of dirname, or read permission is denied
EMFILE	Too many file descriptors in use by process
ENFILE	Too many files are currently open in the system.
ENOENT	Directory does not exist, or name is an empty string.
ENOMEM	Insufficient memory to complete the operation.
ENOTDIR	name is not a directory.

DESCRIPTION:

This routine opens a directory stream corresponding to the directory specified by the dirname argument. The directory stream is positioned at the first entry.

NOTES:

The routine is implemented in Cygnus newlib.

5.4.2 readdir - Reads a directory

CALLING SEQUENCE:

```
#include <sys/types.h>
#include <dirent.h>
    int readdir(
    DIR *dirp
);
```

STATUS CODES:

EBADF	Invalid file descriptor

DESCRIPTION:

The readdir() function returns a pointer to a structure dirent representing the next directory entry from the directory stream pointed to by dirp. On end-of-file, NULL is returned.

The `readdir()` function may (or may not) return entries for `.` or `..` Your program should tolerate reading dot and dot-dot but not require them.

The data pointed to be `readdir()` may be overwritten by another call to `readdir()` for the same directory stream. It will not be overwritten by a call for another directory.

NOTES:

If `ptr` is not a pointer returned by `malloc()`, `calloc()`, or `realloc()` or has been deallocated with `free()` or `realloc()`, the results are not portable and are probably disastrous.

The routine is implemented in Cygnus newlib.

5.4.3 rewinddir - Resets the readdir() pointer

CALLING SEQUENCE:

```
#include <sys/types.h>
#include <dirent.h>
void rewinddir(
    DIR *dirp
);
```

STATUS CODES:

No value is returned.

DESCRIPTION:

The `rewinddir()` function resets the position associated with the directory stream pointed to by `dirp`. It also causes the directory stream to refer to the current state of the directory.

NOTES:

NONE

If `dirp` is not a pointer by `opendir()`, the results are undefined.

The routine is implemented in Cygnus newlib.

5.4.4 scandir - Scan a directory for matching entries

CALLING SEQUENCE:

```
#include <dirent.h>
int scandir(
    const char       *dir,
    struct dirent ***namelist,
    int  (*select)(const struct dirent *),
    int  (*compar)(const struct dirent **, const struct dirent **)
);
```

STATUS CODES:

`ENOMEM`	Insufficient memory to complete the operation.

DESCRIPTION:

The `scandir()` function scans the directory `dir`, calling `select()` on each directory entry. Entries for which `select()` returns non-zero are stored in strings allocated via `malloc()`, sorted using `qsort()` with the comparison function `compar()`, and collected in array `namelist` which is allocated via `malloc()`. If `select` is `NULL`, all entries are selected.

NOTES:

The routine is implemented in Cygnus newlib.

5.4.5 telldir - Return current location in directory stream

CALLING SEQUENCE:

```
#include <dirent.h>
off_t telldir(
    DIR *dir
);
```

STATUS CODES:

`EBADF`	Invalid directory stream descriptor `dir`.

DESCRIPTION:

The `telldir()` function returns the current location associated with the directory stream `dir`.

NOTES:

The routine is implemented in Cygnus newlib.

5.4.6 closedir - Ends directory read operation

CALLING SEQUENCE:

```
#include <sys/types.h>
#include <dirent.h>
int closedir(
    DIR *dirp
);
```

STATUS CODES:

`EBADF`	Invalid file descriptor

DESCRIPTION:

The directory stream associated with `dirp` is closed. The value in `dirp` may not be usable after a call to `closedir()`.

NOTES:

NONE

The argument to `closedir()` must be a pointer returned by `opendir()`. If it is not, the results are not portable and most likely unpleasant.

The routine is implemented in Cygnus newlib.

5.4.7 chdir - Changes the current working directory

CALLING SEQUENCE:

```
#include <unistd.h>
int chdir(
    const char *path
);
```

STATUS CODES:

On error, this routine returns -1 and sets `errno` to one of the following:

EACCES	Search permission is denied for a directory in a file's path prefix.
ENAMETOOLON(	Length of a filename string exceeds `PATH_MAX` and `_POSIX_NO_TRUNC` is in effect.
ENOENT	A file or directory does not exist.
ENOTDIR	A component of the specified pathname was not a directory when directory was expected.

DESCRIPTION:

The `chdir()` function causes the directory named by `path` to become the current working directory; that is, the starting point for searches of pathnames not beginning with a slash.

If `chdir()` detects an error, the current working directory is not changed.

NOTES:

NONE

5.4.8 fchdir - Changes the current working directory

CALLING SEQUENCE:

```
#include <unistd.h>
int fchdir(
    int fd
);
```

STATUS CODES:

On error, this routine returns -1 and sets `errno` to one of the following:

EACCES	Search permission is denied for a directory in a file's path prefix.
ENAMETOOLON(	Length of a filename string exceeds `PATH_MAX` and `_POSIX_NO_TRUNC` is in effect.
ENOENT	A file or directory does not exist.
ENOTDIR	A component of the specified pathname was not a directory when directory was expected.

DESCRIPTION:

The `fchdir()` function causes the directory named by `fd` to become the current working directory; that is, the starting point for searches of pathnames not beginning with a slash.

If `fchdir()` detects an error, the current working directory is not changed.

NOTES:

NONE

5.4.9 getcwd - Gets current working directory

CALLING SEQUENCE:

```
#include <unistd.h>
int getcwd( void );
```

STATUS CODES:

`EINVAL`	Invalid argument
`ERANGE`	Result is too large
`EACCES`	Search permission is denied for a directory in a file's path prefix.

DESCRIPTION:

The `getcwd()` function copies the absolute pathname of the current working directory to the character array pointed to by `buf`. The `size` argument is the number of bytes available in `buf`

NOTES:

There is no way to determine the maximum string length that `fetcwd()` may need to return. Applications should tolerate getting `ERANGE` and allocate a larger buffer.

It is possible for `getcwd()` to return EACCES if, say, `login` puts the process into a directory without read access.

The 1988 standard uses `int` instead of `size_t` for the second parameter.

5.4.10 open - Opens a file

CALLING SEQUENCE:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(
    const char *path,
    int         oflag,
    mode_t      mode
);
```

STATUS CODES:

EACCES	Search permission is denied for a directory in a file's path prefix.
EEXIST	The named file already exists.
EINTR	Function was interrupted by a signal.
EISDIR	Attempt to open a directory for writing or to rename a file to be a directory.
EMFILE	Too many file descriptors are in use by this process.
ENAMETOOLO	Length of a filename string exceeds PATH_MAX and _POSIX_NO_TRUNC is in effect.
ENFILE	Too many files are currently open in the system.
ENOENT	A file or directory does not exist.
ENOSPC	No space left on disk.
ENOTDIR	A component of the specified pathname was not a directory when a directory was expected.
ENXIO	No such device. This error may also occur when a device is not ready, for example, a tape drive is off-line.
EROFS	Read-only file system.

DESCRIPTION:

The open function establishes a connection between a file and a file descriptor. The file descriptor is a small integer that is used by I/O functions to reference the file. The path argument points to the pathname for the file.

The oflag argument is the bitwise inclusive OR of the values of symbolic constants. The programmer must specify exactly one of the following three symbols:

O_RDONLY	Open for reading only.
O_WRONLY	Open for writing only.
O_RDWR	Open for reading and writing.

Any combination of the following symbols may also be used.

O_APPEI	Set the file offset to the end-of-file prior to each write.
O_CREA	If the file does not exist, allow it to be created. This flag indicates that the mode argument is present in the call to open.
O_EXCL	This flag may be used only if O_CREAT is also set. It causes the call to open to fail if the file already exists.
O_NOCT	Do not assign controlling terminal.
O_NONBI	Do no wait for the device or file to be ready or available. After the file is open, the read and write calls return immediately. If the process would be delayed in the read or write opermation, -1 is returned and``errno`` is set to EAGAIN instead of blocking the caller.
O_TRUN	This flag should be used only on ordinary files opened for writing. It causes the file to be tuncated to zero length..

Upon successful completion, open returns a non-negative file descriptor.

NOTES:

NONE

5.4.11 creat - Create a new file or rewrite an existing one

CALLING SEQUENCE:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int creat(
    const char *path,
    mode_t      mode
);
```

STATUS CODES:

EEXIST	path already exists and O_CREAT and O_EXCL were used.
EISDIR	path refers to a directory and the access requested involved writing
ETXTBSY	path refers to an executable image which is currently being executed and write access was requested
EFAULT	path points outside your accessible address space
EACCES	The requested access to the file is not allowed, or one of the directories in path did not allow search (execute) permission.
ENAMETOOL	path was too long.
ENOENT	A directory component in path does not exist or is a dangling symbolic link.
ENOTDIR	A component used as a directory in path is not, in fact, a directory.
EMFILE	The process alreadyh has the maximum number of files open.
ENFILE	The limit on the total number of files open on the system has been reached.
ENOMEM	Insufficient kernel memory was available.
EROFS	path refers to a file on a read-only filesystem and write access was requested

DESCRIPTION:

creat attempts to create a file and return a file descriptor for use in read, write, etc.

NOTES:

NONE

The routine is implemented in Cygnus newlib.

5.4.12 umask - Sets a file creation mask.

CALLING SEQUENCE:

```
#include <sys/types.h>
#include <sys/stat.h>
mode_t umask(
    mode_t cmask
);
```

STATUS CODES:

DESCRIPTION:

The umask() function sets the process file creation mask to cmask. The file creation mask is used during open(), creat(), mkdir(), mkfifo() calls to turn off permission bits in the mode

argument. Bit positions that are set in `cmask` are cleared in the mode of the created file.

NOTES:

NONE

The `cmask` argument should have only permission bits set. All other bits should be zero.

In a system which supports multiple processes, the file creation mask is inherited across `fork()` and `exec()` calls. This makes it possible to alter the default permission bits of created files. RTEMS does not support multiple processes so this behavior is not possible.

5.4.13 link - Creates a link to a file

CALLING SEQUENCE:

```
#include <unistd.h>
int link(
    const char *existing,
    const char *new
);
```

STATUS CODES:

EACCES	Search permission is denied for a directory in a file's path prefix
EEXIST	The named file already exists.
EMLINK	The number of links would exceed LINK_MAX.
ENAMETOOL	Length of a filename string exceeds PATH_MAX and _POSIX_NO_TRUNC is in effect.
ENOENT	A file or directory does not exist.
ENOSPC	No space left on disk.
ENOTDIR	A component of the specified pathname was not a directory when a directory was expected.
EPERM	Operation is not permitted. Process does not have the appropriate priviledges or permissions to perform the requested operations.
EROFS	Read-only file system.
EXDEV	Attempt to link a file to another file system.

DESCRIPTION:

The `link()` function atomically creates a new link for an existing file and increments the link count for the file.

If the `link()` function fails, no directories are modified.

The `existing` argument should not be a directory.

The caller may (or may not) need permission to access the existing file.

NOTES:

NONE

5.4.14 symlink - Creates a symbolic link to a file

CALLING SEQUENCE:

```
#include <unistd.h>
int symlink(
    const char *topath,
    const char *frompath
);
```

STATUS CODES:

EACCES	Search permission is denied for a directory in a file's path prefix
EEXIST	The named file already exists.
ENAMETOOL	Length of a filename string exceeds PATH_MAX and _POSIX_NO_TRUNC is in effect.
ENOENT	A file or directory does not exist.
ENOSPC	No space left on disk.
ENOTDIR	A component of the specified pathname was not a directory when a directory was expected.
EPERM	Operation is not permitted. Process does not have the appropriate priviledges or permissions to perform the requested operations.
EROFS	Read-only file system.

DESCRIPTION:

The `symlink()` function creates a symbolic link from the frombath to the topath. The symbolic link will be interpreted at run-time.

If the `symlink()` function fails, no directories are modified.

The caller may (or may not) need permission to access the existing file.

NOTES:

NONE

5.4.15 readlink - Obtain the name of a symbolic link destination

CALLING SEQUENCE:

```
#include <unistd.h>
int readlink(
    const char *path,
    char       *buf,
    size_t      bufsize
);
```

STATUS CODES:

EACCES	Search permission is denied for a directory in a file's path prefix
ENAMETOOLONG	Length of a filename string exceeds PATH_MAX and _POSIX_NO_TRUNC is in effect.
ENOENT	A file or directory does not exist.
ENOTDIR	A component of the prefix pathname was not a directory when a directory was expected.
ELOOP	Too many symbolic links were encountered in the pathname.
EINVAL	The pathname does not refer to a symbolic link
EFAULT	An invalid pointer was passed into the readlink() routine.

DESCRIPTION:

The readlink() function places the symbolic link destination into buf argument and returns the number of characters copied.

If the symbolic link destination is longer than bufsize characters the name will be truncated.

NOTES:

NONE

5.4.16 mkdir - Makes a directory

CALLING SEQUENCE:

```
#include <sys/types.h>
#include <sys/stat.h>
int mkdir(
    const char *path,
    mode_t      mode
);
```

STATUS CODES:

EACCES	Search permission is denied for a directory in a file's path prefix
EEXIST	The name file already exist.
EMLINK	The number of links would exceed LINK_MAX
ENAMETOOLONG	Length of a filename string exceeds PATH_MAX and _POSIX_NO_TRUNC is in effect.
ENOENT	A file or directory does not exist.
ENOSPC	No space left on disk.
ENOTDIR	A component of the specified pathname was not a directory when a directory was expected.
EROFS	Read-only file system.

DESCRIPTION:

The mkdir() function creates a new diectory named path. The permission bits (modified by the file creation mask) are set from mode. The owner and group IDs for the directory are set from the effective user ID and group ID.

The new directory may (or may not) contain entries for . and .. but is otherwise empty.

NOTES:

NONE

5.4.17 mkfifo - Makes a FIFO special file

CALLING SEQUENCE:

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(
    const char *path,
    mode_t      mode
);
```

STATUS CODES:

EACCES	Search permission is denied for a directory in a file's path prefix
EEXIST	The named file already exists.
ENOENT	A file or directory does not exist.
ENOSPC	No space left on disk.
ENOTDIR	A component of the specified path was not a directory when a directory was expected.
EROFS	Read-only file system.

DESCRIPTION:

The mkfifo() function creates a new FIFO special file named path. The permission bits (modified by the file creation mask) are set from mode. The owner and group IDs for the FIFO are set from the efective user ID and group ID.

NOTES:

NONE

5.4.18 unlink - Removes a directory entry

CALLING SEQUENCE:

```
#include <unistd.h>
int unlink(
    const char path
);
```

STATUS CODES:

EACCES	Search permission is denied for a directory in a file's path prefix
EBUSY	The directory is in use.
ENAMETOOL	Length of a filename string exceeds PATH_MAX and _POSIX_NO_TRUNC is in effect.
ENOENT	A file or directory does not exist.
ENOTDIR	A component of the specified path was not a directory when a directory was expected.
EPERM	Operation is not permitted. Process does not have the appropriate priviledges or permissions to perform the requested operations.
EROFS	Read-only file system.

DESCRIPTION:

The unlink function removes the link named by path and decrements the link count of the file referenced by the link. When the link count goes to zero and no process has the file open, the space occupied by the file is freed and the file is no longer accessible.

NOTES:

NONE

5.4.19 rmdir - Delete a directory

CALLING SEQUENCE:

```
#include <unistd.h>
int rmdir(
    const char *pathname
);
```

STATUS CODES:

EPERM	The filesystem containing pathname does not support the removal of directories.
EFAULT	pathname points ouside your accessible address space.
EACCES	Write access to the directory containing pathname was not allowed for the process's effective uid, or one of the directories in``pathname`` did not allow search (execute) permission.
EPERM	The directory containing pathname has the stickybit (S_ISVTX) set and the process's effective uid is neither the uid of the file to be delected nor that of the director containing it.
ENAMETC	pathname was too long.
ENOENT	A dirctory component in pathname does not exist or is a dangling symbolic link.
ENOTDIF	pathname, or a component used as a directory in pathname, is not, in fact, a directory.
ENOTEMF	pathname contains entries other than . and .. .
EBUSY	pathname is the current working directory or root directory of some process
EBUSY	pathname is the current directory or root directory of some process.
ENOMEM	Insufficient kernel memory was available
EROGS	pathname refers to a file on a read-only filesystem.
ELOOP	pathname contains a reference to a circular symbolic link

DESCRIPTION:

rmdir deletes a directory, which must be empty

NOTES:

NONE

5.4.20 rename - Renames a file

CALLING SEQUENCE:

```
#include <unistd.h>
int rename(
    const char *old,
    const char *new
);
```

STATUS CODES:

EACCES	Search permission is denied for a directory in a file's path prefix.
EBUSY	The directory is in use.
EEXIST	The named file already exists.
EINVAL	Invalid argument.
EISDIR	Attempt to open a directory for writing or to rename a file to be a directory.
EMLINK	The number of links would exceed LINK_MAX.
ENAMETOOLON(	Length of a filename string exceeds PATH_MAX and _POSIX_NO_TRUNC is in effect.
ENOENT	A file or directory does no exist.
ENOSPC	No space left on disk.
ENOTDIR	A component of the specified pathname was not a directory when a directory was expected.
ENOTEMPTY	Attempt to delete or rename a non-empty directory.
EROFS	Read-only file system
EXDEV	Attempt to link a file to another file system.

DESCRIPTION:

The rename() function causes the file known bo old to now be known as new.

Ordinary files may be renamed to ordinary files, and directories may be renamed to directories; however, files cannot be converted using rename(). The new pathname may not contain a path prefix of old.

NOTES:

If a file already exists by the name new, it is removed. The rename() function is atomic. If the rename() detects an error, no files are removed. This guarantees that the rename("x", "x") does not remove x.

You may not rename dot or dot-dot.

The routine is implemented in Cygnus newlib using link() and unlink().

5.4.21 stat - Gets information about a file

CALLING SEQUENCE:

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(
    const char  *path,
    struct stat *buf
);
```

STATUS CODES:

EACCES	Search permission is denied for a directory in a file's path prefix.
EBADF	Invalid file descriptor.
ENAMETOOLON(	Length of a filename string exceeds PATH_MAX and _POSIX_NO_TRUNC is in effect.
ENOENT	A file or directory does not exist.
ENOTDIR	A component of the specified pathname was not a directory when a directory was expected.

DESCRIPTION:

The path argument points to a pathname for a file. Read, write, or execute permission for the file is not required, but all directories listed in path must be searchable. The stat() function obtains information about the named file and writes it to the area pointed to by buf.

NOTES:

NONE

5.4.22 fstat - Gets file status

CALLING SEQUENCE:

```
#include <sys/types.h>
#include <sys/stat.h>
int fstat(
    int           fildes,
    struct stat *buf
);
```

STATUS CODES:

EBADF	Invalid file descriptor

DESCRIPTION:

The fstat() function obtains information about the file associated with fildes and writes it to the area pointed to by the buf argument.

NOTES:

If the filesystem object referred to by fildes is a link, then the information returned in buf refers to the destination of that link. This is in contrast to lstat() which does not follow the link.

5.4.23 lstat - Gets file status

CALLING SEQUENCE:

```
#include <sys/types.h>
#include <sys/stat.h>
int lstat(
    int           fildes,
    struct stat *buf
);
```

STATUS CODES:

EBADF	Invalid file descriptor

DESCRIPTION:

The `lstat()` function obtains information about the file associated with `fildes` and writes it to the area pointed to by the `buf` argument.

NOTES:

If the filesystem object referred to by `fildes` is a link, then the information returned in `buf` refers to the link itself. This is in contrast to `fstat()` which follows the link.

The `lstat()` routine is defined by BSD 4.3 and SVR4 and not included in POSIX 1003.1b-1996.

5.4.24 access - Check permissions for a file

CALLING SEQUENCE:

```
#include <unistd.h>
int access(
    const char *pathname,
    int         mode
);
```

STATUS CODES:

EACCES	The requested access would be denied, either to the file itself or one of the directories in pathname.
EFAULT	pathname points outside your accessible address space.
EINVAL	Mode was incorrectly specified.
ENAMETOOL(	pathname is too long.
ENOENT	A directory component in pathname would have been accessible but does not exist or was a dangling symbolic link.
ENOTDIR	A component used as a directory in pathname is not, in fact, a directory.
ENOMEM	Insufficient kernel memory was available.

DESCRIPTION:

Access checks whether the process would be allowed to read, write or test for existence of the file (or other file system object) whose name is pathname. If pathname is a symbolic link permissions of the file referred by this symbolic link are tested.

Mode is a mask consisting of one or more of R_OK, W_OK, X_OK and F_OK.

NOTES:

NONE

5.4.25 chmod - Changes file mode.

CALLING SEQUENCE:

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod(
    const char *path,
    mode_t      mode
);
```

STATUS CODES:

EACCES	Search permission is denied for a directory in a file's path prefix
ENAMETOOL	Length of a filename string exceeds PATH_MAX and _POSIX_NO_TRUNC is in effect.
ENOENT	A file or directory does not exist.
ENOTDIR	A component of the specified pathname was not a directory when a directory was expected.
EPERM	Operation is not permitted. Process does not have the appropriate priviledges or permissions to perform the requested operations.
EROFS	Read-only file system.

DESCRIPTION:

Set the file permission bits, the set user ID bit, and the set group ID bit for the file named by path to mode. If the effective user ID does not match the owner of the file and the calling process does not have the appropriate privileges, chmod() returns -1 and sets errno to EPERM.

NOTES:

NONE

5.4.26 fchmod - Changes permissions of a file

CALLING SEQUENCE:

```
#include <sys/types.h>
#include <sys/stat.h>
int fchmod(
    int    fildes,
    mode_t mode
);
```

STATUS CODES:

EACCES	Search permission is denied for a directory in a file's path prefix.
EBADF	The descriptor is not valid.
EFAULT	path points outside your accessible address space.
EIO	A low-level I/o error occurred while modifying the inode.
ELOOP	path contains a circular reference
ENAMETOOLON	Length of a filename string exceeds PATH_MAX and _POSIX_NO_TRUNC is in effect.
ENOENT	A file or directory does no exist.
ENOMEM	Insufficient kernel memory was avaliable.
ENOTDIR	A component of the specified pathname was not a directory when a directory was expected.
EPERM	The effective UID does not match the owner of the file, and is not zero
EROFS	Read-only file system

DESCRIPTION:

The mode of the file given by path or referenced by filedes is changed.

NOTES:

NONE

5.4.27 getdents - Get directory entries

CALLING SEQUENCE:

```
#include <unistd.h>
#include <linux/dirent.h>
#include <linux/unistd.h>
long getdents(
    int   dd_fd,
    char *dd_buf,
    int   dd_len
);
```

STATUS CODES:

A successful call to `getdents` returns th the number of bytes read. On end of directory, 0 is returned. When an error occurs, -1 is returned, and `errno` is set appropriately.

`EBADF`	Invalid file descriptor `fd`.
`EFAULT`	Argument points outside the calling process's address space.
`EINVAL`	Result buffer is too small.
`ENOENT`	No such directory.
`ENOTDIR`	File descriptor does not refer to a directory.

DESCRIPTION:

`getdents` reads several `dirent` structures from the directory pointed by `fd` into the memory area pointed to by `dirp`. The parameter `count` is the size of the memory area.

NOTES:

NONE

5.4.28 chown - Changes the owner and/or group of a file.

CALLING SEQUENCE:

```
#include <sys/types.h>
#include <unistd.h>
int chown(
    const char *path,
    uid_t       owner,
    gid_t       group
);
```

STATUS CODES:

EACCES	Search permission is denied for a directory in a file's path prefix
EINVAL	Invalid argument
ENAMETOOL	Length of a filename string exceeds PATH_MAX and _POSIX_NO_TRUNC is in effect.
ENOENT	A file or directory does not exist.
ENOTDIR	A component of the specified pathname was not a directory when a directory was expected.
EPERM	Operation is not permitted. Process does not have the appropriate priviledges or permissions to perform the requested operations.
EROFS	Read-only file system.

DESCRIPTION:

The user ID and group ID of the file named by path are set to owner and path, respectively.

For regular files, the set group ID (S_ISGID) and set user ID (S_ISUID) bits are cleared.

Some systems consider it a security violation to allow the owner of a file to be changed, If users are billed for disk space usage, loaning a file to another user could result in incorrect billing. The chown() function may be restricted to privileged users for some or all files. The group ID can still be changed to one of the supplementary group IDs.

NOTES:

This function may be restricted for some file. The pathconf function can be used to test the _PC_CHOWN_RESTRICTED flag.

5.4.29 utime - Change access and/or modification times of an inode

CALLING SEQUENCE:

```
#include <sys/types.h>
int utime(
    const char     *filename,
    struct utimbuf *buf
);
```

STATUS CODES:

EACCES	Permission to write the file is denied
ENOENT	Filename does not exist

DESCRIPTION:

Utime changes the access and modification times of the inode specified by filename to the actime and modtime fields of buf respectively. If buf is NULL, then the access and modification times of the file are set to the current time.

NOTES:

NONE

5.4.30 ftruncate - truncate a file to a specified length

CALLING SEQUENCE:

```
#include <unistd.h>
int ftrunctate(
    int    fd,
    size_t length
);
```

STATUS CODES:

ENOTDIR	A component of the path prefix is not a directory.
EINVAL	The pathname contains a character with the high-order bit set.
ENAMETOOL	The length of the specified pathname exceeds PATH_MAX bytes, or the length of a component of the pathname exceeds NAME_MAX bytes.
ENOENT	The named file does not exist.
EACCES	The named file is not writable by the user.
EACCES	Search permission is denied for a component of the path prefix.
ELOOP	Too many symbolic links were encountered in translating the pathname
EISDIR	The named file is a directory.
EROFS	The named file resides on a read-only file system
ETXTBSY	The file is a pure procedure (shared text) file that is being executed
EIO	An I/O error occurred updating the inode.
EFAULT	Path points outside the process's allocated address space.
EBADF	The fd is not a valid descriptor.

DESCRIPTION:

truncate() causes the file named by path or referenced by fd to be truncated to at most length bytes in size. If the file previously was larger than this size, the extra data is lost. With ftruncate(), the file must be open for writing.

NOTES:

NONE

5.4.31 truncate - truncate a file to a specified length

CALLING SEQUENCE:

```
#include <unistd.h>
int truncate(
    const char *path,
    size_t      length
);
```

STATUS CODES:

ENOTDIR	A component of the path prefix is not a directory.
EINVAL	The pathname contains a character with the high-order bit set.
ENAMETOOL	The length of the specified pathname exceeds PATH_MAX bytes, or the length of a component of the pathname exceeds NAME_MAX bytes.
ENOENT	The named file does not exist.
EACCES	The named file is not writable by the user.
EACCES	Search permission is denied for a component of the path prefix.
ELOOP	Too many symbolic links were encountered in translating the pathname
EISDIR	The named file is a directory.
EROFS	The named file resides on a read-only file system
ETXTBSY	The file is a pure procedure (shared text) file that is being executed
EIO	An I/O error occurred updating the inode.
EFAULT	Path points outside the process's allocated address space.
EBADF	The fd is not a valid descriptor.

DESCRIPTION:

truncate() causes the file named by path or referenced by``fd`` to be truncated to at most length bytes in size. If the file previously was larger than this size, the extra data is lost. With ftruncate(), the file must be open for writing.

NOTES:

NONE

5.4.32 pathconf - Gets configuration values for files

CALLING SEQUENCE:

```
#include <unistd.h>
int pathconf(
    const char *path,
    int         name
);
```

STATUS CODES:

EINVAL	Invalid argument
EACCES	Permission to write the file is denied
ENAMETOOLONG	Length of a filename string exceeds PATH_MAX and _POSIX_NO_TRUNC is in effect.
ENOENT	A file or directory does not exist
ENOTDIR	A component of the specified path was not a directory whan a directory was expected.

DESCRIPTION:

pathconf() gets a value for the configuration option name for the open file descriptor filedes.

The possible values for name are:

_PC_LINK_MAX	Returns the maximum number of links to the file. If filedes or``path`` refer to a directory, then the value applies to the whole directory. The corresponding macro is _POSIX_LINK_MAX.
_PC_MAX_CANON	Returns the maximum length of a formatted input line, where filedes or path must refer to a terminal. The corresponding macro is _POSIX_MAX_CANON.
_PC_MAX_INPUT	Returns the maximum length of an input line, where filedes or path must refer to a terminal. The corresponding macro is``_POSIX_MAX_INPUT``.
_PC_NAME_MAX	Returns the maximum length of a filename in the directory path or filedes. The process is allowed to create. The corresponding macro is _POSIX_NAME_MAX.
_PC_PATH_MAX	returns the maximum length of a relative pathname when path or``filedes`` is the current working directory. The corresponding macro is _POSIX_PATH_MAX.
_PC_PIPE_BUF	returns the size of the pipe buffer, where filedes must refer to a pipe or FIFO and path must refer to a FIFO. The corresponding macro is _POSIX_PIPE_BUF.
_PC_CHOWN_RESTRICTE	Returns nonzero if the chown(2) call may not be used on this file. If``filedes`` or path refer to a directory, then this applies to all files in that directory. The corresponding macro is _POSIX_CHOWN_RESTRICTED.

NOTES:

Files with name lengths longer than the value returned for name equal _PC_NAME_MAX may exist in the given directory.

5.4.33 fpathconf - Gets configuration values for files

CALLING SEQUENCE:

```
#include <unistd.h>
int fpathconf(
    int filedes,
    int name
);
```

STATUS CODES:

EINVAL	Invalid argument
EACCES	Permission to write the file is denied
ENAMETOOLONG	Length of a filename string exceeds PATH_MAX and _POSIX_NO_TRUNC is in effect.
ENOENT	A file or directory does not exist
ENOTDIR	A component of the specified path was not a directory whan a directory was expected.

DESCRIPTION:

pathconf() gets a value for the configuration option name for the open file descriptor filedes.

The possible values for name are:

_PC_ LINK_ MAX	Returns the maximum number of links to the file. If filedes or path refer to a directory, then the value applies to the whole directory. The corresponding macro is _POSIX_LINK_MAX.
_PC_ MAX_ CANON	returns the maximum length of a formatted input line, where filedes or path must refer to a terminal. The corresponding macro is _POSIX_MAX_CANON.
_PC_ MAX_ INPUT	Returns the maximum length of an input line, where filedes or path must refer to a terminal. The corresponding macro is _POSIX_MAX_INPUT.
_PC_ NAME_ MAX	Returns the maximum length of a filename in the directory path or filedes. The process is allowed to create. The corresponding macro is _POSIX_NAME_MAX.
_PC_ PATH_ MAX	Returns the maximum length of a relative pathname when path or filedes is the current working directory. The corresponding macro is _POSIX_PATH_MAX.
_PC_ PIPE_ BUF	Returns the size of the pipe buffer, where filedes must refer to a pipe or FIFO and path must refer to a FIFO. The corresponding macro is _POSIX_PIPE_BUF.
_PC_ CHOWN_ RESTRICTE	Returns nonzero if the chown() call may not be used on this file. If filedes or path refer to a directory, then this applies to all files in that directory. The corresponding macro is _POSIX_CHOWN_RESTRICTED.

NOTES:

NONE

5.4.34 mknod - create a directory

CALLING SEQUENCE:

```
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
long mknod(
    const char *pathname,
    mode_t      mode,
    dev_t       dev
);
```

STATUS CODES:

mknod returns zero on success, or -1 if an error occurred (in which case, errno is set appropriately).

ENAMETOOL	pathname was too long.
ENOENT	A directory component in pathname does not exist or is a dangling symbolic link.
ENOTDIR	A component used in the directory pathname is not, in fact, a directory.
ENOMEM	Insufficient kernel memory was available
EROFS	pathname refers to a file on a read-only filesystem.
ELOOP	pathname contains a reference to a circular symbolic link, ie a symbolic link whose expansion contains a reference to itself.
ENOSPC	The device containing pathname has no room for the new node.

DESCRIPTION:

mknod attempts to create a filesystem node (file, device special file or named pipe) named pathname, specified by mode and dev.

mode specifies both the permissions to use and the type of node to be created.

It should be a combination (using bitwise OR) of one of the file types listed below and the permissions for the new node.

The permissions are modified by the process's umask in the usual way: the permissions of the created node are (mode & ~umask).

The file type should be one of S_IFREG, S_IFCHR, S_IFBLK and S_IFIFO to specify a normal file (which will be created empty), character special file, block special file or FIFO (named pipe), respectively, or zero, which will create a normal file.

If the file type is S_IFCHR or S_IFBLK then dev specifies the major and minor numbers of the newly created device special file; otherwise it is ignored.

The newly created node will be owned by the effective uid of the process. If the directory containing the node has the set group id bit set, or if the filesystem is mounted with BSD group semantics, the new node will inherit the group ownership from its parent directory; otherwise it will be owned by the effective gid of the process.

NOTES:

NONE

CHAPTER

SIX

INPUT AND OUTPUT PRIMITIVES MANAGER

6.1 Introduction

The input and output primitives manager is . . .

The directives provided by the input and output primitives manager are:

- *pipe* (page 77) - Create an Inter-Process Channel
- *dup* (page 77) - Duplicates an open file descriptor
- *dup2* (page 78) - Duplicates an open file descriptor
- *close* (page 78) - Closes a file
- *read* (page 79) - Reads from a file
- *write* (page 80) - Writes to a file
- *fcntl* (page 80) - Manipulates an open file descriptor
- *lseek* (page 82) - Reposition read/write file offset
- *fsync* (page 82) - Synchronize file complete in-core state with that on disk
- *fdatasync* (page 83) - Synchronize file in-core data with that on disk
- *sync* (page 84) - Schedule file system updates
- *mount* (page 84) - Mount a file system
- *unmount* (page 85) - Unmount file systems
- *readv* (page 85) - Vectored read from a file
- *writev* (page 86) - Vectored write to a file
- *aio_read* (page 86) - Asynchronous Read
- *aio_write* (page 87) - Asynchronous Write
- *lio_listio* (page 88) - List Directed I/O
- *aio_error* (page 89) - Retrieve Error Status of Asynchronous I/O Operation
- **aio_return_** - Retrieve Return Status Asynchronous I/O Operation
- *aio_cancel* (page 90) - Cancel Asynchronous I/O Request
- *aio_suspend* (page 90) - Wait for Asynchronous I/O Request
- *aio_fsync* (page 91) - Asynchronous File Synchronization

6.2 Background

There is currently no text in this section.

6.3 Operations

There is currently no text in this section.

6.4 Directives

This section details the input and output primitives manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

6.4.1 pipe - Create an Inter-Process Channel

CALLING SEQUENCE:

```
#include <unistd.h>
int pipe(
    int fildes[2]
);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

This routine is not currently supported by RTEMS but could be in a future version.

6.4.2 dup - Duplicates an open file descriptor

CALLING SEQUENCE:

```
#include <unistd.h>
int dup(
    int fildes
);
```

STATUS CODES:

EBADF	Invalid file descriptor.
EINTR	Function was interrupted by a signal.
EMFILE	The process already has the maximum number of file descriptors open and tried to open a new one.

DESCRIPTION:

The dup function returns the lowest numbered available file descriptor. This new desciptor refers to the same open file as the original descriptor and shares any locks.

NOTES:

NONE

6.4.3 dup2 - Duplicates an open file descriptor

CALLING SEQUENCE:

```
#include <unistd.h>
int dup2(
    int fildes,
    int fildes2
);
```

STATUS CODES:

EBADF	Invalid file descriptor.
EINTR	Function was interrupted by a signal.
EMFILE	The process already has the maximum number of file descriptors open and tried to open a new one.

DESCRIPTION:

dup2 creates a copy of the file descriptor oldfd.

The old and new descriptors may be used interchangeably. They share locks, file position pointers and flags; for example, if the file position is modified by using lseek on one of the descriptors, the position is also changed for the other.

NOTES:

NONE

6.4.4 close - Closes a file

CALLING SEQUENCE:

```
#include <unistd.h>
int close(
    int fildes
);
```

STATUS CODES:

EBADF	Invalid file descriptor
EINTR	Function was interrupted by a signal.

DESCRIPTION:

The close() function deallocates the file descriptor named by fildes and makes it available for reuse. All outstanding record locks owned by this process for the file are unlocked.

NOTES:

A signal can interrupt the close() function. In that case, close() returns -1 with errno set to EINTR. The file may or may not be closed.

6.4.5 read - Reads from a file

CALLING SEQUENCE:

```
#include <unistd.h>
ssize_t read(
    int fildes,
    void *buf,
    size_t nbyte
);
```

STATUS CODES:

On error, this routine returns -1 and sets `errno` to one of the following:

`EAGAIN`	The O_NONBLOCK flag is set for a file descriptor and the process would be delayed in the I/O operation.
`EBADF`	Invalid file descriptor
`EINTR`	Function was interrupted by a signal.
`EIO`	Input or output error
`EINVAL`	Bad buffer pointer

DESCRIPTION:

The `read()` function reads `nbyte` bytes from the file associated with `fildes` into the buffer pointed to by `buf`.

The `read()` function returns the number of bytes actually read and placed in the buffer. This will be less than `nbyte` if:

- The number of bytes left in the file is less than `nbyte`.
- The `read()` request was interrupted by a signal.
- The file is a pipe or FIFO or special file with less than `nbytes` immediately available for reading.

When attempting to read from any empty pipe or FIFO:

- If no process has the pipe open for writing, zero is returned to indicate end-of-file.
- If some process has the pipe open for writing and O_NONBLOCK is set, -1 is returned and `errno` is set to EAGAIN.
- If some process has the pipe open for writing and O_NONBLOCK is clear, `read()` waits for some data to be written or the pipe to be closed.

When attempting to read from a file other than a pipe or FIFO and no data is available.

- If O_NONBLOCK is set, -1 is returned and `errno` is set to EAGAIN.
- If O_NONBLOCK is clear, `read()` waits for some data to become available.
- The O_NONBLOCK flag is ignored if data is available.

NOTES:

NONE

6.4.6 write - Writes to a file

CALLING SEQUENCE:

```
#include <unistd.h>
ssize_t write(
    int fildes,
    const void *buf,
    size_t nbyte
);
```

STATUS CODES:

EAGAIN	The O_NONBLOCK flag is set for a file descriptor and the process would be delayed in the I/O operation.
EBADF	Invalid file descriptor
EFBIG	An attempt was made to write to a file that exceeds the maximum file size
EINTR	The function was interrupted by a signal.
EIO	Input or output error.
ENOSPC	No space left on disk.
EPIPE	Attempt to write to a pope or FIFO with no reader.
EINVAL	Bad buffer pointer

DESCRIPTION:

The `write()` function writes `nbyte` from the array pointed to by `buf` into the file associated with `fildes`.

If `nybte` is zero and the file is a regular file, the `write()` function returns zero and has no other effect. If `nbyte` is zero and the file is a special file, te results are not portable.

The `write()` function returns the number of bytes written. This number will be less than `nbytes` if there is an error. It will never be greater than `nbytes`.

NOTES:

NONE

6.4.7 fcntl - Manipulates an open file descriptor

CALLING SEQUENCE:

```
#include <fcntl.h>
int fcntl(
    int fildes,
    int cmd,
    ...
);
```

STATUS CODES:

EACCES	Search permission is denied for a direcotry in a file's path prefix.
EAGAIN	The O_NONBLOCK flag is set for a file descriptor and the process would be delayed in the I/O operation.
EBADF	Invalid file descriptor
EDEADLK	An `fcntl` with function `F_SETLKW` would cause a deadlock.
EINTR	The functioin was interrupted by a signal.
EINVAL	Invalid argument
EMFILE	Too many file descriptor or in use by the process.
ENOLCK	No locks available

DESCRIPTION:

`fcntl()` performs one of various miscellaneous operations on``fd``. The operation in question is determined by `cmd`:

F_DUPFD	Makes `arg` be a copy of `fd`, closing `fd` first if necessary. The same functionality can be more easily achieved by using `dup2()`. The old and new descriptors may be used interchangeably. They share locks, file position pointers and flags; for example, if the file position is modified by using `lseek()` on one of the descriptors, the position is also changed for the other. The two descriptors do not share the close-on-exec flag, however. The close-on-exec flag of the copy is off, meaning that it will be closed on exec. On success, the new descriptor is returned.
F_GETFD	Read the close-on-exec flag. If the low-order bit is 0, the file will remain open across exec, otherwise it will be closed.
F_SETFD	Set the close-on-exec flag to the value specified by `arg` (only the least significant bit is used).
F_GETFL	Read the descriptor's flags (all flags (as set by open()) are returned).
F_SETFL	Set the descriptor's flags to the value specified by `arg`. Only``O_APPEND`` and `O_NONBLOCK` may be set. The flags are shared between copies (made with `dup()` etc.) of the same file descriptor. The flags and their semantics are described in `open()`.
F_GETLK, F_SETLK and F_SETLKW	Manage discretionary file locks. The third argument `arg` is a pointer to a struct flock (that may be overwritten by this call).
F_GETLK	Return the flock structure that prevents us from obtaining the lock, or set the``l_type`` field of the lock to `F_UNLCK` if there is no obstruction.
F_SETLK	The lock is set (when `l_type` is `F_RDLCK` or `F_WRLCK`) or cleared (when it is `F_UNLCK`. If lock is held by someone else, this call returns -1 and sets `errno` to EACCES or EAGAIN.
F_SETLKW	Like `F_SETLK`, but instead of returning an error we wait for the lock to be released.
F_GETOWN	Get the process ID (or process group) of the owner of a socket. Process groups are returned as negative values.
F_SETOWN	Set the process or process group that owns a socket. For these commands, ownership means receiving `SIGIO` or `SIGURG` signals. Process groups are specified using negative values.

NOTES:

The errors returned by dup2 are different from those returned by F_DUPFD.

6.4.8 lseek - Reposition read/write file offset

CALLING SEQUENCE:

```
#include <unistd.h>
off_t lseek(
    int fildes,
    off_t offset,
    int whence
);
```

STATUS CODES:

EBADF	fildes is not an open file descriptor.
ESPIPE	fildes is associated with a pipe, socket or FIFO.
EINVAL	whence is not a proper value.

DESCRIPTION:

The lseek function repositions the offset of the file descriptor fildes to the argument offset according to the directive whence. The argument fildes must be an open file descriptor. Lseek repositions the file pointer fildes as follows:

- If whence is SEEK_SET, the offset is set to offset bytes.
- If whence is SEEK_CUR, the offset is set to its current location plus offset bytes.
- If whence is SEEK_END, the offset is set to the size of the file plus offset bytes.

The lseek function allows the file offset to be set beyond the end of the existing end-of-file of the file. If data is later written at this point, subsequent reads of the data in the gap return bytes of zeros (until data is actually written into the gap).

Some devices are incapable of seeking. The value of the pointer associated with such a device is undefined.

NOTES:

NONE

6.4.9 fsync - Synchronize file complete in-core state with that on disk

CALLING SEQUENCE:

```
#include <unistd.h>
int fsync(
    int fildes
);
```

STATUS CODES:

On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

EBADF	fd is not a valid descriptor open for writing
EINVAL	**fd is bound to a special file which does not support support** synchronization
EROFS	**fd is bound to a special file which does not support support** synchronization
EIO	An error occurred during synchronization

DESCRIPTION:

fsync copies all in-core parts of a file to disk.

NOTES:

NONE

6.4.10 fdatasync - Synchronize file in-core data with that on disk

CALLING SEQUENCE:

```
#include <unistd.h>
int fdatasync(
    int fildes
);
```

STATUS CODES:

On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

EBADF	fd is not a valid file descriptor open for writing.
EINVAL	fd is bound to a special file which does not support synchronization.
EIO	An error occurred during synchronization.
EROFS	fd is bound to a special file which dows not support synchronization.

DESCRIPTION:

fdatasync flushes all data buffers of a file to disk (before the system call returns). It resembles fsync but is not required to update the metadata such as access time.

Applications that access databases or log files often write a tiny data fragment (e.g., one line in a log file) and then call fsync immediately in order to ensure that the written data is physically stored on the harddisk. Unfortunately, fsync will always initiate two write operations: one for the newly written data and another one in order to update the modification time stored in the inode. If the modification time is not a part of the transaction concept fdatasync can be used to avoid unnecessary inode disk write operations.

NOTES:

NONE

6.4.11 sync - Schedule file system updates

CALLING SEQUENCE:

```
#include <unistd.h>
void sync(
    void
);
```

STATUS CODES:

NONE

DESCRIPTION:

The sync service causes all information in memory that updates file systems to be scheduled for writing out to all file systems.

NOTES:

The writing of data to the file systems is only guaranteed to be scheduled upon return. It is not necessarily complete upon return from sync.

6.4.12 mount - Mount a file system

CALLING SEQUENCE:

```
#include <libio.h>
int mount(
    rtems_filesystem_mount_table_entry_t **mt_entry,
    rtems_filesystem_operations_table *fs_ops,
    rtems_filesystem_options_t fsoptions,
    char *device,
    char *mount_point
);
```

STATUS CODES:

- – ENOMEM
  - – Unable to allocate memory needed.
- – EINVAL
  - – The filesystem does not support being mounted.
- – EINVAL
  - – Attempt to mount a read-only filesystem as writeable.

DESCRIPTION:

The mount routines mounts the filesystem class which uses the filesystem operations specified by fs_ops and fsoptions. The filesystem is mounted at the directory mount_point and the mode of the mounted filesystem is specified by fsoptions. If this filesystem class requires a device, then the name of the device must be specified by device.

If this operation succeeds, the mount table entry for the mounted filesystem is returned in mt_entry.

NOTES:

This method is not defined in the POSIX standard.

6.4.13 unmount - Unmount file systems

CALLING SEQUENCE:

```
#include <libio.h>
int unmount(
    const char *mount_path
);
```

STATUS CODES:

- – EBUSY
  - – Filesystem is in use or the root filesystem.
- – EACCESS
  - – Unable to allocate memory needed.

DESCRIPTION:

The unmount routine removes the attachment of the filesystem specified by mount_path.

NOTES:

This method is not defined in the POSIX standard.

6.4.14 readv - Vectored read from a file

CALLING SEQUENCE:

```
#include <sys/uio.h>
ssize_t readv(
    int fildes,
    const struct iovec *iov,
    int iovcnt
);
```

STATUS CODES:

In addition to the errors detected by *Input and Output Primitives Manager read - Reads from a file, read()*, this routine may return -1 and sets errno based upon the following errors:

EINVAL	The sum of the iov_len values in the iov array overflowed an ssize_t.
EINVAL	The iovcnt argument was less than or equal to 0, or greater than IOV_MAX.

DESCRIPTION:

The readv() function is equivalent to read() except as described here. The readv() function shall place the input data into the iovcnt buffers specified by the members of the iov array: iov[0], iov[1], ..., iov[iovcnt-1].

Each iovec entry specifies the base address and length of an area in memory where data should be placed. The readv() function always fills an area completely before proceeding to the next.

NOTES:

NONE

6.4.15 writev - Vectored write to a file

CALLING SEQUENCE:

```
#include <sys/uio.h>
ssize_t writev(
    int fildes,
    const struct iovec *iov,
    int iovcnt
);
```

STATUS CODES:

In addition to the errors detected by *Input and Output Primitives Manager write - Write to a file, write()*, this routine may return -1 and sets `errno` based upon the following errors:

`EINVAL`	The sum of the `iov_len` values in the iov array overflowed an `ssize_t`.
`EINVAL`	The `iovcnt` argument was less than or equal to 0, or greater than `IOV_MAX`.

DESCRIPTION:

The `writev()` function is equivalent to `write()`, except as noted here. The `writev()` function gathers output data from the `iovcnt` buffers specified by the members of the `iov array`: `iov[0]`, `iov[1]`, `...`, `iov[iovcnt-1]`. The `iovcnt` argument is valid if greater than 0 and less than or equal to `IOV_MAX`.

Each `iovec` entry specifies the base address and length of an area in memory from which data should be written. The `writev()` function always writes a complete area before proceeding to the next.

If `fd` refers to a regular file and all of the `iov_len` members in the array pointed to by `iov` are 0, `writev()` returns 0 and has no other effect. For other file types, the behavior is unspecified by POSIX.

NOTES:

NONE

6.4.16 aio_read - Asynchronous Read

CALLING SEQUENCE:

```
#include <aio.h>
int aio_read(
    struct aiocb *aiocbp
);
```

STATUS CODES:

If the request is successfully enqueued, zero is returned. On error, this routine returns -1 and sets `errno` to one of the following:

EBADF	The file descriptor is not open for reading.
EINVAL	Invalid aio_reqprio, aio_offset, or aio_nbytes.
EAGAIN	Not enough memory to queue the request.
EAGAIN	the addition of a new request to the queue would violate the RTEMS_AIO_MAX limit.
EINVAL	The starting position is past the maximum offset for the file.
EINVAL	aiocbp->aio_sigevent does not point to a valid sigevent struct.
EINVAL	aiocbp is a NULL pointer.

DESCRIPTION:

The aio_read() function is the asynchronous equivalent of read(). This function returns immediately, the request is serviced by thread(s) running in the background.

The parameters for the read are specified in the aiocbp structure.

NOTES:

NONE

6.4.17 aio_write - Asynchronous Write

CALLING SEQUENCE:

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

STATUS CODES:

If the request is successfully enqueued, zero is returned. On error, this routine returns -1 and sets errno to one of the following:

EBADF	The file descriptor is not open for writing.
EINVAL	Invalid aio_reqprio, aio_offset, or aio_nbytes.
EAGAIN	Not enough memory to queue the request.
EAGAIN	the addition of a new request to the queue would violate the RTEMS_AIO_MAX limit.
EINVAL	aiocbp->aio_sigevent does not point to a valid sigevent struct.
EINVAL	aiocbp is a NULL pointer.

DESCRIPTION:

The aio_write() function is the asynchronous equivalent of write(). This function returns immediately, the request is serviced by thread(s) running in the background.

The parameters for the write are specified in the aiocbp structure.

NOTES:

NONE

6.4.18 lio_listio - List Directed I/O

CALLING SEQUENCE:

```
#include <aio.h>
int lio_listio(
    int mode,
    struct aiocb *restrict const list[restrict],
    int nent,
    struct sigevent *restrict sig
);
```

STATUS CODES:

If the call to `lio_listio` is successful, zero is returned. On error, this routine returns -1 and sets `errno` to one of the following:

`ENOSYS`	The project has been build with RTEMS_POSIX_API not defined.
`EAGAIN`	The call failed due to resources limitations.
`EAGAIN`	The number of entries indicated by nent value would cause the `RTEMS_AIO_MAX` limit to be excedeed.
`EINVAL`	list is a `NULL` pointer.
`EINVAL`	`mode` is not a valid value.
`EINVAL`	the value of `nent` is not valid or higher than `AIO_LISTIO_MAX`.
`EINVAL`	the `sigevent` struct pointed by `sig` is not valid.
`EINTR`	The wait for list completion during a `LIO_WAIT` operation was interrupted by an external event.
`EIO`	One or more of the individual I/O operations failed.

DESCRIPTION:

The `lio_listio()` function allows for the simultaneous initiation of multiple asynchronous I/O operations.

Each operation is described by an `aiocb` structure in the array `list`.

The `mode` parameter determines when the function will return. If `mode` is `LIO_WAIT` the function returns when the I/O operation have completed, if `mode` is `LIO_NOWAIT` the function returns after enqueueing the operations.

If `mode` is `LIO_NOWAIT`, the sigevent struct pointed by `sig` is used to notify list completion.

NOTES:

When the `mode` is `LIO_NOWAIT` and the `sigev_notify` field of `sig` is set to `SIGEV_SIGNAL`, a signal is sent to the process to notify the completion of the list. Since each RTEMS application is logically a single POSIX process, if the user wants to wait for the signal (using, for example, `sigwait()`), it is necessary to ensure that the signal is blocked by every thread.

This function is only available when `RTEMS_POSIX_API` is defined. To do so, it's necessary to add `RTEMS_POSIX_API = True` to the `config.ini` file.

6.4.19 aio_error - Retrieve Error Status of Asynchronous I/O Operation

CALLING SEQUENCE:

```
#include <aio.h>
int aio_error(
    const struct aiocb *aiocbp
);
```

STATUS CODES:

The function return the error status of the request, if 0 is returned the operation completed without errors.

If the request is still in progress, the function returns `EINPROGRESS`.

On error, this routine returns -1 and sets `errno` to one of the following:

`EINVAL`	The return status for the request has already been retrieved.
`EINVAL`	`aiocbp` is a NULL pointer.

DESCRIPTION:

The `aio_error()` function retrieves the error status of the request.

`aiocbp` is a pointer to the request control block.

NOTES:

NONE .. _aio_return:

6.4.20 aio_return - Retrieve Return Status of Asynchronous I/O Operation

CALLING SEQUENCE:

```
#include <aio.h>
ssize_t aio_return(
    struct aiocb *aiocbp
);
```

STATUS CODES:

If the result can be returned, it is returned as defined by the various operations.

On error, this routine returns -1 and sets `errno` to one of the following:

`EINVAL`	The return status for the request has already been retrieved.
`EINVAL`	`aiocbp` is a NULL pointer.

DESCRIPTION:

The `aio_return()` function retrieves the return status of the asynchronous I/O operation. `aiocbp` is a pointer to the request control block.

NOTES:

NONE

6.4.21 aio_cancel - Cancel Asynchronous I/O Request

CALLING SEQUENCE:

```
#include <aio.h>
int aio_cancel(
    int fildes,
    struct aiocb *aiocbp
);
```

STATUS CODES:

If the function terminated without errors, the return value has one of the following values:

AIO_CANCELED	The requested operation(s) were canceled.
AIO_NOTCANCELED	Some operations could not be canceled because they are in progress.
AIO_ALLDONE	None of the operations could be canceled because they are already complete.

If the file descriptor is invalid, -1 is returned and errno is set to EBADF

DESCRIPTION:

The aio_cancel() function attempts to cancel asynchronous I/O operations.

filedes is the file descriptor associated with the operations to be canceled. aiocbp is a pointer to an asynchronous I/O control block.

If aiocbp is NULL, the function will attempt to eliminate all the operations enqueued for the specified filedes.

If aiocbp points to a control block, then only the referenced operation shall be eliminated. The aio_filedef value of aiocbp must be equal to filedes, otherwise the function will return with an error.

NOTES:

NONE

6.4.22 aio_suspend - Wait for Asynchronous I/O Request

CALLING SEQUENCE:

```
#include <aio.h>
int aio_suspend(
    const struct aiocb *const list[],
    int nent,
    const struct timespec *timeout
);
```

STATUS CODES:

On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

E	The

DESCRIPTION:

The `aio_suspend()` function suspends the calling process until one or more operations have completed or until the specified `timeout` has expired. `list` contains the requests that must complete.

NOTES:

This routine is not currently supported by RTEMS.

6.4.23 aio_fsync - Asynchronous File Synchronization

CALLING SEQUENCE:

```
#include <aio.h>
int aio_fsync(
    int op,
    struct aiocb *aiocbp
);
```

STATUS CODES:

If the requests are succesfully enqueued, zero is returned. On error, this routine returns -1 and sets `errno` to one of the following:

`EAGAIN`	The operation could not be queued due to temporary resource limitations.
`EAGAIN`	the addition of a new request to the queue would violate the `RTEMS_AIO_MAX` limit.
`EBADF`	The aio_fildes member of aiocbp is not a valid file descriptor.
`EINVAL`	A value of op other than O_SYNC or O_DSYNC was specified.
`EINVAL`	`aiocbp->aio_sigevent` does not point to a valid sigevent struct.
`EINVAL`	`aiocbp` is a NULL pointer.

DESCRIPTION:

The `aio_fsync()` function initiates an asynchronous file sync operation. `op` specifies what kind of synchronization should be performed. If `op` is `O_SYNC`, all currently queued I/O operations shall be synchronized as if by a call to `fsync()`. If `op` is `O_DSYNC`, all currently queued I/O operations shall be synchronized as if by a call to `fdatasync()`.

NOTES:

Currently, `O_DSYNC` and `O_SYNC` are mapped to the same value. As a result, every file will be synced as if by a call to `fsync()`.

CHAPTER

SEVEN

DEVICE- AND CLASS- SPECIFIC FUNCTIONS MANAGER

7.1 Introduction

The device- and class- specific functions manager is . . .

The directives provided by the device- and class- specific functions manager are:

- *cfgetispeed* (page 97) - Reads terminal input baud rate
- *cfgetospeed* (page 97) - Reads terminal output baud rate
- *cfsetispeed* (page 98) - Sets terminal input baud rate
- *cfsetospeed* (page 98) - Set terminal output baud rate
- *tcgetattr* (page 99) - Gets terminal attributes
- *tcsetattr* (page 99) - Set terminal attributes
- *tcsendbreak* (page 99) - Sends a break to a terminal
- *tcdrain* (page 100) - Waits for all output to be transmitted to the terminal
- *tcflush* (page 100) - Discards terminal data
- *tcflow* (page 101) - Suspends/restarts terminal output
- *tcgetpgrp* (page 101) - Gets foreground process group ID
- *tcsetpgrp* (page 101) - Sets foreground process group ID

7.2 Background

There is currently no text in this section.

7.3 Operations

There is currently no text in this section.

7.4 Directives

This section details the device- and class- specific functions manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

7.4.1 cfgetispeed - Reads terminal input baud rate

CALLING SEQUENCE:

```
#include <termios.h>
speed_t cfgetispeed(
    const struct termios *termios_p
);
```

STATUS CODES:

The `cfgetispeed()` function returns a code for baud rate.

DESCRIPTION:

The `cfsetispeed()` function stores a code for the terminal speed stored in a struct termios. The codes are defined in `<termios.h>` by the macros `B0`, `B50`, `B75`, `B110`, `B134`, `B150`, `B200`, `B300`, `B600`, `B1200`, `B1800`, `B2400`, `B4800`, `B9600`, `B19200`, and `B38400`.

The `cfsetispeed()` function does not do anything to the hardware. It merely stores a value for use by `tcsetattr()`.

NOTES:

Baud rates are defined by symbols, such as `B110`, `B1200`, `B2400`. The actual number returned for any given speed may change from system to system.

7.4.2 cfgetospeed - Reads terminal output baud rate

CALLING SEQUENCE:

```
#include <termios.h>
speed_t cfgetospeed(
    const struct termios *termios_p
);
```

STATUS CODES:

The `cfgetospeed()` function returns the termios code for the baud rate.

DESCRIPTION:

The `cfgetospeed()` function returns a code for the terminal speed stored in a `struct termios`. The codes are defined in `<termios.h>` by the macros `B0`, `B50`, `B75`, `B110`, `B134`, `B150`, `B200`, `B300`, `B600`, `B1200`, `B1800`, `B2400`, `B4800`, `B9600`, `B19200`, and `B38400`.

The `cfgetospeed()` function does not do anything to the hardware. It merely returns the value stored by a previous call to `tcgetattr()`.

NOTES:

Baud rates are defined by symbols, such as `B110`, `B1200`, `B2400`. The actual number returned for any given speed may change from system to system.

7.4.3 cfsetispeed - Sets terminal input baud rate

CALLING SEQUENCE:

```
#include <termios.h>
int cfsetispeed(
    struct termios *termios_p,
    speed_t speed
);
```

STATUS CODES:

The `cfsetispeed()` function returns a zero when successful and returns -1 when an error occurs.

DESCRIPTION:

The `cfsetispeed()` function stores a code for the terminal speed stored in a struct termios. The codes are defined in `<termios.h>` by the macros `B0`, `B50`, `B75`, `B110`, `B134`, `B150`, `B200`, `B300`, `B600`, `B1200`, `B1800`, `B2400`, `B4800`, `B9600`, `B19200`, and `B38400`.

NOTES:

This function merely stores a value in the `termios` structure. It does not change the terminal speed until a `tcsetattr()` is done. It does not detect impossible terminal speeds.

7.4.4 cfsetospeed - Sets terminal output baud rate

CALLING SEQUENCE:

```
#include <termios.h>
int cfsetospeed(
    struct termios *termios_p,
    speed_t speed
);
```

STATUS CODES:

The `cfsetospeed()` function returns a zero when successful and returns -1 when an error occurs.

DESCRIPTION:

The `cfsetospeed()` function stores a code for the terminal speed stored in a struct `termios`. The codes are defiined in `<termios.h>` by the macros `B0`, `B50`, `B75`, `B110`, `B134`, `B150`, `B200`, `B300`, `B600`, `B1200`, `B1800`, `B2400`, `B4800`, `B9600`, `B19200`, and `B38400`.

The `cfsetospeed()` function does not do anything to the hardware. It merely stores a value for use by `tcsetattr()`.

NOTES:

This function merely stores a value in the `termios` structure. It does not change the terminal speed until a `tcsetattr()` is done. It does not detect impossible terminal speeds.

7.4.5 tcgetattr - Gets terminal attributes

CALLING SEQUENCE:

```
#include <termios.h>
int tcgetattr(
    int fildes,
    struct termios *termios_p
);
```

STATUS CODES:

EBADF	Invalid file descriptor
ENOOTY	Terminal control function attempted for a file that is not a terminal.

DESCRIPTION:

The `tcgetattr()` gets the parameters associated with the terminal referred to by `fildes` and stores them into the `termios()` structure pointed to by `termios_p`.

NOTES:

NONE

7.4.6 tcsetattr - Set terminal attributes

CALLING SEQUENCE:

```
#include <termios.h>
int tcsetattr(
    int fildes,
    int optional_actions,
    const struct termios *termios_p
);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

7.4.7 tcsendbreak - Sends a break to a terminal

CALLING SEQUENCE:

```
#include <termios.h>
int tcsendbreak(
    int fildes,
    int duration
);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

This routine is not currently supported by RTEMS but could be in a future version.

7.4.8 tcdrain - Waits for all output to be transmitted to the terminal.

CALLING SEQUENCE:

```
#include <termios.h>
int tcdrain(
    int fildes
);
```

STATUS CODES:

EBADF	Invalid file descriptor
EINTR	Function was interrupted by a signal
ENOTTY	Terminal control function attempted for a file that is not a terminal.

DESCRIPTION:

The `tcdrain()` function waits until all output written to `fildes` has been transmitted.

NOTES:

NONE

7.4.9 tcflush - Discards terminal data

CALLING SEQUENCE:

```
#include <termios.h>
int tcflush(
    int fildes,
    int queue_selector
);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

This routine is not currently supported by RTEMS but could be in a future version.

7.4.10 tcflow - Suspends/restarts terminal output.

CALLING SEQUENCE:

```
#include <termios.h>
int tcflow(
    int fildes,
    int action
);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

This routine is not currently supported by RTEMS but could be in a future version.

7.4.11 tcgetpgrp - Gets foreground process group ID

CALLING SEQUENCE:

```
#include <unistd.h>
pid_t tcgetpgrp(
    int fildes
);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

This routine is not currently supported by RTEMS but could be in a future version.

7.4.12 tcsetpgrp - Sets foreground process group ID

CALLING SEQUENCE:

```
#include <unistd.h>
int tcsetpgrp(
    int fildes,
    pid_t pgid_id
);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

This routine is not currently supported by RTEMS but could be in a future version.

CHAPTER

EIGHT

LANGUAGE-SPECIFIC SERVICES FOR THE C PROGRAMMING LANGUAGE MANAGER

8.1 Introduction

The language-specific services for the C programming language manager is . . .

The directives provided by the language-specific services for the C programming language manager are:

- *setlocale* (page 107) - Set the Current Locale
- *fileno* (page 107) - Obtain File Descriptor Number for this File
- *fdopen* (page 107) - Associate Stream with File Descriptor
- *flockfile* (page 108) - Acquire Ownership of File Stream
- *ftrylockfile* (page 108) - Poll to Acquire Ownership of File Stream
- *funlockfile* (page 108) - Release Ownership of File Stream
- *getc_unlocked* (page 108) - Get Character without Locking
- *getchar_unlocked* (page 109) - Get Character from stdin without Locking
- *putc_unlocked* (page 109) - Put Character without Locking
- *putchar_unlocked* (page 109) - Put Character to stdin without Locking
- *setjmp* (page 110) - Save Context for Non-Local Goto
- *longjmp* (page 110) - Non-Local Jump to a Saved Context
- *sigsetjmp* (page 110) - Save Context with Signal Status for Non-Local Goto
- *siglongjmp* (page 110) - Non-Local Jump with Signal Status to a Saved Context
- *tzset* (page 111) - Initialize Time Conversion Information
- *strtok_r* (page 111) - Reentrant Extract Token from String
- *asctime_r* (page 111) - Reentrant struct tm to ASCII Time Conversion
- *ctime_r* (page 112) - Reentrant time_t to ASCII Time Conversion
- *gmtime_r* (page 112) - Reentrant UTC Time Conversion
- *localtime_r* (page 112) - Reentrant Local Time Conversion
- *rand_r* (page 113) - Reentrant Random Number Generation

8.2 Background

There is currently no text in this section.

8.3 Operations

There is currently no text in this section.

8.4 Directives

This section details the language-specific services for the C programming language manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

8.4.1 setlocale - Set the Current Locale

CALLING SEQUENCE:

```
#include <locale.h>
char *setlocale(int category, const char *locale);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

8.4.2 fileno - Obtain File Descriptor Number for this File

CALLING SEQUENCE:

```
#include <stdio.h>
int fileno(FILE *stream);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

8.4.3 fdopen - Associate Stream with File Descriptor

CALLING SEQUENCE:

```
#include <stdio.h>
FILE *fdopen(int fildes, const char *mode);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

8.4.4 flockfile - Acquire Ownership of File Stream

CALLING SEQUENCE:

```
#include <stdio.h>
void flockfile(FILE *file);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

8.4.5 ftrylockfile - Poll to Acquire Ownership of File Stream

CALLING SEQUENCE:

```
#include <stdio.h>
int ftrylockfile(FILE *file);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

8.4.6 funlockfile - Release Ownership of File Stream

CALLING SEQUENCE:

```
#include <stdio.h>
void funlockfile(FILE *file);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

8.4.7 getc_unlocked - Get Character without Locking

CALLING SEQUENCE:

```
#include <stdio.h>
int getc_unlocked(FILE *stream);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

8.4.8 getchar_unlocked - Get Character from stdin without Locking

CALLING SEQUENCE:

```
#include <stdio.h>
int getchar_unlocked(void);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

8.4.9 putc_unlocked - Put Character without Locking

CALLING SEQUENCE:

```
#include <stdio.h>
int putc_unlocked(int c, FILE *stream);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

8.4.10 putchar_unlocked - Put Character to stdin without Locking

CALLING SEQUENCE:

```
#include <stdio.h>
int putchar_unlocked(int c);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

8.4.11 setjmp - Save Context for Non-Local Goto

CALLING SEQUENCE:

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

8.4.12 longjmp - Non-Local Jump to a Saved Context

CALLING SEQUENCE:

```
#include <setjmp.h>
void longjmp(jmp_buf env, int val);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

8.4.13 sigsetjmp - Save Context with Signal Status for Non-Local Goto

CALLING SEQUENCE:

```
#include <setjmp.h>
int sigsetjmp(sigjmp_buf env, int savemask);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

8.4.14 siglongjmp - Non-Local Jump with Signal Status to a Saved Context

CALLING SEQUENCE:

```
#include <setjmp.h>
void siglongjmp(sigjmp_buf env, int val);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

8.4.15 tzset - Initialize Time Conversion Information

CALLING SEQUENCE:

```
#include <time.h>
extern int daylight;
extern long timezone;
extern char *tzname[2];
void tzset(void);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

8.4.16 strtok_r - Reentrant Extract Token from String

CALLING SEQUENCE:

```
#include <string.h>
char *strtok_r(char *restrict s, const char *restrict sep,
char **restrict state);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

8.4.17 asctime_r - Reentrant struct tm to ASCII Time Conversion

CALLING SEQUENCE:

```
#include <time.h>
char *asctime_r(const struct tm *restrict tm, char *restrict buf);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

8.4.18 ctime_r - Reentrant time_t to ASCII Time Conversion

CALLING SEQUENCE:

```
#include <time.h>
char *ctime_r(const time_t *clock, char *buf);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

8.4.19 gmtime_r - Reentrant UTC Time Conversion

CALLING SEQUENCE:

```
#include <time.h>
struct tm *gmtime_r(const time_t *restrict timer,
struct tm *restrict result);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

8.4.20 localtime_r - Reentrant Local Time Conversion

CALLING SEQUENCE:

```
#include <time.h>
struct tm *localtime_r(const time_t *restrict timer,
struct tm *restrict result);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

8.4.21 rand_r - Reentrant Random Number Generation

CALLING SEQUENCE:

```
#include <stdlib.h>
int rand_r(unsigned *seed);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

CHAPTER

NINE

SYSTEM DATABASES MANAGER

9.1 Introduction

The system databases manager is . . .

The directives provided by the system databases manager are:

- *getgrgid* (page 119) - Get Group File Entry for ID
- *getgrgid_r* (page 119) - Reentrant Get Group File Entry
- *getgrnam* (page 119) - Get Group File Entry for Name
- *getgrnam_r* (page 120) - Reentrant Get Group File Entry for Name
- *getpwuid* (page 120) - Get Password File Entry for UID
- *getpwuid_r* (page 120) - Reentrant Get Password File Entry for UID
- *getpwnam* (page 121) - Get Password File Entry for Name
- *getpwnam_r* (page 121) - Reentrant Get Password File Entry for Name

9.2 Background

There is currently no text in this section.

9.3 Operations

There is currently no text in this section.

9.4 Directives

This section details the system databases manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

9.4.1 getgrgid - Get Group File Entry for ID

CALLING SEQUENCE:

```
#include <grp.h>
struct group *getgrgid(
    gid_t gid
);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

9.4.2 getgrgid_r - Reentrant Get Group File Entry

CALLING SEQUENCE:

```
#include <grp.h>
int getgrgid_r(
    gid_t gid,
    struct group *grp,
    char *buffer,
    size_t bufsize,
    struct group **result
);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

9.4.3 getgrnam - Get Group File Entry for Name

CALLING SEQUENCE:

```
#include <grp.h>
struct group *getgrnam(
    const char *name
);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

9.4.4 getgrnam_r - Reentrant Get Group File Entry for Name

CALLING SEQUENCE:

```
#include <grp.h>
int getgrnam_r(
    const char *name,
    struct group *grp,
    char *buffer,
    size_t bufsize,
    struct group **result
);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

9.4.5 getpwuid - Get Password File Entry for UID

CALLING SEQUENCE:

```
#include <pwd.h>
struct passwd *getpwuid(
    uid_t uid
);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

9.4.6 getpwuid_r - Reentrant Get Password File Entry for UID

CALLING SEQUENCE:

```
#include <pwd.h>
int getpwuid_r(
    uid_t uid,
    struct passwd *pwd,
    char *buffer,
    size_t bufsize,
    struct passwd **result
);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

9.4.7 getpwnam - Password File Entry for Name

CALLING SEQUENCE:

```
#include <pwd.h>
struct passwd *getpwnam(
    const char *name
);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

9.4.8 getpwnam_r - Reentrant Get Password File Entry for Name

CALLING SEQUENCE:

```
#include <pwd.h>
int getpwnam_r(
    const char *name,
    struct passwd *pwd,
    char *buffer,
    size_t bufsize,
    struct passwd **result
);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

CHAPTER

TEN

SEMAPHORE MANAGER

10.1 Introduction

The semaphore manager provides functions to allocate, delete, and control semaphores. This manager is based on the POSIX 1003.1 standard.

The directives provided by the semaphore manager are:

- *sem_init* (page 127) - Initialize an unnamed semaphore
- *sem_destroy* (page 127) - Destroy an unnamed semaphore
- *sem_open* (page 128) - Open a named semaphore
- *sem_close* (page 129) - Close a named semaphore
- *sem_unlink* (page 129) - Remove a named semaphore
- *sem_wait* (page 130) - Lock a semaphore
- *sem_trywait* (page 130) - Lock a semaphore
- *sem_timedwait* (page 131) - Wait on a Semaphore for a Specified Time
- *sem_post* (page 131) - Unlock a semaphore
- *sem_getvalue* (page 132) - Get the value of a semeaphore

10.2 Background

10.2.1 Theory

Semaphores are used for synchronization and mutual exclusion by indicating the availability and number of resources. The task (the task which is returning resources) notifying other tasks of an event increases the number of resources held by the semaphore by one. The task (the task which will obtain resources) waiting for the event decreases the number of resources held by the semaphore by one. If the number of resources held by a semaphore is insufficient (namely 0), the task requiring resources will wait until the next time resources are returned to the semaphore. If there is more than one task waiting for a semaphore, the tasks will be placed in the queue.

10.2.2 "sem_t" Structure

The `sem_t` structure is used to represent semaphores. It is passed as an argument to the semaphore directives and is defined as follows:

```
typedef int sem_t;
```

10.2.3 Building a Semaphore Attribute Set

10.3 Operations

10.3.1 Using as a Binary Semaphore

Although POSIX supports mutexes, they are only visible between threads. To work between processes, a binary semaphore must be used.

Creating a semaphore with a limit on the count of 1 effectively restricts the semaphore to being a binary semaphore. When the binary semaphore is available, the count is 1. When the binary semaphore is unavailable, the count is 0.

Since this does not result in a true binary semaphore, advanced binary features like the Priority Inheritance and Priority Ceiling Protocols are not available.

There is currently no text in this section.

10.4 Directives

This section details the semaphore manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

10.4.1 sem_init - Initialize an unnamed semaphore

CALLING SEQUENCE:

```
int sem_init(
    sem_t        *sem,
    int           pshared,
    unsigned int  value
);
```

STATUS CODES:

EINVA	The value argument exceeds SEM_VALUE_MAX
ENOSP	A resource required to initialize the semaphore has been exhausted The limit on semaphores (SEM_VALUE_MAX) has been reached
ENOSY	The function sem_init is not supported by this implementation
EPERM	The process lacks appropriate privileges to initialize the semaphore

DESCRIPTION:

The `sem_init` function is used to initialize the unnamed semaphore referred to by `sem`. The value of the initialized semaphore is the parameter `value`. The semaphore remains valid until it is destroyed.

NOTES:

If the functions completes successfully, it shall return a value of zero. otherwise, it shall return a value of -1 and set `errno` to specify the error that occurred.

Multiprocessing is currently not supported in this implementation.

10.4.2 sem_destroy - Destroy an unnamed semaphore

CALLING SEQUENCE:

```
int sem_destroy(
    sem_t *sem
);
```

STATUS CODES:

EINVAL	The value argument exceeds SEM_VALUE_MAX
ENOSYS	The function `sem_init` is not supported by this implementation
EBUSY	There are currently processes blocked on the semaphore

DESCRIPTION:

The sem_destroy function is used to destroy an unnamed semaphore refered to by sem. sem_destroy can only be used on a semaphore that was created using sem_init.

NOTES:

If the functions completes successfully, it shall return a value of zero. Otherwise, it shall return a value of -1 and set errno to specify the error that occurred.

Multiprocessing is currently not supported in this implementation.

10.4.3 sem_open - Open a named semaphore

CALLING SEQUENCE:

```
int sem_open(
    const char *name,
    int         oflag
);
```

ARGUMENTS:

The following flag bit may be set in oflag:

O_CREA	Creates the semaphore if it does not already exist. If O_CREAT is set and the semaphore already exists then O_CREAT has no effect. Otherwise, sem_open() creates a semaphore. The O_CREAT flag requires the third and fourth argument: mode and value of type mode_t and unsigned int, respectively.
O_EXCL	If O_EXCL and O_CREAT are set, all call to sem_open() shall fail if the semaphore name exists

STATUS CODES:

EACCES	Valid name specified but oflag permissions are denied, or the semaphore name specified does not exist and permission to create the named semaphore is denied.
EEXIST	O_CREAT and O_EXCL are set and the named semaphore already exists.
EINTR	The sem_open() operation was interrupted by a signal.
EINVAL	The sem_open() operation is not supported for the given name.
EMFILE	Too many semaphore descriptors or file descriptors in use by this process.
ENAMETO	The length of the name exceed PATH_MAX or name component is longer than NAME_MAX while POSIX_NO_TRUNC is in effect.
ENOENT	O_CREAT is not set and the named semaphore does not exist.
ENOSPC	There is insufficient space for the creation of a new named semaphore.
ENOSYS	The function sem_open() is not supported by this implementation.

DESCRIPTION:

The sem_open() function establishes a connection between a specified semaphore and a process. After a call to sem_open with a specified semaphore name, a process can reference to semaphore by the associated name using the address returned by the call. The oflag arguments listed above control the state of the semaphore by determining if the semaphore is created or accessed by a call to sem_open().

NOTES:

10.4.4 sem_close - Close a named semaphore

CALLING SEQUENCE:

```
int sem_close(
    sem_t *sem_close
);
```

STATUS CODES:

EACCES	The semaphore argument is not a valid semaphore descriptor.
ENOSYS	The function sem_close is not supported by this implementation.

DESCRIPTION:

The sem_close() function is used to indicate that the calling process is finished using the named semaphore indicated by sem. The function sem_close deallocates any system resources that were previously allocated by a sem_open system call. If sem_close() completes successfully it returns a 1, otherwise a value of -1 is return and errno is set.

NOTES:

10.4.5 sem_unlink - Unlink a semaphore

CALLING SEQUENCE:

```
int sem_unlink(
    const char *name
);
```

STATUS CODES:

EACCESS	Permission is denied to unlink a semaphore.
ENAMETOOLONG	The length of the strong name exceed NAME_MAX while POSIX_NO_TRUNC is in effect.
ENOENT	The name of the semaphore does not exist.
ENOSPC	There is insufficient space for the creation of a new named semaphore.
ENOSYS	The function sem_unlink is not supported by this implementation.

DESCRIPTION:

The sem_unlink() function shall remove the semaphore name by the string name. If a process is currently accessing the name semaphore, the sem_unlink command has no effect. If one or more processes have the semaphore open when the sem_unlink function is called, the destruction of semaphores shall be postponed until all reference to semaphore are destroyed by calls to sem_close, _exit(), or exec. After all references have been destroyed, it returns immediately.

If the termination is successful, the function shall return 0. Otherwise, a -1 is returned and the errno is set.

NOTES:

10.4.6 sem_wait - Wait on a Semaphore

CALLING SEQUENCE:

```
int sem_wait(
    sem_t *sem
);
```

STATUS CODES:

EINVAL	The sem argument does not refer to a valid semaphore

DESCRIPTION:

This function attempts to lock a semaphore specified by sem. If the semaphore is available, then the semaphore is locked (i.e., the semaphore value is decremented). If the semaphore is unavailable (i.e., the semaphore value is zero), then the function will block until the semaphore becomes available. It will then successfully lock the semaphore. The semaphore remains locked until released by a sem_post() call.

If the call is unsuccessful, then the function returns -1 and sets errno to the appropriate error code.

NOTES:

Multiprocessing is not supported in this implementation.

10.4.7 sem_trywait - Non-blocking Wait on a Semaphore

CALLING SEQUENCE:

```
int sem_trywait(
    sem_t *sem
);
```

STATUS CODES:

EAGAIN	The semaphore is not available (i.e., the semaphore value is zero), so the semaphore could not be locked.
EINVAL	The sem argument does not refewr to a valid semaphore

DESCRIPTION:

This function attempts to lock a semaphore specified by sem. If the semaphore is available, then the semaphore is locked (i.e., the semaphore value is decremented) and the function returns a value of 0. The semaphore remains locked until released by a sem_post() call. If the semaphore is unavailable (i.e., the semaphore value is zero), then the function will return a value of -1 immediately and set errno to EAGAIN.

If the call is unsuccessful, then the function returns -1 and sets errno to the appropriate error code.

NOTES:

Multiprocessing is not supported in this implementation.

10.4.8 sem_timedwait - Wait on a Semaphore for a Specified Time

CALLING SEQUENCE:

```
int sem_timedwait(
    sem_t                *sem,
    const struct timespec *abstime
);
```

STATUS CODES:

EINVAL	The sem argument does not refewr to a valid semaphore
EINVAL	The nanoseconds field of timeout is invalid.
ETIMEDOUT	The calling thread was unable to get the semaphore within the specified timeout period.

DESCRIPTION:

This function attemtps to lock a semaphore specified by sem, and will wait for the semaphore until the absolute time specified by abstime. If the semaphore is available, then the semaphore is locked (i.e., the semaphore value is decremented) and the function returns a value of 0. The semaphore remains locked until released by a sem_post() call. If the semaphore is unavailable, then the function will wait for the semaphore to become available for the amount of time specified by timeout.

If the semaphore does not become available within the interval specified by timeout, then the function returns -1 and sets errno to EAGAIN. If any other error occurs, the function returns -1 and sets errno to the appropriate error code.

NOTES:

Multiprocessing is not supported in this implementation.

10.4.9 sem_post - Unlock a Semaphore

CALLING SEQUENCE:

```
int sem_post(
    sem_t *sem
);
```

STATUS CODES:

EINVAL	The sem argument does not refer to a valid semaphore

DESCRIPTION:

This function attempts to release the semaphore specified by sem. If other tasks are waiting on the semaphore, then one of those tasks (which one depends on the scheduler being used) is allowed to lock the semaphore and return from its sem_wait(), sem_trywait(), or sem_timedwait() call. If there are no other tasks waiting on the semaphore, then the semaphore value is simply incremented. sem_post() returns 0 upon successful completion.

If an error occurs, the function returns -1 and sets errno to the appropriate error code.

NOTES:

Multiprocessing is not supported in this implementation.

10.4.10 sem_getvalue - Get the value of a semaphore

CALLING SEQUENCE:

```
int sem_getvalue(
    sem_t *sem,
    int   *sval
);
```

STATUS CODES:

EINVAL	The sem argument does not refer to a valid semaphore
ENOSYS	The function sem_getvalue is not supported by this implementation

DESCRIPTION:

The sem_getvalue functions sets the location referenced by the sval argument to the value of the semaphore without affecting the state of the semaphore. The updated value represents a semaphore value that occurred at some point during the call, but is not necessarily the actual value of the semaphore when it returns to the calling process.

If sem is locked, the value returned by sem_getvalue will be zero or a negative number whose absolute value is the number of processes waiting for the semaphore at some point during the call.

NOTES:

If the functions completes successfully, it shall return a value of zero. Otherwise, it shall return a value of -1 and set errno to specify the error that occurred.

CHAPTER

ELEVEN

MUTEX MANAGER

11.1 Introduction

The mutex manager implements the functionality required of the mutex manager as defined by POSIX 1003.1b-1996. This standard requires that a compliant operating system provide the facilties to ensure that threads can operate with mutual exclusion from one another and defines the API that must be provided.

The services provided by the mutex manager are:

- *pthread_mutexattr_init* (page 137) - Initialize a Mutex Attribute Set
- *pthread_mutexattr_destroy* (page 137) - Destroy a Mutex Attribute Set
- *pthread_mutexattr_setprotocol* (page 138) - Set the Blocking Protocol
- *pthread_mutexattr_getprotocol* (page 138) - Get the Blocking Protocol
- *pthread_mutexattr_setprioceiling* (page 139) - Set the Priority Ceiling
- *pthread_mutexattr_getprioceiling* (page 139) - Get the Priority Ceiling
- *pthread_mutexattr_setpshared* (page 140) - Set the Visibility
- *pthread_mutexattr_getpshared* (page 140) - Get the Visibility
- *pthread_mutex_init* (page 140) - Initialize a Mutex
- *pthread_mutex_destroy* (page 141) - Destroy a Mutex
- *pthread_mutex_lock* (page 141) - Lock a Mutex
- *pthread_mutex_trylock* (page 142) - Poll to Lock a Mutex
- *pthread_mutex_timedlock* (page 142) - Lock a Mutex with Timeout
- *pthread_mutex_unlock* (page 143) - Unlock a Mutex
- *pthread_mutex_setprioceiling* (page 143) - Dynamically Set the Priority Ceiling
- *pthread_mutex_getprioceiling* (page 143) - Dynamically Get the Priority Ceiling

11.2 Background

11.2.1 Mutex Attributes

Mutex attributes are utilized only at mutex creation time. A mutex attribute structure may be initialized and passed as an argument to the `mutex_init` routine. Note that the priority ceiling of a mutex may be set at run-time.

blocking protcol	is the XXX
priority ceiling	is the XXX
pshared	is the XXX

11.2.2 PTHREAD_MUTEX_INITIALIZER

This is a special value that a variable of type `pthread_mutex_t` may be statically initialized to as shown below:

```
1 pthread_mutex_t my_mutex = PTHREAD_MUTEX_INITIALIZER;
```

This indicates that `my_mutex` will be automatically initialized by an implicit call to `pthread_mutex_init` the first time the mutex is used.

Note that the mutex will be initialized with default attributes.

11.3 Operations

There is currently no text in this section.

11.4 Services

This section details the mutex manager's services. A subsection is dedicated to each of this manager's services and describes the calling sequence, related constants, usage, and status codes.

11.4.1 pthread_mutexattr_init - Initialize a Mutex Attribute Set

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_mutexattr_init(
    pthread_mutexattr_t *attr
);
```

STATUS CODES:

EINVAL
The attribute pointer argument is invalid.

DESCRIPTION:

The `pthread_mutexattr_init` routine initializes the mutex attributes object specified by `attr` with the default value for all of the individual attributes.

NOTES:

XXX insert list of default attributes here.

11.4.2 pthread_mutexattr_destroy - Destroy a Mutex Attribute Set

CALLING SEQUENCE:

```
#include <pthread.h>
    int pthread_mutexattr_destroy(
    pthread_mutexattr_t *attr
);
```

STATUS CODES:

`EINVAL`	The attribute pointer argument is invalid.
`EINVAL`	The attribute set is not initialized.

DESCRIPTION:

The `pthread_mutex_attr_destroy` routine is used to destroy a mutex attributes object. The behavior of using an attributes object after it is destroyed is implementation dependent.

NOTES:

NONE

11.4.3 pthread_mutexattr_setprotocol - Set the Blocking Protocol

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_mutexattr_setprotocol(
    pthread_mutexattr_t *attr,
    int                  protocol
);
```

STATUS CODES:

EINVAL	The attribute pointer argument is invalid.
EINVAL	The attribute set is not initialized.
EINVAL	The protocol argument is invalid.

DESCRIPTION:

The `pthread_mutexattr_setprotocol` routine is used to set value of the `protocol` attribute. This attribute controls the order in which threads waiting on this mutex will receive it.

The `protocol` can be one of the following:

PTHREAD_PRIO_NONE	in which case blocking order is FIFO.
PTHREAD_PRIO_INHERIT	in which case blocking order is priority with the priority inheritance protocol in effect.
PTHREAD_PRIO_PROTECT	in which case blocking order is priority with the priority ceiling protocol in effect.

NOTES:

There is currently no way to get simple priority blocking ordering with POSIX mutexes even though this could easily by supported by RTEMS.

11.4.4 pthread_mutexattr_getprotocol - Get the Blocking Protocol

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_mutexattr_getprotocol(
    pthread_mutexattr_t *attr,
    int                 *protocol
);
```

STATUS CODES:

EINVAL	The attribute pointer argument is invalid.
EINVAL	The attribute set is not initialized.
EINVAL	The protocol pointer argument is invalid.

DESCRIPTION:

The pthread_mutexattr_getprotocol routine is used to obtain the value of the protocol attribute. This attribute controls the order in which threads waiting on this mutex will receive it.

NOTES:

NONE

11.4.5 pthread_mutexattr_setprioceiling - Set the Priority Ceiling

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_mutexattr_setprioceiling(
    pthread_mutexattr_t *attr,
    int                 prioceiling
);
```

STATUS CODES:

EINVAL	The attribute pointer argument is invalid.
EINVAL	The attribute set is not initialized.
EINVAL	The prioceiling argument is invalid.

DESCRIPTION:

The pthread_mutexattr_setprioceiling routine is used to set value of the prioceiling attribute. This attribute specifies the priority that is the ceiling for threads obtaining this mutex. Any task obtaining this mutex may not be of greater priority that the ceiling. If it is of lower priority, then its priority will be elevated to prioceiling.

NOTES:

NONE

11.4.6 pthread_mutexattr_getprioceiling - Get the Priority Ceiling

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_mutexattr_getprioceiling(
    const pthread_mutexattr_t *attr,
    int                       *prioceiling
);
```

STATUS CODES:

EINVAL	The attribute pointer argument is invalid.
EINVAL	The attribute set is not initialized.
EINVAL	The prioceiling pointer argument is invalid.

DESCRIPTION:

The pthread_mutexattr_getprioceiling routine is used to obtain the value of the prioceiling attribute. This attribute specifies the priority ceiling for this mutex.

NOTES:

NONE

11.4.7 pthread_mutexattr_setpshared - Set the Visibility

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_mutexattr_setpshared(
    pthread_mutexattr_t *attr,
    int                  pshared
);
```

STATUS CODES:

EINVAL	The attribute pointer argument is invalid.
EINVAL	The attribute set is not initialized.
EINVAL	The pshared argument is invalid.

DESCRIPTION:

NOTES:

11.4.8 pthread_mutexattr_getpshared - Get the Visibility

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_mutexattr_getpshared(
    const pthread_mutexattr_t *attr,
    int                       *pshared
);
```

STATUS CODES:

EINVAL	The attribute pointer argument is invalid.
EINVAL	The attribute set is not initialized.
EINVAL	The pshared pointer argument is invalid.

DESCRIPTION:

NOTES:

11.4.9 pthread_mutex_init - Initialize a Mutex

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_mutex_init(
    pthread_mutex_t           *mutex,
    const pthread_mutexattr_t *attr
);
```

STATUS CODES:

EINVAL	The attribute set is not initialized.
EINVAL	The specified protocol is invalid.
EAGAIN	The system lacked the necessary resources to initialize another mutex.
ENOMEM	Insufficient memory exists to initialize the mutex.
EBUSY	Attempted to reinialize the object reference by mutex, a previously initialized, but not yet destroyed.

DESCRIPTION:

NOTES:

11.4.10 pthread_mutex_destroy - Destroy a Mutex

CALLING SEQUENCE:

```
#include <pthread.h>
    int pthread_mutex_destroy(
    pthread_mutex_t *mutex
);
```

STATUS CODES:

EINVAL	The specified mutex is invalid.
EBUSY	Attempted to destroy the object reference by mutex, while it is locked or referenced by another thread.

DESCRIPTION:

NOTES:

11.4.11 pthread_mutex_lock - Lock a Mutex

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_mutex_lock(
    pthread_mutex_t *mutex
);
```

STATUS CODES:

EINVAL	The specified mutex is invalid.
EINVAL	The mutex has the protocol attribute of PTHREAD_PRIO_PROTECT and the priority of the calling thread is higher than the current priority ceiling.
EDEADL	The current thread already owns the mutex.

DESCRIPTION:

NOTES:

11.4.12 pthread_mutex_trylock - Poll to Lock a Mutex

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_mutex_trylock(
    pthread_mutex_t *mutex
);
```

STATUS CODES:

EINVA	The specified mutex is invalid.
EINVA	The mutex has the protocol attribute of PTHREAD_PRIO_PROTECT and the priority of the calling thread is higher than the current priority ceiling.
EBUSY	The mutex is already locked.

DESCRIPTION:

NOTES:

11.4.13 pthread_mutex_timedlock - Lock a Mutex with Timeout

CALLING SEQUENCE:

```
#include <pthread.h>
#include <time.h>
int pthread_mutex_timedlock(
    pthread_mutex_t       *mutex,
    const struct timespec *timeout
);
```

STATUS CODES:

EINVAL	The specified mutex is invalid.
EINVAL	The nanoseconds field of timeout is invalid.
EINVAL	The mutex has the protocol attribute of PTHREAD_PRIO_PROTECT and the priority of the calling thread is higher than the current priority ceiling.
EDEADLK	The current thread already owns the mutex.
ETIMEDO	The calling thread was unable to obtain the mutex within the specified timeout period.

DESCRIPTION:

NOTES:

11.4.14 pthread_mutex_unlock - Unlock a Mutex

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_mutex_unlock(
    pthread_mutex_t *mutex
);
```

STATUS CODES:

EINVAL	The specified mutex is invalid.

DESCRIPTION:

NOTES:

11.4.15 pthread_mutex_setprioceiling - Dynamically Set the Priority Ceiling

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_mutex_setprioceiling(
    pthread_mutex_t *mutex,
    int              prioceiling,
    int             *oldceiling
);
```

STATUS CODES:

EINVAL	The oldceiling pointer parameter is invalid.
EINVAL	The prioceiling parameter is an invalid priority.
EINVAL	The specified mutex is invalid.

DESCRIPTION:

NOTES:

11.4.16 pthread_mutex_getprioceiling - Get the Current Priority Ceiling

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_mutex_getprioceiling(
    pthread_mutex_t *mutex,
    int             *prioceiling
);
```

STATUS CODES:

EINVAL	The prioceiling pointer parameter is invalid.
EINVAL	The specified mutex is invalid.

DESCRIPTION:

NOTES:

CHAPTER

TWELVE

CONDITION VARIABLE MANAGER

12.1 Introduction

The condition variable manager . . .

The directives provided by the condition variable manager are:

- *pthread_condattr_init* (page 149) - Initialize a Condition Variable Attribute Set
- *pthread_condattr_destroy* (page 149) - Destroy a Condition Variable Attribute Set
- *pthread_condattr_setpshared* (page 149) - Set Process Shared Attribute
- *pthread_condattr_getpshared* (page 150) - Get Process Shared Attribute
- *pthread_cond_init* (page 150) - Initialize a Condition Variable
- *pthread_cond_destroy* (page 150) - Destroy a Condition Variable
- *pthread_cond_signal* (page 151) - Signal a Condition Variable
- *pthread_cond_broadcast* (page 151) - Broadcast a Condition Variable
- *pthread_cond_wait* (page 152) - Wait on a Condition Variable
- *pthread_cond_timedwait* (page 152) - With with Timeout a Condition Variable

12.2 Background

There is currently no text in this section.

12.3 Operations

There is currently no text in this section.

12.4 Directives

This section details the condition variable manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

12.4.1 pthread_condattr_init - Initialize a Condition Variable Attribute Set

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_condattr_init(
    pthread_condattr_t *attr
);
```

STATUS CODES:

- – ENOMEM
  – Insufficient memory is available to initialize the condition variable attributes object.

DESCRIPTION:

NOTES:

12.4.2 pthread_condattr_destroy - Destroy a Condition Variable Attribute Set

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_condattr_destroy(
    pthread_condattr_t *attr
);
```

STATUS CODES:

EINVAL	The attribute object specified is invalid.

DESCRIPTION:

NOTES:

12.4.3 pthread_condattr_setpshared - Set Process Shared Attribute

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_condattr_setpshared(
    pthread_condattr_t *attr,
    int                 pshared
);
```

STATUS CODES:

EINVAL	Invalid argument passed.

DESCRIPTION:

NOTES:

12.4.4 pthread_condattr_getpshared - Get Process Shared Attribute

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_condattr_getpshared(
    const pthread_condattr_t *attr,
    int                      *pshared
);
```

STATUS CODES:

EINVAL	Invalid argument passed.

DESCRIPTION:

NOTES:

12.4.5 pthread_cond_init - Initialize a Condition Variable

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_cond_init(
    pthread_cond_t           *cond,
    const pthread_condattr_t *attr
);
```

STATUS CODES:

EAGAIN	The system lacked a resource other than memory necessary to create the initialize the condition variable object.
ENOMEM	Insufficient memory is available to initialize the condition variable object.
EBUSY	The specified condition variable has already been initialized.
EINVAL	The specified attribute value is invalid.

DESCRIPTION:

NOTES:

12.4.6 pthread_cond_destroy - Destroy a Condition Variable

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_cond_destroy(
    pthread_cond_t *cond
);
```

STATUS CODES:

EINVAL	The specified condition variable is invalid.
EBUSY	The specified condition variable is currently in use.

DESCRIPTION:

NOTES:

12.4.7 pthread_cond_signal - Signal a Condition Variable

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_cond_signal(
    pthread_cond_t *cond
);
```

STATUS CODES:

EINVAL	The specified condition variable is not valid.

DESCRIPTION:

NOTES:

This routine should not be invoked from a handler from an asynchronous signal handler or an interrupt service routine.

12.4.8 pthread_cond_broadcast - Broadcast a Condition Variable

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_cond_broadcast(
    pthread_cond_t *cond
);
```

STATUS CODES:

EINVAL	The specified condition variable is not valid.

DESCRIPTION:

NOTES:

This routine should not be invoked from a handler from an asynchronous signal handler or an interrupt service routine.

12.4.9 pthread_cond_wait - Wait on a Condition Variable

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_cond_wait(
    pthread_cond_t *cond,
    pthread_mutex_t *mutex
);
```

STATUS CODES:

EINV	The specified condition variable or mutex is not initialized OR different mutexes were specified for concurrent `pthread_cond_wait()` and `pthread_cond_timedwait()` operations on the same condition variable OR the mutex was not owned by the current thread at the time of the call.

DESCRIPTION:

NOTES:

12.4.10 pthread_cond_timedwait - Wait with Timeout a Condition Variable

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_cond_timedwait(
    pthread_cond_t        *cond,
    pthread_mutex_t       *mutex,
    const struct timespec *abstime
);
```

STATUS CODES:

EINV	The nanoseconds field of timeout is invalid.
EINV	The specified condition variable or mutex is not initialized OR different mutexes were specified for concurrent `pthread_cond_wait()` and `pthread_cond_timedwait()` operations on the same condition variable OR the mutex was not owned by the current thread at the time of the call.
ETIM	The specified time has elapsed without the condition variable being satisfied.

DESCRIPTION:

NOTES:

CHAPTER

THIRTEEN

MEMORY MANAGEMENT MANAGER

13.1 Introduction

The memory management manager is . . .

The directives provided by the memory management manager are:

- *mlockall* (page 157) - Lock the Address Space of a Process
- *munlockall* (page 157) - Unlock the Address Space of a Process
- *mlock* (page 157) - Lock a Range of the Process Address Space
- *munlock* (page 158) - Unlock a Range of the Process Address Space
- *mmap* (page 158) - Map Process Addresses to a Memory Object
- *munmap* (page 159) - Unmap Previously Mapped Addresses
- *mprotect* (page 160) - Change Memory Protection
- *msync* (page 160) - Memory Object Synchronization
- *shm_open* (page 161) - Open a Shared Memory Object
- *shm_unlink* (page 162) - Remove a Shared Memory Object

13.2 Background

There is currently no text in this section.

13.3 Operations

There is currently no text in this section.

13.4 Directives

This section details the memory management manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

13.4.1 mlockall - Lock the Address Space of a Process

CALLING SEQUENCE:

```
#include <sys/mman.h>
int mlockall(
    int flags
);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

13.4.2 munlockall - Unlock the Address Space of a Process

CALLING SEQUENCE:

```
#include <sys/mman.h>
int munlockall(
    void
);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

13.4.3 mlock - Lock a Range of the Process Address Space

CALLING SEQUENCE:

```
#include <sys/mman.h>
int mlock(
    const void *addr,
    size_t len
);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

13.4.4 munlock - Unlock a Range of the Process Address Space

CALLING SEQUENCE:

```
#include <sys/mman.h>
int munlock(
    const void *addr,
    size_t len
);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

13.4.5 mmap - Map Process Addresses to a Memory Object

CALLING SEQUENCE:

```
#include <sys/mman.h>
void *mmap(
    void *addr,
    size_t len,
    int prot,
    int flags,
    int fildes,
    off_t off
);
```

STATUS CODES:

EBADF	The fildes argument is not a valid open file descriptor.
EINVA	The value of len is zero.
EINVA	The value of flags is invalid (neither MAP_PRIVATE nor MAP_SHARED is set).
EINVA	The addr argument (if MAP_FIXED was specified) or off is not a multiple of the page size as returned by sysconf(), or is considered invalid by the implementation.
ENODE	The fildes argument refers to a file whose type is not supported by mmap.
ENOME	MAP_FIXED was specified, and the range [addr,addr+len) exceeds that allowed for the address space of a process; or, if MAP_FIXED was not specified and there is insufficient room in the address space to effect the mapping.
ENOTS	MAP_FIXED or MAP_PRIVATE was specified in the flags argument and the implementation does not support this functionality.
ENOTS	The implementation does not support the combination of accesses requested in the prot argument.
ENXIO	Addresses in the range [off,off+len) are invalid for the object specified by fildes.
ENXIO	MAP_FIXED was specified in flags and the combination of addr, len, and off is invalid for the object specified by fildes.
EOVER	The file is a regular file and the value of off plus len exceeds the offset maximum established in the open file description associated with fildes.

DESCRIPTION:

mmap establishes a mapping between an address pa for len bytes to the memory object represented by the file descriptor fildes at offset off for len bytes. The value of pa is an implementation-defined function of the parameter addr and the values of flags. A successful mmap() call shall return pa as its result. An unsuccessful call returns MAP_FAILED and sets errno accordingly.

NOTES:

RTEMS is a single address space operating system without privilege separation between the kernel and user space. Therefore, the implementation of mmap has a number of implementation-specific issues to be aware of:

- Read, write and execute permissions are allowed because the memory in RTEMS does not normally have protections but we cannot hide access to memory. Thus, the use of PROT_NONE for the prot argument is not supported. Similarly, there is no restriction of write access, so PROT_WRITE must be in the prot argument.
- Anonymous mappings must have fildes set to -1 and off set to 0. Shared mappings are not supported with Anonymous mappings.
- MAP_FIXED is not supported for shared memory objects with MAP_SHARED.
- Support for shared mappings is dependent on the underlying object's filesystem implementation of an mmap_h file operation handler.

13.4.6 munmap - Unmap Previously Mapped Addresses

CALLING SEQUENCE:

```
1 #include <sys/mman.h>
2 int munmap(
3     void *addr,
```

(continues on next page)

(continued from previous page)

```
    size_t len
);
```

STATUS CODES:

EINVAL	Addresses in the range [addr,addr+len) are outside the valid range for the address space.
EINVAL	The len argument is 0.

DESCRIPTION:

The `munmap()` function shall remove any mappings for those entire pages containing any part of the address space of the process starting at `addr` and continuing for `len` bytes. If there are no mappings in the specified address range, then `munmap()` has no effect.

Upon successful completion, `munmap()` shall return 0; otherwise, it shall return -1 and set `errno` to indicate the error.

NOTES:

13.4.7 mprotect - Change Memory Protection

CALLING SEQUENCE:

```
#include <sys/mman.h>
int mprotect(
    void *addr,
    size_t len,
    int prot
);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

13.4.8 msync - Memory Object Synchronization

CALLING SEQUENCE:

```
#include <sys/mman.h>
int msync(
    void *addr,
    size_t len,
    int flags
);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

13.4.9 shm_open - Open a Shared Memory Object

CALLING SEQUENCE:

```
#include <sys/mman.h>
int shm_open(
    const char *name,
    int oflag,
    mode_t mode
);
```

STATUS CODES:

EACCE	The shared memory object exists and the permissions specified by oflag are denied, or the shared memory object does not exist and permission to create the shared memory object is denied, or O_TRUNC is specified and write permission is denied.
EEXIS	O_CREAT and O_EXCL are set and the named shared memory object already exists.
EINVA	The shm_open() operation is not supported for the given name.
EMFIL	All file descriptors available to the process are currently open.
ENFIL	Too many shared memory objects are currently open in the system.
ENOEN	O_CREAT is not set and the named shared memory object does not exist.
ENOSP	There is insufficient space for the creation of the new shared memory object.
ENAME	The length of the name argument exceeds _POSIX_PATH_MAX.

DESCRIPTION:

The shm_open() function shall establish a connection between a shared memory object and a file descriptor. It shall create an open file description that refers to the shared memory object and a file descriptor that refers to that open file description. The name argument points to a string naming a shared memory object.

If successful, shm_open() shall return a file descriptor for the shared memory object. Upon successful completion, the shm_open() function shall return a non-negative integer representing the file descriptor. Otherwise, it shall return -1 and set errno to indicate the error.

NOTES:

An application can set the _POSIX_Shm_Object_operations to control the behavior of shared memory objects when accessed via the file descriptor.

The name must be valid for an RTEMS SuperCore Object.

13.4.10 shm_unlink - Remove a Shared Memory Object

CALLING SEQUENCE:

```
#include <sys/mman.h>
int shm_unlink(
    const char *name
);
```

STATUS CODES:

ENOENT	The named shared memory object does not exist.
ENAMETOOLONG	The length of the name argument exceeds _POSIX_PATH_MAX.

DESCRIPTION:

The shm_unlink() function shall remove the name of the shared memory object named by the string pointed to by name.

If one or more references to the shared memory object exist when the object is unlinked, the name shall be removed before shm_unlink() returns, but the removal of the memory object contents shall be postponed until all open and map references to the shared memory object have been removed.

Even if the object continues to exist after the last shm_unlink(), reuse of the name shall subsequently cause shm_open() to behave as if no shared memory object of this name exists.

Upon successful completion, a value of zero shall be returned. Otherwise, a value of -1 shall be returned and errno set to indicate the error. If -1 is returned, the named shared memory object shall not be changed by this function call.

NOTES:

CHAPTER

FOURTEEN

SCHEDULER MANAGER

14.1 Introduction

The scheduler manager . . .

The directives provided by the scheduler manager are:

- *sched_get_priority_min* (page 167) - Get Minimum Priority Value
- *sched_get_priority_max* (page 167) - Get Maximum Priority Value
- *sched_rr_get_interval* (page 168) - Get Timeslicing Quantum
- *sched_yield* (page 168) - Yield the Processor

14.2 Background

14.2.1 Priority

In the RTEMS implementation of the POSIX API, the priorities range from the low priority of `sched_get_priority_min()` to the highest priority of `sched_get_priority_max()`. Numerically higher values represent higher priorities.

14.2.2 Scheduling Policies

The following scheduling policies are available:

SCHED_FIFO
Priority-based, preemptive scheduling with no timeslicing. This is equivalent to what is called "manual round-robin" scheduling.

SCHED_RR
Priority-based, preemptive scheduling with timeslicing. Time quantums are maintained on a per-thread basis and are not reset at each context switch. Thus, a thread which is preempted and subsequently resumes execution will attempt to complete the unused portion of its time quantum.

SCHED_OTHER
Priority-based, preemptive scheduling with timeslicing. Time quantums are maintained on a per-thread basis and are reset at each context switch.

SCHED_SPORADIC
Priority-based, preemptive scheduling utilizing three additional parameters: budget, replenishment period, and low priority. Under this policy, the thread is allowed to execute for "budget" amount of time before its priority is lowered to "low priority". At the end of each replenishment period, the thread resumes its initial priority and has its budget replenished.

14.3 Operations

There is currently no text in this section.

14.4 Directives

This section details the scheduler manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

14.4.1 sched_get_priority_min - Get Minimum Priority Value

CALLING SEQUENCE:

```
#include <sched.h>
int sched_get_priority_min(
    int policy
);
```

STATUS CODES:

On error, this routine returns -1 and sets `errno` to one of the following:

`EINVAL`	The indicated policy is invalid.

DESCRIPTION:

This routine return the minimum (numerically and logically lowest) priority for the specified `policy`.

NOTES:

NONE

14.4.2 sched_get_priority_max - Get Maximum Priority Value

CALLING SEQUENCE:

```
#include <sched.h>
int sched_get_priority_max(
    int policy
);
```

STATUS CODES:

On error, this routine returns -1 and sets `errno` to one of the following:

`EINVAL`	The indicated policy is invalid.

DESCRIPTION:

This routine return the maximum (numerically and logically highest) priority for the specified `policy`.

NOTES:

NONE

14.4.3 sched_rr_get_interval - Get Timeslicing Quantum

CALLING SEQUENCE:

```
#include <sched.h>
int sched_rr_get_interval(
    pid_t           pid,
    struct timespec *interval
);
```

STATUS CODES:

On error, this routine returns -1 and sets `errno` to one of the following:

`ESRCH`	The indicated process id is invalid.
`EINVAL`	The specified interval pointer parameter is invalid.

DESCRIPTION:

This routine returns the length of the timeslice quantum in the `interval` parameter for the specified `pid`.

NOTES:

The `pid` argument should be 0 to indicate the calling process.

14.4.4 sched_yield - Yield the Processor

CALLING SEQUENCE:

```
#include <sched.h>
int sched_yield( void );
```

STATUS CODES:

This routine always returns zero to indicate success.

DESCRIPTION:

This call forces the calling thread to yield the processor to another thread. Normally this is used to implement voluntary round-robin task scheduling.

NOTES:

NONE

CHAPTER

FIFTEEN

CLOCK MANAGER

15.1 Introduction

The clock manager provides services two primary classes of services. The first focuses on obtaining and setting the current date and time. The other category of services focus on allowing a thread to delay for a specific length of time.

The directives provided by the clock manager are:

- *clock_gettime* (page 173) - Obtain Time of Day
- *clock_settime* (page 173) - Set Time of Day
- *clock_getres* (page 174) - Get Clock Resolution
- *sleep* (page 174) - Delay Process Execution
- *usleep* (page 174) - Delay Process Execution in Microseconds
- *nanosleep* (page 175) - Delay with High Resolution
- *gettimeofday* (page 175) - Get the Time of Day
- *time* (page 176) - Get time in seconds

15.2 Background

There is currently no text in this section.

15.3 Operations

There is currently no text in this section.

15.4 Directives

This section details the clock manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

15.4.1 clock_gettime - Obtain Time of Day

CALLING SEQUENCE:

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

STATUS CODES:

On error, this routine returns -1 and sets `errno` to one of the following:

EINVAL	The tp pointer parameter is invalid.
EINVAL	The clock_id specified is invalid.

DESCRIPTION:

NOTES:

NONE

15.4.2 clock_settime - Set Time of Day

CALLING SEQUENCE:

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

STATUS CODES:

On error, this routine returns -1 and sets `errno` to one of the following:

EINVAL	The tp pointer parameter is invalid.
EINVAL	The clock_id specified is invalid.
EINVAL	The contents of the tp structure are invalid.

DESCRIPTION:

NOTES:

NONE

15.4.3 clock_getres - Get Clock Resolution

CALLING SEQUENCE:

```
#include <time.h>
int clock_getres(
    clockid_t        clock_id,
    struct timespec *res
);
```

STATUS CODES:

On error, this routine returns -1 and sets `errno` to one of the following:

`EINVAL`	The res pointer parameter is invalid.
`EINVAL`	The clock_id specified is invalid.

DESCRIPTION:

NOTES:

If `res` is `NULL`, then the resolution is not returned.

15.4.4 sleep - Delay Process Execution

CALLING SEQUENCE:

```
#include <unistd.h>
unsigned int sleep(
    unsigned int seconds
);
```

STATUS CODES:

This routine returns the number of unslept seconds.

DESCRIPTION:

The `sleep()` function delays the calling thread by the specified number of `seconds`.

NOTES:

This call is interruptible by a signal.

15.4.5 usleep - Delay Process Execution in Microseconds

CALLING SEQUENCE:

```
#include <time.h>
useconds_t usleep(
    useconds_t useconds
);
```

STATUS CODES:

This routine returns the number of unslept seconds.

DESCRIPTION:

The `sleep()` function delays the calling thread by the specified number of `seconds`.

The `usleep()` function suspends the calling thread from execution until either the number of microseconds specified by the `useconds` argument has elapsed or a signal is delivered to the calling thread and its action is to invoke a signal-catching function or to terminate the process.

Because of other activity, or because of the time spent in processing the call, the actual length of time the thread is blocked may be longer than the amount of time specified.

NOTES:

This call is interruptible by a signal.

The Single UNIX Specification allows this service to be implemented using the same timer as that used by the `alarm()` service. This is *NOT* the case for *RTEMS* and this call has no interaction with the `SIGALRM` signal.

15.4.6 nanosleep - Delay with High Resolution

CALLING SEQUENCE:

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

STATUS CODES:

On error, this routine returns -1 and sets `errno` to one of the following:

`EINTR`	The routine was interrupted by a signal.
`EAGAIN`	The requested sleep period specified negative seconds or nanoseconds.
`EINVAL`	The requested sleep period specified an invalid number for the nanoseconds field.

DESCRIPTION:

NOTES:

This call is interruptible by a signal.

15.4.7 gettimeofday - Get the Time of Day

CALLING SEQUENCE:

```
#include <sys/time.h>
#include <unistd.h>
int gettimeofday(
    struct timeval  *tp,
    struct timezone *tzp
);
```

STATUS CODES:

On error, this routine returns -1 and sets `errno` as appropriate.

EPERM	settimeofdat is called by someone other than the superuser.
EINVAL	Timezone (or something else) is invalid.
EFAULT	One of tv or tz pointed outside your accessible address space

DESCRIPTION:

This routine returns the current time of day in the tp structure.

NOTES:

Currently, the timezone information is not supported. The tzp argument is ignored.

15.4.8 time - Get time in seconds

CALLING SEQUENCE:

```
#include <time.h>
int time(
    time_t *tloc
);
```

STATUS CODES:

This routine returns the number of seconds since the Epoch.

DESCRIPTION:

time returns the time since 00:00:00 GMT, January 1, 1970, measured in seconds

If tloc in non null, the return value is also stored in the memory pointed to by t.

NOTES:

NONE

CHAPTER

SIXTEEN

TIMER MANAGER

16.1 Introduction

The timer manager is . . .

The services provided by the timer manager are:

- *timer_create* (page 181) - Create a Per-Process Timer
- *timer_delete* (page 181) - Delete a Per-Process Timer
- *timer_settime* (page 181) - Set Next Timer Expiration
- *timer_gettime* (page 182) - Get Time Remaining on Timer
- *timer_getoverrun* (page 182) - Get Timer Overrun Count

16.2 Background

16.3 Operations

16.4 System Calls

This section details the timer manager's services. A subsection is dedicated to each of this manager's services and describes the calling sequence, related constants, usage, and status codes.

16.4.1 timer_create - Create a Per-Process Timer

CALLING SEQUENCE:

```
#include <time.h>
#include <signal.h>
int timer_create(
    clockid_t          clock_id,
    struct sigevent *evp,
    timer_t         *timerid
);
```

STATUS CODES:

EXXX -

DESCRIPTION:

NOTES:

16.4.2 timer_delete - Delete a Per-Process Timer

CALLING SEQUENCE:

```
#include <time.h>
int timer_delete(
    timer_t timerid
);
```

STATUS CODES:

EXXX -

DESCRIPTION:

NOTES:

16.4.3 timer_settime - Set Next Timer Expiration

CALLING SEQUENCE:

```
#include <time.h>
int timer_settime(
    timer_t                 timerid,
    int                     flags,
    const struct itimerspec *value,
    struct itimerspec       *ovalue
);
```

STATUS CODES:

EXXX -

DESCRIPTION:

NOTES:

16.4.4 timer_gettime - Get Time Remaining on Timer

CALLING SEQUENCE:

```
#include <time.h>
int timer_gettime(
    timer_t            timerid,
    struct itimerspec *value
);
```

STATUS CODES:

EXXX -

DESCRIPTION:

NOTES:

16.4.5 timer_getoverrun - Get Timer Overrun Count

CALLING SEQUENCE:

```
#include <time.h>
int timer_getoverrun(
    timer_t   timerid
);
```

STATUS CODES:

EXXX -

DESCRIPTION:

NOTES:

CHAPTER

SEVENTEEN

MESSAGE PASSING MANAGER

17.1 Introduction

The message passing manager is the means to provide communication and synchronization capabilities using POSIX message queues.

The directives provided by the message passing manager are:

- *mq_open* (page 189) - Open a Message Queue
- *mq_close* (page 190) - Close a Message Queue
- *mq_unlink* (page 191) - Remove a Message Queue
- *mq_send* (page 191) - Send a Message to a Message Queue
- *mq_receive* (page 192) - Receive a Message from a Message Queue
- *mq_notify* (page 193) - Notify Process that a Message is Available
- *mq_setattr* (page 194) - Set Message Queue Attributes
- *mq_getattr* (page 194) - Get Message Queue Attributes

17.2 Background

17.2.1 Theory

Message queues are named objects that operate with readers and writers. In addition, a message queue is a priority queue of discrete messages. POSIX message queues offer a certain, basic amount of application access to, and control over, the message queue geometry that can be changed.

17.2.2 Messages

A message is a variable length buffer where information can be stored to support communication. The length of the message and the information stored in that message are user-defined and can be actual data, pointer(s), or empty. There is a maximum acceptable length for a message that is associated with each message queue.

17.2.3 Message Queues

Message queues are named objects similar to the pipes of POSIX. They are a means of communicating data between multiple processes and for passing messages among tasks and ISRs. Message queues can contain a variable number of messages from 0 to an upper limit that is user defined. The maximum length of the message can be set on a per message queue basis. Normally messages are sent and received from the message queue in FIFO order. However, messages can also be prioritized and a priority queue established for the passing of messages. Synchronization is needed when a task waits for a message to arrive at a queue. Also, a task may poll a queue for the arrival of a message.

The message queue descriptor `mqd_t` represents the message queue. It is passed as an argument to all of the message queue functions.

17.2.4 Building a Message Queue Attribute Set

The `mq_attr` structure is used to define the characteristics of the message queue.

```
struct mq_attr{
    long mq_flags;
    long mq_maxmsg;
    long mq_msgsize;
    long mq_curmsgs;
};
```

All of these attributes are set when the message queue is created using mq_open. The mq_flags field is not used in the creation of a message queue, it is only used by `mq_setattr` and `mq_getattr`. The structure `mq_attr` is passed as an argument to `mq_setattr` and `mq_getattr`.

The mq_flags contain information affecting the behavior of the message queue. The `O_NONBLOCK` `mq_flag` is the only flag that is defined. In `mq_setattr`, the `mq_flag` can be set to dynamically change the blocking and non-blocking behavior of the message queue. If the non-block flag is set then the message queue is non-blocking, and requests to send and receive messages do not block waiting for resources. For a blocking message queue, a request to send might have to wait for an empty message queue, and a request to receive might have to wait for a message to arrive on the queue. Both `mq_maxmsg` and `mq_msgsize` affect the sizing of the message queue. `mq_maxmsg` specifies how many messages the queue can hold at any one time. `mq_msgsize`

specifies the size of any one message on the queue. If either of these limits is exceeded, an error message results.

Upon return from `mq_getattr`, the `mq_curmsgs` is set according to the current state of the message queue. This specifies the number of messages currently on the queue.

17.2.5 Notification of a Message on the Queue

Every message queue has the ability to notify one (and only one) process whenever the queue's state changes from empty (0 messages) to nonempty. This means that the process does not have to block or constantly poll while it waits for a message. By calling `mq_notify`, you can attach a notification request to a message queue. When a message is received by an empty queue, if there are no processes blocked and waiting for the message, then the queue notifies the requesting process of a message arrival. There is only one signal sent by the message queue, after that the notification request is de-registered and another process can attach its notification request. After receipt of a notification, a process must re-register if it wishes to be notified again.

If there is a process blocked and waiting for the message, that process gets the message, and notification is not sent. It is also possible for another process to receive the message after the notification is sent but before the notified process has sent its receive request.

Only one process can have a notification request attached to a message queue at any one time. If another process attempts to register a notification request, it fails. You can de-register for a message queue by passing a NULL to mq_notify, this removes any notification request attached to the queue. Whenever the message queue is closed, all notification attachments are removed.

17.2.6 POSIX Interpretation Issues

There is one significant point of interpretation related to the RTEMS implementation of POSIX message queues:

> What happens to threads already blocked on a message queue when the mode of that same message queue is changed from blocking to non-blocking?

The RTEMS POSIX implementation decided to unblock all waiting tasks with an `EAGAIN` status just as if a non-blocking version of the same operation had returned unsatisfied. This case is not discussed in the POSIX standard and other implementations may have chosen alternative behaviors.

17.3 Operations

17.3.1 Opening or Creating a Message Queue

If the message queue already exists, `mq_open()` opens it, if the message queue does not exist, `mq_open()` creates it. When a message queue is created, the geometry of the message queue is contained in the attribute structure that is passed in as an argument. This includes mq_msgsize that dictates the maximum size of a single message, and the mq_maxmsg that dictates the maximum number of messages the queue can hold at one time. The blocking or non-blocking behavior of the queue can also specified.

17.3.2 Closing a Message Queue

The `mq_close()` function is used to close the connection made to a message queue that was made during mq_open. The message queue itself and the messages on the queue are persistent and remain after the queue is closed.

17.3.3 Removing a Message Queue

The `mq_unlink()` function removes the named message queue. If the message queue is not open when mq_unlink is called, then the queue is immediately eliminated. Any messages that were on the queue are lost, and the queue can not be opened again. If processes have the queue open when mq_unlink is called, the removal of the queue is delayed until the last process using the queue has finished. However, the name of the message queue is removed so that no other process can open it.

17.3.4 Sending a Message to a Message Queue

The `mq_send()` function adds the message in priority order to the message queue. Each message has an assigned a priority. The highest priority message is be at the front of the queue.

The maximum number of messages that a message queue may accept is specified at creation by the `mq_maxmsg` field of the attribute structure. If this amount is exceeded, the behavior of the process is determined according to what `oflag` was used when the message queue was opened. If the queue was opened with `O_NONBLOCK` flag set, the process does not block, and an error is returned. If the `O_NONBLOCK` flag was not set, the process does block and wait for space on the queue.

17.3.5 Receiving a Message from a Message Queue

The `mq_receive()` function is used to receive the oldest of the highest priority message(s) from the message queue specified by mqdes. The messages are received in FIFO order within the priorities. The received message's priority is stored in the location referenced by the `msg_prio`. If the `msg_prio` is a `NULL`, the priority is discarded. The message is removed and stored in an area pointed to by `msg_ptr` whose length is of `msg_len`. The `msg_len` must be at least equal to the `mq_msgsize` attribute of the message queue.

The blocking behavior of the message queue is set by `O_NONBLOCK` at `mq_open` or by setting `O_NONBLOCK` in `mq_flags` in a call to `mq_setattr`. If this is a blocking queue, the process does block and wait on an empty queue. If this a non-blocking queue, the process does not block. Upon successful completion, `mq_receive` returns the length of the selected message in bytes and the message is removed from the queue.

17.3.6 Notification of Receipt of a Message on an Empty Queue

The `mq_notify()` function registers the calling process to be notified of message arrival at an empty message queue. Every message queue has the ability to notify one (and only one) process whenever the queue's state changes from empty (0 messages) to nonempty. This means that the process does not have to block or constantly poll while it waits for a message. By calling `mq_notify`, a notification request is attached to a message queue. When a message is received by an empty queue, if there are no processes blocked and waiting for the message, then the queue notifies the requesting process of a message arrival. There is only one signal sent by the message queue, after that the notification request is de-registered and another process can attach its notification request. After receipt of a notification, a process must re-register if it wishes to be notified again.

If there is a process blocked and waiting for the message, that process gets the message, and notification is not sent. Only one process can have a notification request attached to a message queue at any one time. If another process attempts to register a notification request, it fails. You can de-register for a message queue by passing a `NULL` to `mq_notify`, this removes any notification request attached to the queue. Whenever the message queue is closed, all notification attachments are removed.

17.3.7 Setting the Attributes of a Message Queue

The `mq_setattr()` function is used to set attributes associated with the open message queue description referenced by the message queue descriptor specified by mqdes. The `*omqstat` represents the old or previous attributes. If `omqstat` is non-`NULL`, the function `mq_setattr()` stores, in the location referenced by omqstat, the previous message queue attributes and the current queue status. These values are the same as would be returned by a call to `mq_getattr()` at that point.

There is only one `mq_attr.mq_flag` that can be altered by this call. This is the flag that deals with the blocking and non-blocking behavior of the message queue. If the flag is set then the message queue is non-blocking, and requests to send or receive do not block while waiting for resources. If the flag is not set, then message send and receive may involve waiting for an empty queue or waiting for a message to arrive.

17.3.8 Getting the Attributes of a Message Queue

The `mq_getattr()` function is used to get status information and attributes of the message queue associated with the message queue descriptor. The results are returned in the mq_attr structure referenced by the mqstat argument. All of these attributes are set at create time, except the blocking/non-blocking behavior of the message queue which can be dynamically set by using mq_setattr. The attribute mq_curmsg is set to reflect the number of messages on the queue at the time that `mq_getattr` was called.

17.4 Directives

This section details the message passing manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

17.4.1 mq_open - Open a Message Queue

CALLING SEQUENCE:

```
#include <mqueue.h>
mqd_t mq_open(
    const char     *name,
    int             oflag,
    mode_t          mode,
    struct mq_attr *attr
);
```

STATUS CODES:

EACCES	Either the message queue exists and the permissions requested in oflags were denied, or the message does not exist and permission to create one is denied.
EEXIST	You tried to create a message queue that already exists.
EINVAL	An inappropriate name was given for the message queue, or the values of mq-maxmsg or mq_msgsize were less than 0.
ENOENT	The message queue does not exist, and you did not specify to create it.
EINTR	The call to mq_open was interrupted by a signal.
EMFILE	The process has too many files or message queues open. This is a process limit error.
ENFILE	The system has run out of resources to support more open message queues. This is a system error.
ENAMETO(	mq_name is too long.

DESCRIPTION:

The mq_open() function establishes the connection between a process and a message queue with a message queue descriptor. If the message queue already exists, mq_open opens it, if the message queue does not exist, mq_open creates it. Message queues can have multiple senders and receivers. If mq_open is successful, the function returns a message queue descriptor. Otherwise, the function returns a -1 and sets errno to indicate the error.

The name of the message queue is used as an argument. For the best of portability, the name of the message queue should begin with a "/" and no other "/" should be in the name. Different systems interpret the name in different ways.

The oflags contain information on how the message is opened if the queue already exists. This may be O_RDONLY for read only, O_WRONLY for write only, of O_RDWR, for read and write.

In addition, the oflags contain information needed in the creation of a message queue.

O_NONB	If the non-block flag is set then the message queue is non-blocking, and requests to send and receive messages do not block waiting for resources. If the flag is not set then the message queue is blocking, and a request to send might have to wait for an empty message queue. Similarly, a request to receive might have to wait for a message to arrive on the queue.
O_CREA	This call specifies that the call the mq_open is to create a new message queue. In this case the mode and attribute arguments of the function call are utilized. The message queue is created with a mode similar to the creation of a file, read and write permission creator, group, and others. The geometry of the message queue is contained in the attribute structure. This includes mq_msgsize that dictates the maximum size of a single message, and the mq_maxmsg that dictates the maximum number of messages the queue can hold at one time. If a NULL is used in the mq_attr argument, then the message queue is created with implementation defined defaults.
O_EXCL	is always set if O_CREAT flag is set. If the message queue already exists, O_EXCL causes an error message to be returned, otherwise, the new message queue fails and appends to the existing one.

NOTES:

The mq_open() function does not add or remove messages from the queue. When a new message queue is being created, the mq_flag field of the attribute structure is not used.

17.4.2 mq_close - Close a Message Queue

CALLING SEQUENCE:

```
#include <mqueue.h>
int mq_close(
    mqd_t mqdes
);
```

STATUS CODES:

EINVAL	The descriptor does not represent a valid open message queue

DESCRIPTION:

The mq_close function removes the association between the message queue descriptor, mqdes, and its message queue. If mq_close() is successfully completed, the function returns a value of zero; otherwise, the function returns a value of -1 and sets errno to indicate the error.

NOTES:

If the process had successfully attached a notification request to the message queue via mq_notify, this attachment is removed, and the message queue is available for another process to attach for notification. mq_close has no effect on the contents of the message queue, all the messages that were in the queue remain in the queue.

17.4.3 mq_unlink - Remove a Message Queue

CALLING SEQUENCE:

```
#include <mqueue.h>
int mq_unlink(
    const char *name
);
```

STATUS CODES:

EINVAL	The descriptor does not represent a valid message queue

DESCRIPTION:

The mq_unlink() function removes the named message queue. If the message queue is not open when mq_unlink is called, then the queue is immediately eliminated. Any messages that were on the queue are lost, and the queue can not be opened again. If processes have the queue open when mq_unlink is called, the removal of the queue is delayed until the last process using the queue has finished. However, the name of the message queue is removed so that no other process can open it. Upon successful completion, the function returns a value of zero. Otherwise, the named message queue is not changed by this function call, and the function returns a value of -1 and sets errno to indicate the error.

NOTES:

Calls to mq_open() to re-create the message queue may fail until the message queue is actually removed. However, the mq_unlink() call need not block until all references have been closed; it may return immediately.

17.4.4 mq_send - Send a Message to a Message Queue

CALLING SEQUENCE:

```
#include<mqueue.h>
int mq_send(
    mqd_t        mqdes,
    const char  *msg_ptr,
    size_t       msg_len,
    unsigned int msg_prio
);
```

STATUS CODES:

EBADF	The descriptor does not represent a valid message queue, or the queue was opened for read only O_RDONLY
EINVAL	The value of msg_prio was greater than the MQ_PRIO_MAX.
EMSGSI	The msg_len is greater than the mq_msgsize attribute of the message queue
EAGAIN	The message queue is non-blocking, and there is no room on the queue for another message as specified by the mq_maxmsg.
EINTR	The message queue is blocking. While the process was waiting for free space on the queue, a signal arrived that interrupted the wait.

DESCRIPTION:

The mq_send() function adds the message pointed to by the argument msg_ptr to the message queue specified by mqdes. Each message is assigned a priority , from 0 to MQ_PRIO_MAX. MQ_PRIO_MAX is defined in <limits.h> and must be at least 32. Messages are added to the queue in order of their priority. The highest priority message is at the front of the queue.

The maximum number of messages that a message queue may accept is specified at creation by the mq_maxmsg field of the attribute structure. If this amount is exceeded, the behavior of the process is determined according to what oflag was used when the message queue was opened. If the queue was opened with O_NONBLOCK flag set, then the EAGAIN error is returned. If the O_NONBLOCK flag was not set, the process blocks and waits for space on the queue, unless it is interrupted by a signal.

Upon successful completion, the mq_send() function returns a value of zero. Otherwise, no message is enqueued, the function returns -1, and errno is set to indicate the error.

NOTES:

If the specified message queue is not full, mq_send inserts the message at the position indicated by the msg_prio argument.

17.4.5 mq_receive - Receive a Message from a Message Queue

CALLING SEQUENCE:

```
#include <mqueue.h>
size_t mq_receive(
    mqd_t          mqdes,
    char          *msg_ptr,
    size_t         msg_len,
    unsigned int *msg_prio
);
```

STATUS CODES:

EBADF	The descriptor does not represent a valid message queue, or the queue was opened for write only O_WRONLY
EMSGSI	The msg_len is less than the mq_msgsize attribute of the message queue
EAGAIN	The message queue is non-blocking, and the queue is empty
EAGAIN	The operation would block but has been called from an ISR
EINTR	The message queue is blocking. While the process was waiting for a message to arrive on the queue, a signal arrived that interrupted the wait.

DESCRIPTION:

The mq_receive function is used to receive the oldest of the highest priority message(s) from the message queue specified by mqdes. The messages are received in FIFO order within the priorities. The received message's priority is stored in the location referenced by the msg_prio. If the msg_prio is a NULL, the priority is discarded. The message is removed and stored in an area pointed to by msg_ptr whose length is of msg_len. The msg_len must be at least equal to the mq_msgsize attribute of the message queue.

The blocking behavior of the message queue is set by O_NONBLOCK at mq_open or by setting

O_NONBLOCK in mq_flags in a call to mq_setattr. If this is a blocking queue, the process blocks and waits on an empty queue. If this a non-blocking queue, the process does not block.

Upon successful completion, mq_receive returns the length of the selected message in bytes and the message is removed from the queue. Otherwise, no message is removed from the queue, the function returns a value of -1, and sets errno to indicate the error.

NOTES:

If the size of the buffer in bytes, specified by the msg_len argument, is less than the mq_msgsize attribute of the message queue, the function fails and returns an error

17.4.6 mq_notify - Notify Process that a Message is Available

CALLING SEQUENCE:

```
#include <mqueue.h>
int mq_notify(
    mqd_t                    mqdes,
    const struct sigevent *notification
);
```

STATUS CODES:

EBADF	The descriptor does not refer to a valid message queue
EBUSY	A notification request is already attached to the queue

DESCRIPTION:

If the argument notification is not NULL, this function registers the calling process to be notified of message arrival at an empty message queue associated with the specified message queue descriptor, mqdes.

Every message queue has the ability to notify one (and only one) process whenever the queue's state changes from empty (0 messages) to nonempty. This means that the process does not have to block or constantly poll while it waits for a message. By calling mq_notify, a notification request is attached to a message queue. When a message is received by an empty queue, if there are no processes blocked and waiting for the message, then the queue notifies the requesting process of a message arrival. There is only one signal sent by the message queue, after that the notification request is de-registered and another process can attach its notification request. After receipt of a notification, a process must re-register if it wishes to be notified again.

If there is a process blocked and waiting for the message, that process gets the message, and notification is not be sent. Only one process can have a notification request attached to a message queue at any one time. If another process attempts to register a notification request, it fails. You can de-register for a message queue by passing a NULL to mq_notify; this removes any notification request attached to the queue. Whenever the message queue is closed, all notification attachments are removed.

Upon successful completion, mq_notify returns a value of zero; otherwise, the function returns a value of -1 and sets errno to indicate the error.

NOTES:

It is possible for another process to receive the message after the notification is sent but before the notified process has sent its receive request.

17.4.7 mq_setattr - Set Message Queue Attributes

CALLING SEQUENCE:

```
#include <mqueue.h>
int mq_setattr(
    mqd_t                 mqdes,
    const struct mq_attr *mqstat,
    struct mq_attr       *omqstat
);
```

STATUS CODES:

EBADF	The message queue descriptor does not refer to a valid, open queue.
EINVAL	The mq_flag value is invalid.

DESCRIPTION:

The `mq_setattr` function is used to set attributes associated with the open message queue description referenced by the message queue descriptor specified by mqdes. The `*omqstat` represents the old or previous attributes. If `omqstat` is non-`NULL`, the function `mq_setattr()` stores, in the location referenced by `omqstat`, the previous message queue attributes and the current queue status. These values are the same as would be returned by a call to `mq_getattr()` at that point.

There is only one mq_attr.mq_flag which can be altered by this call. This is the flag that deals with the blocking and non-blocking behavior of the message queue. If the flag is set then the message queue is non-blocking, and requests to send or receive do not block while waiting for resources. If the flag is not set, then message send and receive may involve waiting for an empty queue or waiting for a message to arrive.

Upon successful completion, the function returns a value of zero and the attributes of the message queue have been changed as specified. Otherwise, the message queue attributes is unchanged, and the function returns a value of -1 and sets `errno` to indicate the error.

NOTES:

All other fields in the `mq_attr` are ignored by this call.

17.4.8 mq_getattr - Get Message Queue Attributes

CALLING SEQUENCE:

```
#include <mqueue.h>
int mq_getattr(
    mqd_t           mqdes,
    struct mq_attr *mqstat
);
```

STATUS CODES:

EBADF	The message queue descriptor does not refer to a valid, open message queue.

DESCRIPTION:

The mqdes argument specifies a message queue descriptor. The mq_getattr function is used to get status information and attributes of the message queue associated with the message queue descriptor. The results are returned in the mq_attr structure referenced by the mqstat argument. All of these attributes are set at create time, except the blocking/non-blocking behavior of the message queue which can be dynamically set by using mq_setattr. The attribute mq_curmsg is set to reflect the number of messages on the queue at the time that mq_getattr was called.

Upon successful completion, the mq_getattr function returns zero. Otherwise, the function returns -1 and sets errno to indicate the error.

NOTES:

CHAPTER

EIGHTEEN

THREAD MANAGER

18.1 Introduction

The thread manager implements the functionality required of the thread manager as defined by POSIX 1003.1b. This standard requires that a compliant operating system provide the facilties to manage multiple threads of control and defines the API that must be provided.

The services provided by the thread manager are:

- *pthread_attr_init* (page 202) - Initialize a Thread Attribute Set
- *pthread_attr_destroy* (page 202) - Destroy a Thread Attribute Set
- *pthread_attr_setdetachstate* (page 203) - Set Detach State
- *pthread_attr_getdetachstate* (page 203) - Get Detach State
- *pthread_attr_setstacksize* (page 204) - Set Thread Stack Size
- *pthread_attr_getstacksize* (page 204) - Get Thread Stack Size
- *pthread_attr_setstackaddr* (page 205) - Set Thread Stack Address
- *pthread_attr_getstackaddr* (page 205) - Get Thread Stack Address
- *pthread_attr_setscope* (page 206) - Set Thread Scheduling Scope
- *pthread_attr_getscope* (page 206) - Get Thread Scheduling Scope
- *pthread_attr_setinheritsched* (page 207) - Set Inherit Scheduler Flag
- *pthread_attr_getinheritsched* (page 207) - Get Inherit Scheduler Flag
- *pthread_attr_setschedpolicy* (page 208) - Set Scheduling Policy
- *pthread_attr_getschedpolicy* (page 209) - Get Scheduling Policy
- *pthread_attr_setschedparam* (page 209) - Set Scheduling Parameters
- *pthread_attr_getschedparam* (page 210) - Get Scheduling Parameters
- *pthread_attr_getaffinity_np* (page 210) - Get Thread Affinity Attribute
- *pthread_attr_setaffinity_np* (page 211) - Set Thread Affinity Attribute
- *pthread_create* (page 211) - Create a Thread
- *pthread_exit* (page 212) - Terminate the Current Thread
- *pthread_detach* (page 213) - Detach a Thread
- *pthread_getconcurrency* (page 213) - Get Thread Level of Concurrency
- *pthread_setconcurrency* (page 214) - Set Thread Level of Concurrency
- *pthread_getattr_np* (page 214) - Get Thread Attributes
- *pthread_join* (page 215) - Wait for Thread Termination
- *pthread_self* (page 215) - Get Thread ID
- *pthread_equal* (page 215) - Compare Thread IDs
- *pthread_once* (page 216) - Dynamic Package Initialization
- *pthread_setschedparam* (page 216) - Set Thread Scheduling Parameters
- *pthread_getschedparam* (page 217) - Get Thread Scheduling Parameters

- *pthread_getaffinity_np* (page 218) - Get Thread Affinity
- *pthread_setaffinity_np* (page 218) - Set Thread Affinity

18.2 Background

18.2.1 Thread Attributes

Thread attributes are utilized only at thread creation time. A thread attribute structure may be initialized and passed as an argument to the `pthread_create` routine.

stack address
is the address of the optionally user specified stack area for this thread. If this value is NULL, then RTEMS allocates the memory for the thread stack from the RTEMS Workspace Area. Otherwise, this is the user specified address for the memory to be used for the thread's stack. Each thread must have a distinct stack area. Each processor family has different alignment rules which should be followed.

stack size
is the minimum desired size for this thread's stack area. If the size of this area as specified by the stack size attribute is smaller than the minimum for this processor family and the stack is not user specified, then RTEMS will automatically allocate a stack of the minimum size for this processor family.

contention scope
specifies the scheduling contention scope. RTEMS only supports the PTHREAD_SCOPE_PROCESS scheduling contention scope.

scheduling inheritance
specifies whether a user specified or the scheduling policy and parameters of the currently executing thread are to be used. When this is PTHREAD_INHERIT_SCHED, then the scheduling policy and parameters of the currently executing thread are inherited by the newly created thread.

scheduling policy and parameters
specify the manner in which the thread will contend for the processor. The scheduling parameters are interpreted based on the specified policy. All policies utilize the thread priority parameter.

18.3 Operations

There is currently no text in this section.

18.4 Services

This section details the thread manager's services. A subsection is dedicated to each of this manager's services and describes the calling sequence, related constants, usage, and status codes.

18.4.1 pthread_attr_init - Initialize a Thread Attribute Set

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_attr_init(
    pthread_attr_t *attr
);
```

STATUS CODES:

EINVAL	The attribute pointer argument is invalid.

DESCRIPTION:

The `pthread_attr_init` routine initializes the thread attributes object specified by `attr` with the default value for all of the individual attributes.

NOTES:

The settings in the default attributes are implementation defined. For RTEMS, the default attributes are as follows:

stackadr	is not set to indicate that RTEMS is to allocate the stack memory.
stacksize	is set to `PTHREAD_MINIMUM_STACK_SIZE`.
contention-scope	is set to `PTHREAD_SCOPE_PROCESS`.
inherit-sched	is set to `PTHREAD_INHERIT_SCHED` to indicate that the created thread inherits its scheduling attributes from its parent.
detachstate	is set to `PTHREAD_CREATE_JOINABLE`.

18.4.2 pthread_attr_destroy - Destroy a Thread Attribute Set

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_attr_destroy(
    pthread_attr_t *attr
);
```

STATUS CODES:

EINVAL	The attribute pointer argument is invalid.
EINVAL	The attribute set is not initialized.

DESCRIPTION:

The `pthread_attr_destroy` routine is used to destroy a thread attributes object. The behavior of using an attributes object after it is destroyed is implementation dependent.

NOTES:

NONE

18.4.3 pthread_attr_setdetachstate - Set Detach State

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_attr_setdetachstate(
    pthread_attr_t *attr,
    int             detachstate
);
```

STATUS CODES:

`EINVAL`	The attribute pointer argument is invalid.
`EINVAL`	The attribute set is not initialized.
`EINVAL`	The detachstate argument is invalid.

DESCRIPTION:

The `pthread_attr_setdetachstate` routine is used to value of the `detachstate` attribute. This attribute controls whether the thread is created in a detached state.

The `detachstate` can be either `PTHREAD_CREATE_DETACHED` or `PTHREAD_CREATE_JOINABLE`. The default value for all threads is `PTHREAD_CREATE_JOINABLE`.

NOTES:

If a thread is in a detached state, then the use of the ID with the `pthread_detach` or `pthread_join` routines is an error.

18.4.4 pthread_attr_getdetachstate - Get Detach State

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_attr_getdetachstate(
    const pthread_attr_t *attr,
    int                  *detachstate
);
```

STATUS CODES:

`EINVAL`	The attribute pointer argument is invalid.
`EINVAL`	The attribute set is not initialized.
`EINVAL`	The detatchstate pointer argument is invalid.

DESCRIPTION:

The `pthread_attr_getdetachstate` routine is used to obtain the current value of the `detachstate` attribute as specified by the `attr` thread attribute object.

NOTES:

NONE

18.4.5 pthread_attr_setstacksize - Set Thread Stack Size

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_attr_setstacksize(
    pthread_attr_t *attr,
    size_t          stacksize
);
```

STATUS CODES:

`EINVAL`	The attribute pointer argument is invalid.
`EINVAL`	The attribute set is not initialized.

DESCRIPTION:

The `pthread_attr_setstacksize` routine is used to set the `stacksize` attribute in the `attr` thread attribute object.

NOTES:

As required by POSIX, RTEMS defines the feature symbol `_POSIX_THREAD_ATTR_STACKSIZE` to indicate that this routine is supported.

If the specified stacksize is below the minimum required for this CPU (`PTHREAD_STACK_MIN`, then the stacksize will be set to the minimum for this CPU.

18.4.6 pthread_attr_getstacksize - Get Thread Stack Size

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_attr_getstacksize(
    const pthread_attr_t *attr,
    size_t               *stacksize
);
```

STATUS CODES:

`EINVAL`	The attribute pointer argument is invalid.
`EINVAL`	The attribute set is not initialized.
`EINVAL`	The stacksize pointer argument is invalid.

DESCRIPTION:

The pthread_attr_getstacksize routine is used to obtain the stacksize attribute in the attr thread attribute object.

NOTES:

As required by POSIX, RTEMS defines the feature symbol _POSIX_THREAD_ATTR_STACKSIZE to indicate that this routine is supported.

18.4.7 pthread_attr_setstackaddr - Set Thread Stack Address

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_attr_setstackaddr(
    pthread_attr_t *attr,
    void           *stackaddr
);
```

STATUS CODES:

EINVAL	The attribute pointer argument is invalid.
EINVAL	The attribute set is not initialized.

DESCRIPTION:

The pthread_attr_setstackaddr routine is used to set the stackaddr attribute in the attr thread attribute object.

NOTES:

As required by POSIX, RTEMS defines the feature symbol _POSIX_THREAD_ATTR_STACKADDR to indicate that this routine is supported.

It is imperative to the proper operation of the system that each thread have sufficient stack space.

18.4.8 pthread_attr_getstackaddr - Get Thread Stack Address

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_attr_getstackaddr(
    const pthread_attr_t  *attr,
    void                 **stackaddr
);
```

STATUS CODES:

EINVAL	The attribute pointer argument is invalid.
EINVAL	The attribute set is not initialized.
EINVAL	The stackaddr pointer argument is invalid.

DESCRIPTION:

The pthread_attr_getstackaddr routine is used to obtain the stackaddr attribute in the attr thread attribute object.

NOTES:

As required by POSIX, RTEMS defines the feature symbol _POSIX_THREAD_ATTR_STACKADDR to indicate that this routine is supported.

18.4.9 pthread_attr_setscope - Set Thread Scheduling Scope

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_attr_setscope(
    pthread_attr_t *attr,
    int             contentionscope
);
```

STATUS CODES:

EINVAL	The attribute pointer argument is invalid.
EINVAL	The attribute set is not initialized.
EINVAL	The contention scope specified is not valid.
ENOTSUP	The contention scope specified (PTHREAD_SCOPE_SYSTEM) is not supported.

DESCRIPTION:

The pthread_attr_setscope routine is used to set the contention scope field in the thread attribute object attr to the value specified by contentionscope.

The contentionscope must be either PTHREAD_SCOPE_SYSTEM to indicate that the thread is to be within system scheduling contention or PTHREAD_SCOPE_PROCESS indicating that the thread is to be within the process scheduling contention scope.

NOTES:

As required by POSIX, RTEMS defines the feature symbol _POSIX_THREAD_PRIORITY_SCHEDULING to indicate that the family of routines to which this routine belongs is supported.

18.4.10 pthread_attr_getscope - Get Thread Scheduling Scope

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_attr_getscope(
    const pthread_attr_t *attr,
    int                  *contentionscope
);
```

STATUS CODES:

EINVAL	The attribute pointer argument is invalid.
EINVAL	The attribute set is not initialized.
EINVAL	The contentionscope pointer argument is invalid.

DESCRIPTION:

The pthread_attr_getscope routine is used to obtain the value of the contention scope field in the thread attributes object attr. The current value is returned in contentionscope.

NOTES:

As required by POSIX, RTEMS defines the feature symbol _POSIX_THREAD_PRIORITY_SCHEDULING to indicate that the family of routines to which this routine belongs is supported.

18.4.11 pthread_attr_setinheritsched - Set Inherit Scheduler Flag

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_attr_setinheritsched(
    pthread_attr_t *attr,
    int             inheritsched
);
```

STATUS CODES:

EINVAL	The attribute pointer argument is invalid.
EINVAL	The attribute set is not initialized.
EINVAL	The specified scheduler inheritance argument is invalid.

DESCRIPTION:

The pthread_attr_setinheritsched routine is used to set the inherit scheduler field in the thread attribute object attr to the value specified by inheritsched.

The contentionscope must be either PTHREAD_INHERIT_SCHED to indicate that the thread is to inherit the scheduling policy and parameters fromthe creating thread, or PTHREAD_EXPLICIT_SCHED to indicate that the scheduling policy and parameters for this thread are to be set from the corresponding values in the attributes object. If contentionscope is PTHREAD_INHERIT_SCHED, then the scheduling attributes in the attr structure will be ignored at thread creation time.

NOTES:

As required by POSIX, RTEMS defines the feature symbol _POSIX_THREAD_PRIORITY_SCHEDULING to indicate that the family of routines to which this routine belongs is supported.

18.4.12 pthread_attr_getinheritsched - Get Inherit Scheduler Flag

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_attr_getinheritsched(
    const pthread_attr_t *attr,
    int                  *inheritsched
);
```

STATUS CODES:

EINVAL	The attribute pointer argument is invalid.
EINVAL	The attribute set is not initialized.
EINVAL	The inheritsched pointer argument is invalid.

DESCRIPTION:

The pthread_attr_getinheritsched routine is used to object the current value of the inherit scheduler field in the thread attribute object attr.

NOTES:

As required by POSIX, RTEMS defines the feature symbol _POSIX_THREAD_PRIORITY_SCHEDULING to indicate that the family of routines to which this routine belongs is supported.

18.4.13 pthread_attr_setschedpolicy - Set Scheduling Policy

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_attr_setschedpolicy(
    pthread_attr_t *attr,
    int             policy
);
```

STATUS CODES:

EINVAL	The attribute pointer argument is invalid.
EINVAL	The attribute set is not initialized.
ENOTSUP	The specified scheduler policy argument is invalid.

DESCRIPTION:

The pthread_attr_setschedpolicy routine is used to set the scheduler policy field in the thread attribute object attr to the value specified by policy.

Scheduling policies may be one of the following:

- SCHED_DEFAULT
- SCHED_FIFO
- SCHED_RR
- SCHED_SPORADIC
- SCHED_OTHER

The precise meaning of each of these is discussed elsewhere in this manual.

NOTES:

As required by POSIX, RTEMS defines the feature symbol _POSIX_THREAD_PRIORITY_SCHEDULING to indicate that the family of routines to which this routine belongs is supported.

18.4.14 pthread_attr_getschedpolicy - Get Scheduling Policy

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_attr_getschedpolicy(
    const pthread_attr_t *attr,
    int                  *policy
);
```

STATUS CODES:

EINVAL	The attribute pointer argument is invalid.
EINVAL	The attribute set is not initialized.
EINVAL	The specified scheduler policy argument pointer is invalid.

DESCRIPTION:

The pthread_attr_getschedpolicy routine is used to obtain the scheduler policy field from the thread attribute object attr. The value of this field is returned in policy.

NOTES:

As required by POSIX, RTEMS defines the feature symbol _POSIX_THREAD_PRIORITY_SCHEDULING to indicate that the family of routines to which this routine belongs is supported.

18.4.15 pthread_attr_setschedparam - Set Scheduling Parameters

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_attr_setschedparam(
    pthread_attr_t           *attr,
    const struct sched_param  param
);
```

STATUS CODES:

EINVAL	The attribute pointer argument is invalid.
EINVAL	The attribute set is not initialized.
EINVAL	The specified scheduler parameter argument is invalid.

DESCRIPTION:

The pthread_attr_setschedparam routine is used to set the scheduler parameters field in the thread attribute object attr to the value specified by param.

NOTES:

As required by POSIX, RTEMS defines the feature symbol _POSIX_THREAD_PRIORITY_SCHEDULING to indicate that the family of routines to which this routine belongs is supported.

18.4.16 pthread_attr_getschedparam - Get Scheduling Parameters

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_attr_getschedparam(
    const pthread_attr_t *attr,
    struct sched_param   *param
);
```

STATUS CODES:

EINVAL	The attribute pointer argument is invalid.
EINVAL	The attribute set is not initialized.
EINVAL	The specified scheduler parameter argument pointer is invalid.

DESCRIPTION:

The pthread_attr_getschedparam routine is used to obtain the scheduler parameters field from the thread attribute object attr. The value of this field is returned in param.

NOTES:

As required by POSIX, RTEMS defines the feature symbol _POSIX_THREAD_PRIORITY_SCHEDULING to indicate that the family of routines to which this routine belongs is supported.

18.4.17 pthread_attr_getaffinity_np - Get Thread Affinity Attribute

CALLING SEQUENCE:

```
#define _GNU_SOURCE
#include <pthread.h>
int pthread_attr_getaffinity_np(
    const pthread_attr_t *attr,
    size_t                cpusetsize,
    cpu_set_t            *cpuset
);
```

STATUS CODES:

EFAULT	The attribute pointer argument is invalid.
EFAULT	The cpuset pointer argument is invalid.
EINVAL	The cpusetsize does not match the value of affinitysetsize field in the thread attribute object.

DESCRIPTION:

The pthread_attr_getaffinity_np routine is used to obtain the affinityset field from the thread attribute object attr. The value of this field is returned in cpuset.

NOTES:

NONE

18.4.18 pthread_attr_setaffinity_np - Set Thread Affinity Attribute

CALLING SEQUENCE:

```
#define _GNU_SOURCE
#include <pthread.h>
int pthread_attr_setaffinity_np(
    pthread_attr_t    *attr,
    size_t             cpusetsize,
    const cpu_set_t   *cpuset
);
```

STATUS CODES:

EFAULT	The attribute pointer argument is invalid.
EFAULT	The cpuset pointer argument is invalid.
EINVAL	The cpusetsize does not match the value of affinitysetsize field in the thread attribute object.
EINVAL	The cpuset did not select a valid cpu.
EINVAL	The cpuset selected a cpu that was invalid.

DESCRIPTION:

The pthread_attr_setaffinity_np routine is used to set the affinityset field in the thread attribute object attr. The value of this field is returned in cpuset.

NOTES:

NONE

18.4.19 pthread_create - Create a Thread

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_create(
    pthread_t             *thread,
    const pthread_attr_t  *attr,
    void                 (*start_routine)( void *),
    void                  *arg
);
```

STATUS CODES:

EINV/	The attribute set is not initialized.
EINV/	The user specified a stack address and the size of the area was not large enough to meet this processor's minimum stack requirements.
EINV/	The specified scheduler inheritance policy was invalid.
ENOT$	The specified contention scope was PTHREAD_SCOPE_PROCESS.
EINV/	The specified thread priority was invalid.
EINV/	The specified scheduling policy was invalid.
EINV/	The scheduling policy was SCHED_SPORADIC and the specified replenishment period is less than the initial budget.
EINV/	The scheduling policy was SCHED_SPORADIC and the specified low priority is invalid.
EAGA]	The system lacked the necessary resources to create another thread, or the self imposed limit on the total number of threads in a process PTHREAD_THREAD_MAX would be exceeded.
EINV/	Invalid argument passed.

DESCRIPTION:

The pthread_create routine is used to create a new thread with the attributes specified by attr. If the attr argument is NULL, then the default attribute set will be used. Modification of the contents of attr after this thread is created does not have an impact on this thread.

The thread begins execution at the address specified by start_routine with arg as its only argument. If start_routine returns, then it is functionally equivalent to the thread executing the pthread_exit service.

Upon successful completion, the ID of the created thread is returned in the thread argument.

NOTES:

There is no concept of a single main thread in RTEMS as there is in a tradition UNIX system. POSIX requires that the implicit return of the main thread results in the same effects as if there were a call to exit. This does not occur in RTEMS.

The signal mask of the newly created thread is inherited from its creator and the set of pending signals for this thread is empty.

18.4.20 pthread_exit - Terminate the Current Thread

CALLING SEQUENCE:

```
#include <pthread.h>
void pthread_exit(
    void *status
);
```

STATUS CODES:

NONE

DESCRIPTION:

The pthread_exit routine is used to terminate the calling thread. The status is made available to any successful join with the terminating thread.

When a thread returns from its start routine, it results in an implicit call to the pthread_exit routine with the return value of the function serving as the argument to pthread_exit.

NOTES:

Any cancellation cleanup handlers that hace been pushed and not yet popped shall be popped in reverse of the order that they were pushed. After all cancellation cleanup handlers have been executed, if the thread has any thread-specific data, destructors for that data will be invoked.

Thread termination does not release or free any application visible resources including byt not limited to mutexes, file descriptors, allocated memory, etc.. Similarly, exitting a thread does not result in any process-oriented cleanup activity.

There is no concept of a single main thread in RTEMS as there is in a tradition UNIX system. POSIX requires that the implicit return of the main thread results in the same effects as if there were a call to `exit`. This does not occur in RTEMS.

All access to any automatic variables allocated by the threads is lost when the thread exits. Thus references (i.e. pointers) to local variables of a thread should not be used in a global manner without care. As a specific example, a pointer to a local variable should NOT be used as the return value.

18.4.21 pthread_detach - Detach a Thread

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_detach(
    pthread_t thread
);
```

STATUS CODES:

`ESRCH`	The thread specified is invalid.
`EINVAL`	The thread specified is not a joinable thread.

DESCRIPTION:

The `pthread_detach` routine is used to to indicate that storage for `thread` can be reclaimed when the thread terminates without another thread joinging with it.

NOTES:

If any threads have previously joined with the specified thread, then they will remain joined with that thread. Any subsequent calls to `pthread_join` on the specified thread will fail.

18.4.22 pthread_getconcurrency - Obtain Thread Concurrency

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_getconcurrency(void);
```

STATUS CODES:

This method returns the current concurrency mapping value.

DESCRIPTION:

The pthread_getconcurrency method returns the number of user threads mapped onto kernel threads. For RTEMS, user and kernel threads are mapped 1:1 and per the POSIX standard this method returns 1 initially and the value last set by pthread_setconcurrency otherwise.

NOTES:

NONE

18.4.23 pthread_setconcurrency - Set Thread Concurrency

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_setconcurrency(void);
```

STATUS CODES:

This method returns 0 on success.

DESCRIPTION:

The pthread_setconcurrency method requests the number of user threads mapped onto kernel threads. Per the POSIX standard, this is considered a request and may have no impact.

For RTEMS, user and kernel threads are always mapped 1:1 and thus this method has no change on the mapping. However, pthread_getconcurrency will return the value set.

NOTES:

NONE

18.4.24 pthread_getattr_np - Get Thread Attributes

CALLING SEQUENCE:

```
#define _GNU_SOURCE
#include <pthread.h>
int pthread_getattr_np(
    pthread_t       thread,
    pthread_attr_t *attr
);
```

STATUS CODES:

ESRCH	The thread specified is invalid.
EINVAL	The attribute pointer argument is invalid.

DESCRIPTION:

The pthread_getattr_np routine is used to obtain the attributes associated with thread.

NOTES:

Modification of the execution modes and priority through the Classic API may result in a combination that is not representable in the POSIX API.

18.4.25 pthread_join - Wait for Thread Termination

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_join(
    pthread_t    thread,
    void       **value_ptr
);
```

STATUS CODES:

ESRCH	The thread specified is invalid.
EINVAL	The thread specified is not a joinable thread.
EDEADLK	A deadlock was detected or thread is the calling thread.

DESCRIPTION:

The `pthread_join` routine suspends execution of the calling thread until `thread` terminates. If `thread` has already terminated, then this routine returns immediately. The value returned by `thread` (i.e. passed to `pthread_exit` is returned in `value_ptr`.

When this routine returns, then `thread` has been terminated.

NOTES:

The results of multiple simultaneous joins on the same thread is undefined.

If any threads have previously joined with the specified thread, then they will remain joined with that thread. Any subsequent calls to `pthread_join` on the specified thread will fail.

If value_ptr is NULL, then no value is returned.

18.4.26 pthread_self - Get Thread ID

CALLING SEQUENCE:

```
#include <pthread.h>
pthread_t pthread_self( void );
```

STATUS CODES:

The value returned is the ID of the calling thread.

DESCRIPTION:

This routine returns the ID of the calling thread.

NOTES:

NONE

18.4.27 pthread_equal - Compare Thread IDs

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_equal(
    pthread_t t1,
    pthread_t t2
);
```

STATUS CODES:

zero	The thread ids are not equal.
non-zero	The thread ids are equal.

DESCRIPTION:

The pthread_equal routine is used to compare two thread IDs and determine if they are equal.

NOTES:

The behavior is undefined if the thread IDs are not valid.

18.4.28 pthread_once - Dynamic Package Initialization

CALLING SEQUENCE:

```
#include <pthread.h>
pthread_once_t once_control = PTHREAD_ONCE_INIT;
int pthread_once(
    pthread_once_t   *once_control,
    void            (*init_routine)(void)
);
```

STATUS CODES:

NONE

DESCRIPTION:

The pthread_once routine is used to provide controlled initialization of variables. The first call to pthread_once by any thread with the same once_control will result in the init_routine being invoked with no arguments. Subsequent calls to pthread_once with the same once_control will have no effect.

The init_routine is guaranteed to have run to completion when this routine returns to the caller.

NOTES:

The behavior of pthread_once is undefined if once_control is automatic storage (i.e. on a task stack) or is not initialized using PTHREAD_ONCE_INIT.

18.4.29 pthread_setschedparam - Set Thread Scheduling Parameters

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_setschedparam(
    pthread_t           thread,
    int                 policy,
    struct sched_param *param
);
```

STATUS CODES:

EINVAL	The scheduling parameters indicated by the parameter param is invalid.
EINVAL	The value specified by policy is invalid.
EINVAL	The scheduling policy was SCHED_SPORADIC and the specified replenishment period is less than the initial budget.
EINVAL	The scheduling policy was SCHED_SPORADIC and the specified low priority is invalid.
ESRCH	The thread indicated was invalid.

DESCRIPTION:

The pthread_setschedparam routine is used to set the scheduler parameters currently associated with the thread specified by thread to the policy specified by policy. The contents of param are interpreted based upon the policy argument.

NOTES:

As required by POSIX, RTEMS defines the feature symbol _POSIX_THREAD_PRIORITY_SCHEDULING to indicate that the family of routines to which this routine belongs is supported.

18.4.30 pthread_getschedparam - Get Thread Scheduling Parameters

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_getschedparam(
    pthread_t           thread,
    int                *policy,
    struct sched_param *param
);
```

STATUS CODES:

EINVAL	The policy pointer argument is invalid.
EINVAL	The scheduling parameters pointer argument is invalid.
ESRCH	The thread indicated by the parameter thread is invalid.

DESCRIPTION:

The pthread_getschedparam routine is used to obtain the scheduler policy and parameters associated with thread. The current policy and associated parameters values returned in``policy`` and param, respectively.

NOTES:

As required by POSIX, RTEMS defines the feature symbol _POSIX_THREAD_PRIORITY_SCHEDULING to indicate that the family of routines to which this routine belongs is supported.

18.4.31 pthread_getaffinity_np - Get Thread Affinity

CALLING SEQUENCE:

```
#define _GNU_SOURCE
#include <pthread.h>
int pthread_getaffinity_np(
    const pthread_t       id,
    size_t                cpusetsize,
    cpu_set_t            *cpuset
);
```

STATUS CODES:

EFAULT	The cpuset pointer argument is invalid.
EINVAL	The cpusetsize does not match the value of affinitysetsize field in the thread attribute object.

DESCRIPTION:

The pthread_getaffinity_np routine is used to obtain the affinity.set field from the thread control object associated with the id. The value of this field is returned in cpuset.

NOTES:

NONE

18.4.32 pthread_setaffinity_np - Set Thread Affinity

CALLING SEQUENCE:

```
#define _GNU_SOURCE
#include <pthread.h>
int pthread_setaffinity_np(
    pthread_t          id,
    size_t             cpusetsize,
    const cpu_set_t   *cpuset
);
```

STATUS CODES:

EFAULT	The cpuset pointer argument is invalid.
EINVAL	The cpusetsize does not match the value of affinitysetsize field in the thread attribute object.
EINVAL	The cpuset did not select a valid cpu.
EINVAL	The cpuset selected a cpu that was invalid.

DESCRIPTION:

The `pthread_setaffinity_np` routine is used to set the `affinityset` field of the thread object `id`. The value of this field is returned in `cpuset`

NOTES:

NONE

CHAPTER

NINETEEN

KEY MANAGER

19.1 Introduction

The key manager allows for the creation and deletion of Data keys specific to threads.

The directives provided by the key manager are:

- *pthread_key_create* (page 225) - Create Thread Specific Data Key
- *pthread_key_delete* (page 225) - Delete Thread Specific Data Key
- *pthread_setspecific* (page 226) - Set Thread Specific Key Value
- *pthread_getspecific* (page 226) - Get Thread Specific Key Value

19.2 Background

There is currently no text in this section.

19.3 Operations

There is currently no text in this section.

19.4 Directives

This section details the key manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

19.4.1 pthread_key_create - Create Thread Specific Data Key

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_key_create(
    pthread_key_t *key,
    void (*destructor)( void )
);
```

STATUS CODES:

EAGAIN	There were not enough resources available to create another key.
ENOMEM	Insufficient memory exists to create the key.

DESCRIPTION

The `pthread_key_create()` function shall create a thread-specific data key visible to all threads in the process. Key values provided by `pthread_key_create()` are opaque objects used to locate thread-specific data. Although the same key value may be used by different threads, the values bound to the key by `pthread_setspecific()` are maintained on a per-thread basis and persist for the life of the calling thread.

Upon key creation, the value `NULL` shall be associated with the new key in all active threads. Upon thread creation, the value `NULL` shall be associated with all defined keys in the new thread.

NOTES

An optional destructor function may be associated with each key value. At thread exit, if a key value has a non-`NULL` destructor pointer, and the thread has a non-`NULL` value associated with that key, the value of the key is set to NULL, and then the function pointed to is called with the previously associated value as its sole argument. The order of destructor calls is unspecified if more than one destructor exists for a thread when it exits.

19.4.2 pthread_key_delete - Delete Thread Specific Data Key

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_key_delete(
    pthread_key_t key
);
```

STATUS CODES:

EINVAL	The key was invalid

DESCRIPTION:

The `pthread_key_delete()` function shall delete a thread-specific data key previously returned by `pthread_key_create()`. The thread-specific data values associated with key need not be NULL at the time `pthread_key_delete()` is called. It is the responsibility of the application to free any application storage or perform any cleanup actions for data structures related to the deleted key or associated thread-specific data in any threads; this cleanup can be done either before or after `pthread_key_delete()` is called. Any attempt to use key following the call to `pthread_key_delete()` results in undefined behavior.

NOTES:

The `pthread_key_delete()` function shall be callable from within destructor functions. No destructor functions shall be invoked by `pthread_key_delete()`. Any destructor function that may have been associated with key shall no longer be called upon thread exit.

19.4.3 pthread_setspecific - Set Thread Specific Key Value

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_setspecific(
    pthread_key_t  key,
    const void    *value
);
```

STATUS CODES:

`EINVAL`	The specified key is invalid.

DESCRIPTION:

The `pthread_setspecific()` function shall associate a thread-specific value with a key obtained via a previous call to `pthread_key_create()`. Different threads may bind different values to the same key. These values are typically pointers to blocks of dynamically allocated memory that have been reserved for use by the calling thread.

NOTES:

The effect of calling `pthread_setspecific()` with a key value not obtained from `pthread_key_create()` or after key has been deleted with `pthread_key_delete()` is undefined.

`pthread_setspecific()` may be called from a thread-specific data destructor function. Calling `pthread_setspecific()` from a thread-specific data destructor routine may result either in lost storage (after at least `PTHREAD_DESTRUCTOR_ITERATIONS` attempts at destruction) or in an infinite loop.

19.4.4 pthread_getspecific - Get Thread Specific Key Value

CALLING SEQUENCE:

```
#include <pthread.h>
void *pthread_getspecific(
    pthread_key_t key
);
```

STATUS CODES:

NULL	There is no thread-specific data associated with the specified key.
non-NULL	The data associated with the specified key.

DESCRIPTION:

The pthread_getspecific() function shall return the value currently bound to the specified key on behalf of the calling thread.

NOTES:

The effect of calling pthread_getspecific() with a key value not obtained from pthread_key_create() or after key has been deleted with pthread_key_delete() is undefined.

pthread_getspecific() may be called from a thread-specific data destructor function. A call to pthread_getspecific() for the thread-specific data key being destroyed shall return the value NULL, unless the value is changed (after the destructor starts) by a call to pthread_setspecific().

CHAPTER

TWENTY

THREAD CANCELLATION MANAGER

20.1 Introduction

The thread cancellation manager is . . .

The directives provided by the thread cancellation manager are:

- *pthread_cancel* (page 233) - Cancel Execution of a Thread
- *pthread_setcancelstate* (page 233) - Set Cancelability State
- *pthread_setcanceltype* (page 233) - Set Cancelability Type
- *pthread_testcancel* (page 234) - Create Cancellation Point
- *pthread_cleanup_push* (page 234) - Establish Cancellation Handler
- *pthread_cleanup_pop* (page 234) - Remove Cancellation Handler

20.2 Background

There is currently no text in this section.

20.3 Operations

There is currently no text in this section.

20.4 Directives

This section details the thread cancellation manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

20.4.1 pthread_cancel - Cancel Execution of a Thread

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_cancel(
    pthread_t thread
);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

20.4.2 pthread_setcancelstate - Set Cancelability State

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_setcancelstate(
    int state,
    int *oldstate
);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

20.4.3 pthread_setcanceltype - Set Cancelability Type

CALLING SEQUENCE:

```
#include <pthread.h>
int pthread_setcanceltype(
    int type,
    int *oldtype
);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

20.4.4 pthread_testcancel - Create Cancellation Point

CALLING SEQUENCE:

```
#include <pthread.h>
void pthread_testcancel(
    void
);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

20.4.5 pthread_cleanup_push - Establish Cancellation Handler

CALLING SEQUENCE:

```
#include <pthread.h>
void pthread_cleanup_push(
    void (*routine)(void*),
    void *arg
);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

20.4.6 pthread_cleanup_pop - Remove Cancellation Handler

CALLING SEQUENCE:

```
#include <pthread.h>
void pthread_cleanup_pop(
    int execute
);
```

STATUS CODES:

E	The

DESCRIPTION:

NOTES:

CHAPTER

TWENTYONE

SERVICES PROVIDED BY C LIBRARY (LIBC)

21.1 Introduction

This section lists the routines that provided by the Newlib C Library.

21.2 Standard Utility Functions (stdlib.h)

- `abort` - Abnormal termination of a program
- `abs` - Integer absolute value (magnitude)
- `assert` - Macro for Debugging Diagnostics
- `atexit` - Request execution of functions at program exit
- `atof` - String to double or float
- `atoi` - String to integer
- `bsearch` - Binary search
- `calloc` - Allocate space for arrays
- `div` - Divide two integers
- `ecvtbuf` - Double or float to string of digits
- `ecvt` - Double or float to string of digits (malloc result)
- `__env_lock` - Lock environment list for getenv and setenv
- `gvcvt` - Format double or float as string
- `exit` - End program execution
- `getenv` - Look up environment variable
- `labs` - Long integer absolute value (magnitude)
- `ldiv` - Divide two long integers
- `malloc` - Allocate memory
- `realloc` - Reallocate memory
- `free` - Free previously allocated memory
- `mallinfo` - Get information about allocated memory
- `__malloc_lock` - Lock memory pool for malloc and free
- `mbstowcs` - Minimal multibyte string to wide string converter
- `mblen` - Minimal multibyte length
- `mbtowc` - Minimal multibyte to wide character converter
- `qsort` - Sort an array
- `rand` - Pseudo-random numbers
- `strtod` - String to double or float
- `strtol` - String to long
- `strtoul` - String to unsigned long
- `system` - Execute command string
- `wcstombs` - Minimal wide string to multibyte string converter
- `wctomb` - Minimal wide character to multibyte converter

21.3 Character Type Macros and Functions (ctype.h)

- `isalnum` - Alphanumeric character predicate
- `isalpha` - Alphabetic character predicate
- `isascii` - ASCII character predicate
- `iscntrl` - Control character predicate
- `isdigit` - Decimal digit predicate
- `islower` - Lower-case character predicate
- `isprint` - Printable character predicates (isprint, isgraph)
- `ispunct` - Punctuation character predicate
- `isspace` - Whitespace character predicate
- `isupper` - Uppercase character predicate
- `isxdigit` - Hexadecimal digit predicate
- `toascii` - Force integers to ASCII range
- `tolower` - Translate characters to lower case
- `toupper` - Translate characters to upper case

21.4 Input and Output (stdio.h)

- `clearerr` - Clear file or stream error indicator
- `fclose` - Close a file
- `feof` - Test for end of file
- `ferror` - Test whether read/write error has occurred
- `fflush` - Flush buffered file output
- `fgetc` - Get a character from a file or stream
- `fgetpos` - Record position in a stream or file
- `fgets` - Get character string from a file or stream
- `fiprintf` - Write formatted output to file (integer only)
- `fopen` - Open a file
- `fdopen` - Turn an open file into a stream
- `fputc` - Write a character on a stream or file
- `fputs` - Write a character string in a file or stream
- `fread` - Read array elements from a file
- `freopen` - Open a file using an existing file descriptor
- `fseek` - Set file position
- `fsetpos` - Restore position of a stream or file
- `ftell` - Return position in a stream or file
- `fwrite` - Write array elements from memory to a file or stream
- `getc` - Get a character from a file or stream (macro)
- `getchar` - Get a character from standard input (macro)
- `gets` - Get character string from standard input (obsolete)
- `iprintf` - Write formatted output (integer only)
- `mktemp` - Generate unused file name
- `perror` - Print an error message on standard error
- `putc` - Write a character on a stream or file (macro)
- `putchar` - Write a character on standard output (macro)
- `puts` - Write a character string on standard output
- `remove` - Delete a file's name
- `rename` - Rename a file
- `rewind` - Reinitialize a file or stream
- `setbuf` - Specify full buffering for a file or stream
- `setvbuf` - Specify buffering for a file or stream

- `siprintf` - Write formatted output (integer only)
- `printf` - Write formatted output
- `scanf` - Scan and format input
- `tmpfile` - Create a temporary file
- `tmpnam` - Generate name for a temporary file
- `vprintf` - Format variable argument list

21.5 Strings and Memory (string.h)

- `bcmp` - Compare two memory areas
- `bcopy` - Copy memory regions
- `bzero` - Initialize memory to zero
- `index` - Search for character in string
- `memchr` - Find character in memory
- `memcmp` - Compare two memory areas
- `memcpy` - Copy memory regions
- `memmove` - Move possibly overlapping memory
- `memset` - Set an area of memory
- `rindex` - Reverse search for character in string
- `strcasecmp` - Compare strings ignoring case
- `strcat` - Concatenate strings
- `strchr` - Search for character in string
- `strcmp` - Character string compare
- `strcoll` - Locale specific character string compare
- `strcpy` - Copy string
- `strcspn` - Count chars not in string
- `strerror` - Convert error number to string
- `strlen` - Character string length
- `strlwr` - Convert string to lower case
- `strncasecmp` - Compare strings ignoring case
- `strncat` - Concatenate strings
- `strncmp` - Character string compare
- `strncpy` - Counted copy string
- `strpbrk` - Find chars in string
- `strrchr` - Reverse search for character in string
- `strspn` - Find initial match
- `strstr` - Find string segment
- `strtok` - Get next token from a string
- `strupr` - Convert string to upper case
- `strxfrm` - Transform string

21.6 Signal Handling (signal.h)

- `raise` - Send a signal
- `signal` - Specify handler subroutine for a signal

21.7 Time Functions (time.h)

- `asctime` - Format time as string
- `clock` - Cumulative processor time
- `ctime` - Convert time to local and format as string
- `difftime` - Subtract two times
- `gmtime` - Convert time to UTC (GMT) traditional representation
- `localtime` - Convert time to local representation
- `mktime` - Convert time to arithmetic representation
- `strftime` - Flexible calendar time formatter
- `time` - Get current calendar time (as single number)

21.8 Locale (locale.h)

- `setlocale` - Select or query locale

21.9 Reentrant Versions of Functions

- Equivalent for errno variable: - `errno_r` - XXX
- Locale functions:
  - `localeconv_r` - XXX
  - `setlocale_r` - XXX
- Equivalents for stdio variables:
  - `stdin_r` - XXX
  - `stdout_r` - XXX
  - `stderr_r` - XXX
- Stdio functions:
  - `fdopen_r` - XXX
  - `perror_r` - XXX
  - `tempnam_r` - XXX
  - `fopen_r` - XXX
  - `putchar_r` - XXX
  - `tmpnam_r` - XXX
  - `getchar_r` - XXX
  - `puts_r` - XXX
  - `tmpfile_r` - XXX
  - `gets_r` - XXX
  - `remove_r` - XXX
  - `vfprintf_r` - XXX
  - `iprintf_r` - XXX
  - `rename_r` - XXX
  - `vsnprintf_r` - XXX
  - `mkstemp_r` - XXX
  - `snprintf_r` - XXX
  - `vsprintf_r` - XXX
  - `mktemp_t` - XXX
  - `sprintf_r` - XXX
- Signal functions:
  - `init_signal_r` - XXX
  - `signal_r` - XXX
  - `kill_r` - XXX

  - _sigtramp_r - XXX
  - raise_r - XXX
- Stdlib functions:
  - calloc_r - XXX
  - mblen_r - XXX
  - srand_r - XXX
  - dtoa_r - XXX
  - mbstowcs_r - XXX
  - strtod_r - XXX
  - free_r - XXX
  - mbtowc_r - XXX
  - strtol_r - XXX
  - getenv_r - XXX
  - memalign_r - XXX
  - strtoul_r - XXX
  - mallinfo_r - XXX
  - mstats_r - XXX
  - system_r - XXX
  - malloc_r - XXX
  - rand_r - XXX
  - wcstombs_r - XXX
  - malloc_r - XXX
  - realloc_r - XXX
  - wctomb_r - XXX
  - malloc_stats_r - XXX
  - setenv_r - XXX
- String functions:
  - strtok_r - XXX
- System functions:
  - close_r - XXX
  - link_r - XXX
  - unlink_r - XXX
  - execve_r - XXX
  - lseek_r - XXX

  - `wait_r` - XXX
  - `fcntl_r` - XXX
  - `open_r` - XXX
  - `write_r` - XXX
  - `fork_r` - XXX
  - `read_r` - XXX
  - `fstat_r` - XXX
  - `sbrk_r` - XXX
  - `gettimeofday_r` - XXX
  - `stat_r` - XXX
  - `getpid_r` - XXX
  - `times_r` - XXX
- Time function:
  - `asctime_r` - XXX

21.10 Miscellaneous Macros and Functions

- `unctrl` - Return printable representation of a character

21.11 Variable Argument Lists

- Stdarg (stdarg.h):
  - `va_start` - XXX
  - `va_arg` - XXX
  - `va_end` - XXX
- Vararg (varargs.h):
  - `va_alist` - XXX
  - `va_start-trad` - XXX
  - `va_arg-trad` - XXX
  - `va_end-trad` - XXX

21.12 Reentrant System Calls

- `open_r` - XXX
- `close_r` - XXX
- `lseek_r` - XXX
- `read_r` - XXX
- `write_r` - XXX
- `fork_r` - XXX
- `wait_r` - XXX
- `stat_r` - XXX
- `fstat_r` - XXX
- `link_r` - XXX
- `unlink_r` - XXX
- `sbrk_r` - XXX

CHAPTER

TWENTYTWO

SERVICES PROVIDED BY THE MATH LIBRARY (LIBM)

22.1 Introduction

This section lists the routines that provided by the Newlib Math Library (libm).

22.2 Standard Math Functions (math.h)

- `acos` - Arccosine
- `acosh` - Inverse hyperbolic cosine
- `asin` - Arcsine
- `asinh` - Inverse hyperbolic sine
- `atan` - Arctangent
- `atan2` - Arctangent of y/x
- `atanh` - Inverse hyperbolic tangent
- `jN` - Bessel functions (jN and yN)
- `cbrt` - Cube root
- `copysign` - Sign of Y and magnitude of X
- `cosh` - Hyperbolic cosine
- `erf` - Error function (erf and erfc)
- `exp` - Exponential
- `expm1` - Exponential of x and - 1
- `fabs` - Absolute value (magnitude)
- `floor` - Floor and ceiling (floor and ceil)
- `fmod` - Floating-point remainder (modulo)
- `frexp` - Split floating-point number
- `gamma` - Logarithmic gamma function
- `hypot` - Distance from origin
- `ilogb` - Get exponent
- `infinity` - Floating infinity
- `isnan` - Check type of number
- `ldexp` - Load exponent
- `log` - Natural logarithms
- `log10` - Base 10 logarithms
- `log1p` - Log of 1 + X
- `matherr` - Modifiable math error handler
- `modf` - Split fractional and integer parts
- `nan` - Floating Not a Number
- `nextafter` - Get next representable number
- `pow` - X to the power Y
- `remainder` - remainder of X divided by Y

- `scalbn` - scalbn
- `sin` - Sine or cosine (sin and cos)
- `sinh` - Hyperbolic sine
- `sqrt` - Positive square root
- `tan` - Tangent
- `tanh` - Hyperbolic tangent

CHAPTER

TWENTYTHREE

DEVICE CONTROL

23.1 Introduction

The POSIX Device Control API is defined by POSIX 1003.26 and attempts to provides a portable alternative to the ioctl() service which is not standardized across POSIX implementations. Support for this standard is required by the Open Group's FACE Technical Standard :cits:"FACE:2012:FTS". Unfortunately, this part of the POSIX standard is not widely implemented.

The services provided by the timer manager are:

- **posix_devctl_** - Control a Device

23.2 Background

23.3 Operations

23.4 System Calls

This section details the POSIX device control's services. A subsection is dedicated to each of this manager's services and describes the calling sequence, related constants, usage, and status codes.

23.4.1 posix_devctl - Control a Device

CALLING SEQUENCE:

```
#include <devctl.h>
int posix_devctl(
  int            fd,
  int            dcmd,
  void *restrict dev_data_ptr,
  size_t         nbyte,
  int *restrict  dev_info_ptr
);
```

STATUS CODES:

The status codes returned reflect those returned by the `ioctl()` service and the underlying device drivers.

DESCRIPTION:

This method is intended to be a portable alternative to the `ioctl()` method. The RTEMS implementation follows what is referred to as a library implementation which is a simple wrapper for the `ioctl()` method. The fd, fcmd, dev_data_ptr, and nbyte parameters are passed unmodified to the `ioctl()` method.

If the dev_info_ptr parameter is not NULL, then the location pointed to by dev_info_ptr is set to 0.

NOTES:

NONE

CHAPTER

TWENTYFOUR

STATUS OF IMPLEMENTATION

This chapter provides an overview of the status of the implementation of the POSIX API for RTEMS. The *POSIX 1003.1b Compliance Guide* provides more detailed information regarding the implementation of each of the numerous functions, constants, and macros specified by the POSIX 1003.1b standard.

RTEMS supports many of the process and user/group oriented services in a "single user/single process" manner. This means that although these services may be of limited usefulness or functionality, they are provided and do work in a coherent manner. This is significant when porting existing code from UNIX to RTEMS.

- Implementation
  - The current implementation of `dup()` is insufficient.
  - FIFOs `mkfifo()` are not currently implemented.
  - Asynchronous IO is not implemented.
  - The `flockfile()` family is not implemented
  - getc/putc unlocked family is not implemented
  - Mapped Memory is partially implemented
  - NOTES:
    - For Shared Memory and Mapped Memory services, it is unclear what level of support is appropriate and possible for RTEMS.
- Functional Testing
  - Tests for unimplemented services
- Performance Testing
  - There are no POSIX Performance Tests.
- Documentation
  - Many of the service description pages are not complete in this manual. These need to be completed and information added to the background and operations sections.
  - Example programs (not just tests) would be very nice.

CHAPTER

TWENTYFIVE

COMMAND AND VARIABLE INDEX

INDEX

E

F

G

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y