New Chapters

Edition 1, for RTEMS 4.5.0

6 September 2000

On-Line Applications Research Corporation

On-Line Applications Research Corporation
TEXinfo 1999-04-25.15

Any inquiries concerning RTEMS, its related support components, or its documentation should be directed to either:

```
On-Line Applications Research Corporation
4910-L Corporate Drive
Huntsville, AL 35805
VOICE: (256) 722-9985
FAX:   (256) 722-0985
EMAIL: rtems@OARcorp.com
```

1 Event Logging Manager

1.1 Introduction

The event logging manager provides a portable method for logging system and application events and subsequent processing of those events. The capabilities in this manager were defined in the POSIX 1003.1h/D3 proposed standard titled **Services for Reliable, Available, and Serviceable Systems**.

The directives provided by the event logging manager are:

- `log_create` - Create a log file
- `log_sys_create` - Create a system log file
- `log_write` - Write to the system Log
- `log_write_any` - Write to any log file
- `log_write_entry` - Write entry to any log file
- `log_open` - Open a log file
- `log_read` - Read from a log file
- `log_notify` - Notify Process of writes to the system log
- `log_close` - Close log descriptor
- `log_seek` - Reposition log file offset
- `log_severity_before` - Compare event record severities
- `log_facilityemptyset` - Manipulate log facility sets
- `log_facilityfillset` - Manipulate log facility sets
- `log_facilityaddset` - Manipulate log facility sets
- `log_facilitydelset` - Manipulate log facility sets
- `log_facilityismember` - Manipulate log facility sets
- `log_facilityisvalid` - Manipulate log facility sets

1.2 Background

1.2.1 Log Files and Events

The operating system uses a special log file named `syslog`. This log file is called the system log and is automatically created and tracked by the operating system. The system log is written with the `log_write()` function. An alternative log file may be written using the `log_write_any()` function. It is possible to use `log_read()` to query the system log and and write the records to a non-system log file using `log_write_entry()` to produce a filtered version of the system log. For example you could produce a log of all disk controller faults that have occurred.

A non-system log may be a special log file created by an application to describe application faults, or a subset of the system log created by the application.

1.2.2 Facilities

A facility is an identification code for a subsystem, device, or other object about which information is being written to a log file.

A facility set is a collection of facilities.

1.2.3 Severity

Severity is a rating of the error that is being logged.

1.2.4 Queries

The facility identifier and the event severity are the basis for subsequent log query. A log query is used as a filter to obtain a subset of a given log file. The log file may be configured to send out an event.

1.3 Operations

1.3.1 Creating and Writing a non-System Log

The following code fragment create a non-System log file at /temp/. A real filename previously read entry and buffer `log_buf` of size `readsize` are written into the log. See the discussion on opening and reading a log for how the entry is created.

```
#include <evlog.h>
   :
  logd_t           *outlog = NULL;
  char             *path   = "/temp/";

  log_create( outlog, path );
   :
  log_write_entry( outlog, &entry, log_buf, readsize );
```

1.3.2 Reading a Log

Discuss opening and reading from a log.

```
build a query
log_open
log_read loop
```

1.4 Directives

This section details the event logging 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.

1.4.1 log_write - Write to the system Log

CALLING SEQUENCE:

```
#include <evlog.h>

int log_write(
  const log_facility_t  facility,
  const int             event_id,
  const log_severity_t  severity,
  const void           *buf,
  const size_t          len
);
```

STATUS CODES:

A successful call to `log_write()` returns a value of zero and an unsuccessful call returns the `errno`.

E2BIG This error indicates an inconsistency in the implementation. Report this as a bug.

EINVAL The `facility` argument is not a valid log facility.

EINVAL The `severity` argument exceeds `LOG_SEVERITY_MAX`.

EINVAL The `len` argument exceeds `LOG_MAXIUM_BUFFER_SIZE`.

EINVAL The `len` argument was non-zero and `buf` is `NULL`.

ENOSPC The device which contains the log file has run out of space.

EIO An I/O error occurred in writing to the log file.

DESCRIPTION:

The `log_write` function writes an event record to the system log file. The event record written consists of the event attributes specified by the `facility`, `event_id`, and `severity` arguments as well as the data identified by the `buf` and `len` arguments. The fields of the event record structure to be written are filled in as follows:

log_recid This is set to a monotonically increasing log record id maintained by the system for this individual log file.

log_size This is set to the value of the `len` argument.

log_event_id This is set to the value of the `event_id` argument.

log_facility This is set to the value of the `facility` argument.

log_severity This is set to the value of the `severity` argument.

log_uid This is set to the value returned by `geteuid()`.

log_gid This is set to the value returned by `getegid()`.

log_pid This is set to the value returned by `getpid()`.

log_pgrp This is set to the value returned by `getpgrp()`.

log_time This is set to the value returned by `clock_gettime()` for the `CLOCK_REALTIME clock` source.

NOTES:

The `_POSIX_LOGGING` feature flag is defined to indicate this service is available.

This implementation can not return the `EPERM` error.

1.4.2 log_write_any - Write to the any log file

CALLING SEQUENCE:

```
#include <evlog.h>

int log_write_any(
  const logd_t          logdes,
  const log_facility_t  facility,
  const int             event_id,
  const log_severity_t  severity,
  const void           *buf,
  const size_t          len
);
```

STATUS CODES:

A successful call to `log_write_any()` returns a value of zero and an unsuccessful call returns the `errno`.

E2BIG This error indicates an inconsistency in the implementation. Report this as a bug.

EBADF The `logdes` argument is not a valid log descriptor.

EINVAL The `facility` argument is not a valid log facility.

EINVAL The `severity` argument exceeds `LOG_SEVERITY_MAX`.

EINVAL The `len` argument exceeds `LOG_MAXIMUM_BUFFER_SIZE`.

EINVAL The `len` argument was non-zero and `buf` is `NULL`.

ENOSPC The device which contains the log file has run out of space.

EIO An I/O error occurred in writing to the log file.

DESCRIPTION:

The `log_write_any()` function writes an event record to the log file specified by `logdes`. The event record written consists of the event attributes specified by the `facility`, `event_id`, and `severity` arguments as well as the data identified by the `buf` and `len` arguments. The fields of the event record structure to be written are filled in as follows:

log_recid This is set to a monotonically increasing log record id maintained by the system for this individual log file.

log_size This is set to the value of the `len` argument.

log_event_id This is set to the value of the `event_id` argument.

log_facility This is set to the value of the `facility` argument.

log_severity This is set to the value of the `severity` argument.

log_uid This is set to the value returned by `geteuid()`.

log_gid This is set to the value returned by `getegid()`.

log_pid This is set to the value returned by `getpid()`.

log_pgrp This is set to the value returned by `getpgrp()`.

log_time This is set to the value returned by `clock_gettime()` for the `CLOCK_REALTIME` clock source.

NOTES:

The `_POSIX_LOGGING` feature flag is defined to indicate this service is available.

This implementation can not return the `EPERM` error.

This function is not defined in the POSIX specification. It is an extension provided by this implementation.

1.4.3 log_write_entry - Write entry to any log file

CALLING SEQUENCE:

```
#include <evlog.h>

int log_write_entry(
  const logd_t      logdes,
  struct log_entry *entry,
  const void       *buf,
  const size_t      len
);
```

STATUS CODES:

A successful call to `log_write_entry()` returns a value of zero and an unsuccessful call returns the `errno`.

E2BIG This error indicates an inconsistency in the implementation. Report this as a bug.

EBADF The `logdes` argument is not a valid log descriptor.

EFAULT The `entry` argument is not a valid pointer to a log entry.

EINVAL The `facility` field in `entry` is not a valid log facility.

EINVAL The `severity` field in `entry` exceeds `LOG_SEVERITY_MAX`.

EINVAL The `len` argument exceeds `LOG_MAXIMUM_BUFFER_SIZE`.

EINVAL The `len` argument was non-zero and `buf` is NULL.

ENOSPC The device which contains the log file has run out of space.

EIO An I/O error occurred in writing to the log file.

DESCRIPTION:

The `log_write_entry()` function writes an event record specified by the `entry`, `buf`, and `len` arguments. Most of the fields of the event record pointed to by `entry` are left intact. The following fields are filled in as follows:

log_recid This is set to a monotonically increasing log record id maintained by the system for this individual log file.

log_size This is set to the value of the `len` argument.

This allows existing log entries from one log file to be written to another log file without destroying the logged information.

NOTES:

The `_POSIX_LOGGING` feature flag is defined to indicate this service is available.

This implementation can not return the `EPERM` error.

This function is not defined in the POSIX specification. It is an extension provided by this implementation.

1.4.4 log_open - Open a log file

CALLING SEQUENCE:

```
#include <evlog.h>

int log_open(
  logd_t              *logdes,
  const char          *path,
  const log_query_t   *query
);
```

STATUS CODES:

A successful call to `log_open()` returns a value of zero and an unsuccessful call returns the `errno`.

EACCES Search permission is denied on a component of the `path` prefix, or the log file exists and read permission is denied.

EINTR A signal interrupted the call to `log_open()`.

EINVAL The log_severity field of the query argument exceeds `LOG_SEVERITY_MAX`.

EINVAL The `path` argument referred to a file that was not a log file.

EMFILE Too many log file descriptors are currently in use by this process.

ENAMETOOLONG The length of the path string exceeds `PATH_MAX`, or a pathname component is longer than `NAME_MAX` while `_POSIX_NO_TRUNC` is in effect.

ENFILE Too many files are currently open in the system.

ENOENT The file specified by the `path` argument does not exist.

ENOTDIR A component of the `path` prefix is not a directory.

DESCRIPTION:

The `log_open()` function establishes the connection between a log file and a log file descriptor. It creates an open log file descriptor that refers to this query stream on the specified log file The log file descriptor is used by the other log functions to refer to that log query stream. The `path` argument points to a pathname for a log file. A `path` argument of `NULL` specifies the current system log file.

The `query` argument is not `NULL`, then it points to a log query specification that is used to filter the records in the log file on subsequent `log_read()` operations. This restricts the set of event records read using the returned log file descriptor to those which match the query. A query match occurs for a given record when that record's facility is a member of the query's facility set and the record's severity is greater than or equal to the severity specified in the query.

If the value of the `query` argument is `NULL`, no query filter shall be applied.

NOTES:

The _POSIX_LOGGING feature flag is defined to indicate this service is available.

POSIX specifies that EINVAL will be returned if the log_facilities field of the query argument is not a valid facility set. In this implementation, this condition can never occur.

Many error codes that POSIX specifies to be returned by log_open() should actually be detected by open() and passed back by the log_open() implementation. In this implementation, EACCESS, EMFILE, ENAMETOOLONG, ENFILE, ENOENT, and ENOTDIR are detected in this manner.

1.4.5 log_read - Read from a log file

CALLING SEQUENCE:

```
#include <evlog.h>

int log_read(
  const logd_t      logdes,
  struct log_entry *entry,
  void             *log_buf,
  const size_t      log_len,
  const size_t     *log_sizeread
);
```

STATUS CODES:

A successful call to `log_read()` returns a value of zero and an unsuccessful call returns the `errno`.

E2BIG This error indicates an inconsistency in the implementation. Report this as a bug.

EBADF The `logdes` argument is not a valid log file descriptor.

EFAULT The `entry` argument is not a valid pointer to a log entry structure.

EFAULT The `log_sizeread` argument is not a valid pointer to a size_t.

EBUSY No data available. There are no unread event records remaining in this log file.

EINTR A signal interrupted the call to `log_read()`.

EIO An I/O error occurred in reading from the event log.

EINVAL The matching event record has data associated with it and `log_buf` was not a valid pointer.

EINVAL The matching event record has data associated with it which is longer than `log_len`.

DESCRIPTION:

The `log_read()` function reads the `log_entry` structure and up to `log_len` bytes of data from the next event record of the log file associated with the open log file descriptor `logdes`. The event record read is placed into the `log_entry` structure pointed to by `entry` and any data into the buffer pointed to by `log_buf`. The log record ID of the returned event record is be stored in the `log_recid` member of the `log_entry` structure for the event record.

If the query attribute of the open log file description associated with the `logdes` is set, the event record read will match that query.

If the `log_read()` is successful the call stores the actual length of the data associated with the event record into the location specified by `log_sizeread`. This number will be less than or equal to `log_len`.

NOTES:

The `_POSIX_LOGGING` feature flag is defined to indicate this service is available.

When `EINVAL` is returned, then no data is returned although the event record is returned. This is an extension to the POSIX specification.

The POSIX specification specifically allows `log_read()` to write greater than `log_len` bytes into `log_buf`. This is highly undesirable and this implementation will NOT do this.

1.4.6 log_notify - Notify Process of writes to the system log.

CALLING SEQUENCE:

```
#include <evlog.h>

int log_notify(
  const logd_t           logdes,
  const struct sigevent *notification
);
```

STATUS CODES:

A successful call to `log_notify()` returns a value of zero and an unsuccessful call returns the `errno`.

EBADF The logdes argument is not a valid log file descriptor.

EINVAL The notification argument specifies an invalid signal.

EINVAL The process has requested a notify on a log that will not be written to.

ENOSYS The function `log_notify()` is not supported by this implementation.

DESCRIPTION:

If the argument `notification` is not `NULL` this function registers the calling process to be notified of event records received by the system log, which match the query parameters associated with the open log descriptor specified by `logdes`. The notification specified by the `notification` argument shall be sent to the process when an event record received by the system log is matched by the query attribute of the open log file description associated with the `logdes` log file descriptor. If the calling process has already registered a notification for the `logdes` log file descriptor, the new notification shall replace the existing notification registration.

If the `notification` argument is `NULL` and the calling process is currently registered to be notified for the `logdes` log file descriptor, the existing registration shall be removed.

NOTES:

The `_POSIX_LOGGING` feature flag is defined to indicate this service is available.

1.4.7 log_close - Close log descriptor

CALLING SEQUENCE:

```
#include <evlog.h>

int log_close(
  const logd_t   logdes
);
```

STATUS CODES:

A successful call to `log_close()` returns a value of zero and an unsuccessful call returns the `errno`.

EBADF The logdes argument is not a valid log file descriptor.

DESCRIPTION:

The `log_close()` function deallocates the open log file descriptor indicated by `log_des`.

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

If the link count of the log file is zero, when all log file descriptors have been closed, the space occupied by the log file is freed and the log file shall no longer be accessible.

If the process has successfully registered a notification request for the log file descriptor, the registration is removed.

NOTES:

The `_POSIX_LOGGING` feature flag is defined to indicate this service is available.

1.4.8 log_seek - Reposition log file offset

CALLING SEQUENCE:

```
#include <evlog.h>

int log_seek(
  const logd_t    logdes,
  log_recid_t     log_recid
);
```

STATUS CODES:

A successful call to `log_seek()` returns a value of zero and an unsuccessful call returns the `errno`.

EBADF The `logdes` argument is not a valid log file descriptor.

EINVAL The `log_recid` argument is not a valid record id.

DESCRIPTION:

The `log_seek()` function sets the log file offset of the open log description associated with the `logdes` log file descriptor to the event record in the log file identified by `log_recid`. The `log_recid` argument is either the record id of a valid event record or one of the following values, as defined in the header file `<evlog.h>`:

LOG_SEEK_START Set log file position to point at the first event record in the log file.

LOG_SEEK_END Set log file position to point after the last event record in the log file.

NOTES:

The `_POSIX_LOGGING` feature flag is defined to indicate this service is available.

This implementation can not return EINTR.

This implementation can not return EINVAL to indicate that the `log_recid` argument is not a valid record id.

1.4.9 log_severity_before - Compare event record severities

CALLING SEQUENCE:

```
#include <evlog.h>

int log_severity_before(
  log_severity_t  s1,
  log_severity_t  s2
);
```

STATUS CODES:

0 The severity of `s1` is less than that of `s2`.

1 The severity of `s1` is greater than or equal that of `s2`.

EINVAL The value of either s1 or s2 exceeds `LOG_SEVERITY_MAX`.

DESCRIPTION:

The `log_severity_before()` function compares the severity order of the `s1` and `s2` arguments. If `s1` is of severity greater than or equal to that of `s2`, then this function returns 1. Otherwise, it returns 0.

If either `s1` or `s2` specify invalid severity values, the return value of `log_severity_before()` is unspecified.

NOTES:

The `_POSIX_LOGGING` feature flag is defined to indicate this service is available.

The POSIX specification of the return value for this function is ambiguous. If EINVAL is equal to 1 in an implementation, then the application can not distinguish between greater than and an error condition.

1.4.10 log_facilityemptyset - Manipulate log facility sets

CALLING SEQUENCE:

```
#include <evlog.h>

int log_facilityemptyset(
  log_facility_set_t  *set
);
```

STATUS CODES:

A successful call to `log_facilityemptyset()` returns a value of zero and a unsuccessful call returns the `errno`.

EFAULT The `set` argument is an invalid pointer.

DESCRIPTION:

The `log_facilityemptyset()` function initializes the facility set pointed to by the argument `set`, such that all facilities are excluded.

NOTES:

The `_POSIX_LOGGING` feature flag is defined to indicate this service is available.

Applications shall call either `log_facilityemptyset()` or `log_facilityfillset()` at least once for each object of type `log_facilityset_t` prior to any other use of that object. If such an object is not initialized in this way, but is nonetheless supplied as an argument to any of the `log_facilityaddset()`, `logfacilitydelset()`, `log_facilityismember()` or `log_open()` functions, the results are undefined.

1.4.11 log_facilityfillset - Manipulate log facility sets

CALLING SEQUENCE:

```
#include <evlog.h>

int log_facilityfillset(
  log_facility_set_t  *set
);
```

STATUS CODES:

A successful call to `log_facilityfillset()` returns a value of zero and a unsuccessful call returns the `errno`.

EFAULT The `set` argument is an invalid pointer.

DESCRIPTION:

The `log_facilityfillset()` function initializes the facility set pointed to by the argument `set`, such that all facilities are included.

NOTES:

The `_POSIX_LOGGING` feature flag is defined to indicate this service is available.

Applications shall call either `log_facilityemptyset()` or `log_facilityfillset()` at least once for each object of type `log_facilityset_t` prior to any other use of that object. If such an object is not initialized in this way, but is nonetheless supplied as an argument to any of the `log_facilityaddset()`, `logfacilitydelset()`, `log_facilityismember()` or `log_open()` functions, the results are undefined.

1.4.12 log_facilityaddset - Manipulate log facility sets

CALLING SEQUENCE:

```
#include <evlog.h>

int log_facilityaddset(
  log_facility_set_t  *set,
  log_facility_t       facilityno
);
```

STATUS CODES:

A successful call to `log_facilityaddset()` returns a value of zero and a unsuccessful call returns the `errno`.

EFAULT The `set` argument is an invalid pointer.

EINVAL The `facilityno` argument is not a valid facility.

DESCRIPTION:

The `log_facilityaddset()` function adds the individual facility specified by the value of the argument `facilityno` to the facility set pointed to by the argument `set`.

NOTES:

The `_POSIX_LOGGING` feature flag is defined to indicate this service is available.

Applications shall call either `log_facilityemptyset()` or `log_facilityfillset()` at least once for each object of type `log_facilityset_t` prior to any other use of that object. If such an object is not initialized in this way, but is nonetheless supplied as an argument to any of the `log_facilityaddset()`, `logfacilitydelset()`, `log_facilityismember()` or `log_open()` functions, the results are undefined.

1.4.13 log_facilitydelset - Manipulate log facility sets

CALLING SEQUENCE:

```
#include <evlog.h>

int log_facilitydelset(
  log_facility_set_t  *set,
  log_facility_t       facilityno
);
```

STATUS CODES:

A successful call to `log_facilitydelset()` returns a value of zero and a unsuccessful call returns the `errno`.

EFAULT The `set` argument is an invalid pointer.

EINVAL The `facilityno` argument is not a valid facility.

DESCRIPTION:

The `log_facilitydelset()` function deletes the individual facility specified by the value of the argument `facilityno` from the facility set pointed to by the argument `set`.

NOTES:

The `_POSIX_LOGGING` feature flag is defined to indicate this service is available.

Applications shall call either `log_facilityemptyset()` or `log_facilityfillset()` at least once for each object of type `log_facilityset_t` prior to any other use of that object. If such an object is not initialized in this way, but is nonetheless supplied as an argument to any of the `log_facilityaddset()`, `logfacilitydelset()`, `log_facilityismember()` or `log_open()` functions, the results are undefined.

1.4.14 log_facilityismember - Manipulate log facility sets

CALLING SEQUENCE:

```
#include <evlog.h>

int log_facilityismember(
  const log_facility_set_t *set,
  log_facility_t            facilityno,
  const int                *member
);
```

STATUS CODES:

A successful call to `log_facilityismember()` returns a value of zero and a unsuccessful call returns the `errno`.

EFAULT The `set` or `member` argument is an invalid pointer.

EINVAL The `facilityno` argument is not a valid facility.

DESCRIPTION:

The `log_facilityismember()` function tests whether the facility specified by the value of the argument `facilityno` is a member of the set pointed to by the argument `set`. Upon successful completion, the `log_facilityismember()` function either returns a value of one to the location specified by `member` if the specified facility is a member of the specified set or value of zero to the location specified by `member` if the specified facility is not a member of the specified set.

NOTES:

The `_POSIX_LOGGING` feature flag is defined to indicate this service is available.

Applications shall call either `log_facilityemptyset()` or `log_facilityfillset()` at least once for each object of type `log_facilityset_t` prior to any other use of that object. If such an object is not initialized in this way, but is nonetheless supplied as an argument to any of the `log_facilityaddset()`, `logfacilitydelset()`, `log_facilityismember()` or `log_open()` functions, the results are undefined.

1.4.15 log_facilityisvalid - Manipulate log facility sets

CALLING SEQUENCE:

```
#include <evlog.h>

int log_facilityisvalid(
  log_facility_t        facilityno
);
```

STATUS CODES:

A return value of zero indicates that the `facilityno` is valid and a return value other than zero represents an `errno`.

EFAULT The `set` or `member` argument is an invalid pointer.

EINVAL The `facilityno` argument is not a valid facility.

DESCRIPTION:

The `log_facilityisvalid()` function tests whether the facility specified by the value of the argument `facilityno` is a valid facility number. Upon successful completion, the the `log_facilityisvalid()` function either returns a value of 0 if the specified facility is a valid facility or value of EINVAL if the specified facility is not a valid facility.

NOTES:

The `_POSIX_LOGGING` feature flag is defined to indicate this service is available.

Applications shall call either `log_facilityemptyset()` or `log_facilityfillset()` at least once for each object of type `log_facilityset_t` prior to any other use of that object. If such an object is not initialized in this way, but is nonetheless supplied as an argument to any of the `log_facilityaddset()`, `logfacilitydelset()`, `log_facilityismember()` or `log_open()` functions, the results are undefined.

1.4.16 log_create - Creates a log file

CALLING SEQUENCE:

```
#include <evlog.h>

int log_create(
  logd_t      *ld,
  const char   *path,
);
```

STATUS CODES:

A successful call to `log_create()` returns a value of zero and a unsuccessful call returns the `errno`.

EEXIST The `path` already exists and O_CREAT and O_EXCL were used.

EISDIR The `path` refers to a directory and the access requested involved writing.

ETXTBSY The `path` refers to an executable image which is currently being executed and write access was requested.

EFAULT The `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.

ENAMETOOLONG The `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 already 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 The `path` refers to a file on a read-only filesystem and write access was requested.

ELOOP The `path` contains a reference to a circular symbolic link, ie a symbolic link whose expansion contains a reference to itself.

DESCRIPTION:

This function attempts to create a file associated with the `logdes` argument in the directory provided by the argument `path`.

NOTES:

The _POSIX_LOGGING feature flag is defined to indicate this service is available.

1.4.17 log_sys_create - Creates a system log file

CALLING SEQUENCE:

```
#include <evlog.h>

int log_sys_create();
```

STATUS CODES:

A successful call to `log_sys_create()` returns a value of zero and a unsuccessful call returns the `errno`.

EEXIST The directory path to the system log already exist.

DESCRIPTION:

This function will create a predefined system log directory path and system log file if they do not already exist.

NOTES:

The `_POSIX_LOGGING` feature flag is defined to indicate this service is available.

2 Process Dump Control Manager

2.1 Introduction

The process dump control manager provides a portable interface for changing the path to which a process dump is written. The capabilities in this manager were defined in the POSIX 1003.1h/D3 proposed standard titled **Services for Reliable, Available, and Serviceable Systems**.

The directives provided by the process dump control manager are:

- `dump_setpath` - Dump File Control

2.2 Background

There is currently no text in this section.

2.3 Operations

There is currently no text in this section.

2.4 Directives

This section details the process dump control 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 dump_setpath - Dump File Control

CALLING SEQUENCE:

```
#include <dump.h>

int dump_setpath(
  const char      *path
);
```

STATUS CODES:

EACESS Search permission is denied for a component of the path prefix, or write permission is denied on the directory containing the file.

ENAMETOOLONG The length of the argument exceeds `PATH_MAX` or a pathname component is longer than `NAME_MAX` while `_POSIX_NO_TRUNC` is in effect.

ENOENT The path argument points to an empty string.

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

EROFS The directory entry specified resides on a read-only file system.

DESCRIPTION:

The `dump_setpath()` function defines the pathname where process dumps are written. The pathname pointed to by `path` defines where a process dump file is written if the calling process terminates with a dump file. The `path` argument does not name a directory.

If the `path` argument is NULL, the system does not write a process dump file if the calling process terminates with a dump file. If the `dump_setpath` function fails, the pathname for writing process dumps does not change.

NOTES:

The `_POSIX_DUMP` feature flag is defined to indicate this service is available.

3 Configuration Space Manager

3.1 Introduction

The configuration space manager provides a portable interface for manipulating configuration data. The capabilities in this manager were defined in the POSIX 1003.1h/D3 proposed standard titled **Services for Reliable, Available, and Serviceable Systems**.

The directives provided by the configuration space manager are:

- `cfg_mount` - Mount a Configuration Space
- `cfg_unmount` - Unmount a Configuration Space
- `cfg_mknod` - Create a Configuration Node
- `cfg_get` - Get Configuration Node Value
- `cfg_set` - Set Configuration Node Value
- `cfg_link` - Create a Configuration Link
- `cfg_unlink` - Remove a Configuration Link
- `cfg_open` - Open a Configuration Space
- `cfg_read` - Read a Configuration Space
- `cfg_children` - Get Node Entries
- `cfg_mark` - Set Configuration Space Option
- `cfg_readdir` - Reads a directory
- `cfg_umask` - Sets a file creation mask
- `cfg_chmod` - Changes file mode
- `cfg_chown` - Changes the owner and/or group of a file

3.2 Background

3.2.1 Configuration Nodes

3.2.2 Configuration Space

3.2.3 Format of a Configuration Space File

3.3 Operations

3.3.1 Mount and Unmounting

3.3.2 Creating a Configuration Node

3.3.3 Removing a Configuration Node

3.3.4 Manipulating a Configuration Node

3.3.5 Traversing a Configuration Space

3.4 Directives

This section details the configuration space 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 cfg_mount - Mount a Configuration Space

CALLING SEQUENCE:

```
#include <cfg.h>

int cfg_mount(
  const char     *file,
  const char     *cfgpath,
  log_facility_t  notification,
);
```

STATUS CODES:

A successful call to `cfg_mount()` returns a value of zero and an unsuccessful call returns the `errno`.

EPERM The caller does not have the appropriate privilege.

EACCES Search permission is denied for a component of the path prefix.

EEXIST The file specified by the `file` argument does not exist

ENAMETOOLONG A component of a pathname exceeded `NAME_MAX` characters, or an entire path name exceed `PATH_MAX` characters while `_POSIX_NO_TRUNC` is in effect.

ENOENT A component of `cfgpath` does not exist.

ENOTDIR A component of the `file` path prefix is not a directory.

EBUSY The configuration space defined by `file` is already mounted.

EINVAL The notification argument specifies an invalid log facility.

DESCRIPTION:

The `cfg_mount()` function maps a configuration space defined by the file identified by the the `file` argument. The distinguished node of the mapped configuration space is mounted in the active space at the point identified by the `cfgpath` configuration pathname.

The `notification` argument specifies how changes to the mapped configuration space are communicated to the application. If the `notification` argument is NULL, no notification will be be performed for the mapped configuration space. If the Event Logging option is defined, the notification argument defines the facility to which changes in the mapped configuration space are logged. Otherwise, the `notification` argument specifies an implementation defined method of notifying the application of changes to the mapped configuration space.

NOTES:

The `_POSIX_CFG` feature flag is defined to indicate this service is available.

3.4.2 cfg_unmount - Unmount a Configuration Space

CALLING SEQUENCE:

```
#include <cfg.h>

int cfg_unmount(
  const char     *cfgpath
);
```

STATUS CODES:

A successful call to `cfg_umount()` returns a value of zero and an unsuccessful call returns the `errno`.

EPERM The caller does not have the appropriate privileges.

EACCES Search permission is denied for a component of the path prefix.

ENOENT A component of `cfgpath` does not exist.

ENAMETOOLONG A component of a pathname exceeded `NAME_MAX` characters, or an entire path name exceed `PATH_MAX` characters while `_POSIX_NO_TRUNC` is in effect.

EINVAL The requested node is not the distinguished node of a mounted configuration space.

EBUSY One or more processes has an open configuration traversal stream for the configuration space whose distinguished node is referenced by the cfgpath argument.

ELOOP A node appears more than once in the path specified by the `cfgpath` argument

ELOOP More than `SYMLOOP_MAX` symbolic links were encountered during resolution of the cfgpath argument

DESCRIPTION:

The `cfg_umount()` function unmaps the configuration space whose distinguished node is mapped in the active space at the location defined by `cfgpath` configuration pathname. All system resources allocated for this configuration space should be deallocated.

NOTES:

The `_POSIX_CFG` feature flag is defined to indicate this service is available.

3.4.3 cfg_mknod - Create a Configuration Node

CALLING SEQUENCE:

```
#include <cfg.h>

int cfg_mknod(
  const char   *cfgpath,
  mode_t        mode,
  cfg_type_t    type
);
```

STATUS CODES:

A successful call to `cfg_mknod()` returns a value of zero and an unsuccessful call returns the `errno`.

ENAMETOOLONG	A component of a pathname exceeded `NAME_MAX` characters, or an entire path name exceed `PATH_MAX` characters while `_POSIX_NO_TRUNC` is in effect.
ENOENT	A component of the path prefix does not exist.
EACCES	Search permission is denied for a component of the path prefix.
EPERM	The calling process does not have the appropriate privilege.
EEXIST	The named node exists.
EINVAL	The value of `mode` is invalid.
EINVAL	The value of `type` is invalid.
ELOOP	A node appears more than once in the path specified by the `cfg_path` argument
ELOOP	More than `SYMLOOP_MAX` symbolic links were encountered during resolution of the `cfgpath` argument.
EROFS	The named `node` resides on a read-only configuration space.

DESCRIPTION:

The `cfg_mknod()` function creates a new node in the configuration space which contains the pathname prefix of `cfgpath`. The node name is defined by the pathname suffix of `cfgpath`. The node permissions are specified by the value of `mode`. The node type is specified by the value of `type`.

NOTES:

The `_POSIX_CFG` feature flag is defined to indicate this service is available.

3.4.4 cfg_get - Get Configuration Node Value

CALLING SEQUENCE:

```
#include <cfg.h>

int cfg_get(
  const char  *cfgpath
  cfg_value_t *value
);
```

STATUS CODES:

A successful call to `cfg_get()` returns a value of zero and an unsuccessful call returns the `errno`.

ENAMETOOLONG A component of a pathname exceeded `NAME_MAX` characters, or an entire path name exceed `PATH_MAX` characters while `_POSIX_NO_TRUNC` is in effect.

ENOENT A component of `cfgpath` does not exist.

EACCES Search permission is denied for a component of the path prefix.

EPERM The calling process does not have the appropriate privileges.

ELOOP A node appears more than once in the path specified by the `cfgpath` argument

ELOOP More than `SYMLOOP_MAX` symbolic links were encountered during resolution of the `cfgpath` argument.

DESCRIPTION:

The `cfg_get()` function stores the value attribute of the configuration node identified by `cfgpath`, into the buffer described by the `value` pointer.

NOTES:

The `_POSIX_CFG` feature flag is defined to indicate this service is available.

3.4.5 cfg_set - Set Configuration Node Value

CALLING SEQUENCE:

```
#include <cfg.h>

int cfg_set(
  const char  *cfgpath
  cfg_value_t *value
);
```

STATUS CODES:

A successful call to `cfg_set()` returns a value of zero and an unsuccessful call returns the `errno`.

ENAMETOOLONG A component of a pathname exceeded `NAME_MAX` characters, or an entire path name exceed `PATH_MAX` characters while `_POSIX_NO_TRUNC` is in effect.

ENOENT A component of `cfgpath` does not exist

EACCES Search permission is denied for a component of the path prefix.

EPERM The calling process does not have the appropriate privilege.

ELOOP A node appears more than once in the path specified by the `cfgpath` argument.

ELOOP More than `SYMLOOP_MAX` symbolic links were encountered during resolution of the cfgpath argument.

DESCRIPTION:

The `cfg_set()` function stores the value specified by the `value` argument in the configuration node defined by the `cfgpath` argument.

NOTES:

The `_POSIX_CFG` feature flag is defined to indicate this service is available.

3.4.6 cfg_link - Create a Configuration Link

CALLING SEQUENCE:

```
#include <cfg.h>

int cfg_link(
  const char *src
  const char *dest
);
```

STATUS CODES:

A successful call to `cfg_link()` returns a value of zero and an unsuccessful call returns the `errno`.

ENAMETOOLONG	A component of a pathname exceeded `NAME_MAX` characters, or an entire path name exceed `PATH_MAX` characters while `_POSIX_NO_TRUNC` is in effect.
ENOENT	A component of either path prefix does not exist.
EACCES	A component of either path prefix denies search permission.
EACCES	The requested link requires writing in a node with a mode that denies write permission.
ENOENT	The node named by `src` does not exist.
EEXIST	The node named by `dest` does exist.
EPERM	The calling process does not have the appropriate privilege to modify the node indicated by the `src` argument.
EXDEV	The link named by `dest` and the node named by `src` are from different configuration spaces.
ENOSPC	The node in which the entry for the new link is being placed cannot be extended because there is no space left on the configuration space containing the node.
EIO	An I/O error occurred while reading from or writing to the configuration space to make the link entry.
EROFS	The requested link requires writing in a node on a read-only configuration space.

DESCRIPTION:

The `src` and `dest` arguments point to pathnames which name existing nodes. The `cfg_link()` function atomically creates a link between specified nodes, and increment by one the link count of the node specified by the `src` argument.

If the `cfg_link()` function fails, no link is created, and the link count of the node remains unchanged by this function call.

NOTES:

The _POSIX_CFG feature flag is defined to indicate this service is available.

3.4.7 cfg_unlink - Remove a Configuration Link

CALLING SEQUENCE:

```
#include <cfg.h>

int cfg_unlink(
  const char    *cfgpath
);
```

STATUS CODES:

A successful call to `cfg_unlink()` returns a value of zero and an unsuccessful call returns the `errno`.

ENAMETOOLONG A component of a pathname exceeded `NAME_MAX` characters, or an entire path name exceed `PATH_MAX` characters.

EACCES Search permission is denied on the node containing the link to be removed.

EACCES Write permission is denied on the node containing the link to be removed.

ENOENT A component of `cfgpath` does not exist.

EPERM The calling process does not have the appropriate privilege to modify the node indicated by the path prefix of the `cfgpath` argument.

EBUSY The node to be unlinked is the distinguished node of a mounted configuration space.

EIO An I/O error occurred while deleting the link entry or deallocating the node.

EROFS The named node resides in a read-only configuration space.

ELOOP A node appears more than once in the path specified by the `cfgpath` argument.

ELOOP More than `SYMLOOP_MAX` symbolic links were encountered during resolution of the cfgpath argument.

DESCRIPTION:

The `cfg_unlink()` function removes the link between the node specified by the `cfgpath` path prefix and the parent node specified by `cfgpath`, and decrements the link count of the `cfgpath` node.

When the link count of the node becomes zero, the space occupied by the node is freed and the node is no longer be accessible.

NOTES:

The `_POSIX_CFG` feature flag is defined to indicate this service is available.

3.4.8 cfg_open - Open a Configuration Space

CALLING SEQUENCE:

```
#include <cfg.h>

int cfg_open(
  const char     *pathnames[],
  int             options,
  int           (*compar)(const CFGENT **f1, const CFGENT **f2),
  CFG           **cfgstream
);
```

STATUS CODES:

A successful call to `cfg_open()` returns a value of zero and an unsuccessful call returns the `errno`.

EACCES	Search permission is denied for any component of a pathname.
ELOOP	A loop exists in symbolic links encountered during resolution of a pathname.
ENAMETOOLONG	The length of a pathname exceeds `PATH_MAX`, or a pathname component is longer than `NAME_MAX` while `_POSIX_NO_TRUNC`
ENOENT	The pathname argument is an empty string or the named node does not exist.
EINVAL	Either both or neither of `CFG_LOGICAL` and `CFG_PHYSICAL` are specified by the `options` argument
ENOMEM	Not enough memory is available to create the necessary structures.
ELOOP	More than `SYMLOOP_MAX` symbolic links were encountered during resolution of the `pathnames` argument.
ENAMETOOLONG	As a result of encountering a symbolic link in resolution of the pathname specified by the `pathnames` argument, the length of the substituted pathname string exceeded `PATH_MAX`.

DESCRIPTION:

The `cfg_open()` function opens a configuration traversal stream rooted in the configuration nodes name by the `pathnames` argument. It stores a pointer to a CFG object that represents that stream at the location identified the `cfgstream` pointer. The `pathnames` argument is an array of character pointers to NULL-terminated strings. The last member of this array is a NULL pointer.

The value of `options` is the bitwise inclusive OR of values from the following lists. Applications supply exactly one of the first two values below in `options`.

CFG_LOGICAL When symbolic links referencing existing nodes are encountered during the traversal, the `cfg_info` field of the returned CFGENT structure describes the target node pointed to by the link instead of the link itself, unless the target node does not exist. If the target node has children, the pre-order return, followed by the return of structures referencing all of its descendants, followed by a post-order return, is done.

CFG_PHYSICAL When symbolic links are encountered during the traversal, the `cfg_info` field is used to describe the symbolic link.

Any combination of the remaining flags can be specified in the value of `options`

CFG_COMFOLLOW When symbolic links referencing existing nodes are specified in the `pathnames` argument, the `cfg_info` field of the returned CFGENT structure describes the target node pointed to by the link instead of the link itself, unless the target node does not exist. If the target node has children, the pre-order return, followed by the return of structures referencing all its descendants, followed by a post-order return, is done.

CFG_XDEV The configuration space functions do not return a CFGENT structure for any node in a different configuration space than the configuration space of the nodes identified by the CFGENT structures for the `pathnames` argument.

The `cfg_open()` argument `compar` is either a NULL or point to a function that is called with two pointers to pointers to CFGENT structures that returns less than, equal to , or greater than zero if the node referenced by the first argument is considered to be respectively less than, equal to, or greater than the node referenced by the second. The CFGENT structure fields provided to the comparison routine is as described with the exception that the contents of the `cfg_path` and `cfg_pathlen` fields are unspecified.

This comparison routine is used to determine the order in which nodes in directories encountered during the traversal are returned, and the order of traversal when more than one node is specified in the `pathnames` argument to `cfg_open()`. If a comparison routine is specified, the order of traversal is from the least to the greatest. If the `compar` argument is NULL, the order of traversal shall is listed in the `pathnames` argument.

NOTES:

The `_POSIX_CFG` feature flag is defined to indicate this service is available.

3.4.9 cfg_read - Read a Configuration Space

CALLING SEQUENCE:

```
#include <cfg.h>

int cfg_read(
  CFG           *cfgp,
  CFGENT       **node
);
```

STATUS CODES:

A successful call to `cfg_read()` returns a value of zero and an unsuccessful call returns the `errno`.

EACCES Search permission is denied for any component of a pathname.

EBADF The `cfgp` argument does not refer to an open configuration space.

ELOOP A loop exists in symbolic links encountered during resolution of a pathname.

ENOENT A named `node` does not exist.

ENOMEM Not enough memory is available to create the necessary structures.

ELOOP More than `SYMLOOP_MAX` symbolic links were encountered during resolution of the cfgpath argument.

ENAMETOOLONG As a result of encountering a symbolic link in resolution of the pathname specified by the pathnames argument, the length of the substituted pathname string exceeded `PATH_MATH`.

DESCRIPTION:

The `cfg_read()` function returns a pointer to a CFGENT structure representing a node in the configuration space to which `cfgp` refers. The returned pointer is stored at the location indicated by the `node` argument.

The child nodes of each node in the configuration tree is returned by `cfg_read()`. If a comparison routine was specified to the `cfg_open()` function, the order of return of the child nodes is as specified by the `compar` routine, from least to greatest. Otherwise, the order of return is unspecified.

Structures referencing nodes with children is returned by the function `cfg_read()` at least twice [unless the application specifies otherwise with `cfg_mark()`]-once immediately before the structures representing their descendants, are returned (pre-order), and once immediately after structures representing all of their descendants, if any, are returned (post-order). The CFGENT structure returned in post-order (with the exception of the `cfg_info` field) is identical to that returned in pre-order. Structures referencing nodes of other types is returned at least once.

The fields of the CFGENT structure contains the following information:

cfg_parent
A pointer to the structure returned by the `cfg_read()` function for the node that contains the entry for the current node. A `cfg_parent` structure is provided for the node(s) specified by the `pathnames` argument to the `cfg_open()` function, but the contents of other than its `cfg_number`, `cfg_pointer`, `cfg_parent`, and `cfg_parent`, and `cfg_level` fields are unspecified. Its `cfg_link` field is unspecified.

cfg_link
Upon return from the `cfg_children()` function, the `cfg_link` field points to the next CFGENT structure in a NULL-terminated linked list of CFGENT structures. Otherwise, the content of the `cfg_link` field is unspecified.

cfg_cycle
If the structure being returned by `cfg_read()` represents a node that appears in the `cfg_parent` linked list tree, the `cfg_cycle` field shall point to the structure representing that entry from the `cfg_parent` linked list. Otherwise the content of the `cfg_cycle` field is unspecified.

cfg_number
The `cfg_number` field is provided for use by the application program. It is initialized to zero for each new node returned by the `cfg_read()` function, but is not further modified by the configuration space routines.

cfg_pointer
The `cfg_pointer` field is provided for use by the application program. It is initialized to NULL for each new node returned by the `cfg_read()` function, but is not further modified by the configuration space routines.

cfg_path
A pathname for the node including and relative to the argument supplied to the `cfg_open()` routine for this configuration space. This pathname may be longer than `PATH_MAX` bytes. This pathname is NULL-terminated.

cfg_name
The nodename of the node.

cfg_pathlen
The length of the string pointed at by the `cfg_path` field when returned by `cfg_read()`.

cfg_namelen
The length of the string pointed at by the `cfg_name` field.

cfg_level
The depth of the current entry in the configuration space. The `cfg_level` field of the `cfg_parent` structure for each of the node(s) specified in the `pathnames` argument to the `cfg_open()` function is set to 0, and this number is incremented for each node level descendant.

cfg_info
This field contains one of the values listed below. If an object can have more than one info value, the first appropriate value listed below is returned.

- **CFG_D** The structure represents a node with children in pre-order.

CFG_DC The structure represents a node that is a parent of the node most recently returned by `cfg_read()`. The `cfg_cycle` field references the structure previously returned by `cfg_read` that is the same as the returned structure.

CFG_DEFAULT The structure represents a node that is not represented by one of the other node types

CFG_DNR The structure represents a node, not of type symlink, that is unreadable. The variable `cfg_errno` is set to the appropriate value.

CFG_DP The structure represents a node with children in post-order. This value occurs only if CFG_D has previously been returned for this entry.

CFG_ERR The structure represents a node for which an error has occurred. The variable `cfg_errno` is set to the appropriate value.

CFG_F The structure represents a node without children.

CFG_SL The structure represents a node of type symbolic link.

CFG_SLNONET The structure represents a node of type symbolic link with a target node for which node characteristic information cannot be obtained.

Structures returned by `cfg_read()` with a `cfg_info` field equal to CFG_D is accessible until a subsequent call, on the same configuration traversal stream, to `cfg_close()`, or to `cfg_read()` after they have been returned by the `cfg_read` function in post-order. Structures returned by `cfg_read()` with an `cfg_info` field not equal to CFG_D is accessible until a subsequent call, on the same configuration traversal stream, to `cfg_close()` or `cfg_read()`.

The content of the `cfg_path` field is specified only for the structure most recently returned by `cfg_read()`.

The specified fields in structures in the list representing nodes for which structures have previously been returned by `cfg_children()`, is identical to those returned by `cfg_children()`, except that the contents of the `cfg_path` and `cfg_pathlen` fields are unspecified.

NOTES:

The `_POSIX_CFG` feature flag is defined to indicate this service is available.

3.4.10 cfg_children - Get Node Entries

CALLING SEQUENCE:

```
#include <cfg.h>

int cfg_children(
  CFG          *cfgp,
  int           options,
  CFGENT      **children
);
```

STATUS CODES:

A successful call to `cfg_children()` returns a value of zero and an unsuccessful call returns the `errno`.

EACCES Search permission is denied for any component of a pathname

EBADF The `cfgp` argument does not refer to an open configuration space.

ELOOP A loop exists in symbolic links encountered during resolution of a pathname.

ENAMETOOLONG The length of a pathname exceeds `PATH_MAX`, or a pathname component is longer than `NAME_MAX` while `_POSIX_NO_TRUNC` is in effect.

EINVAL The specified value of the `options` argument is invalid.

ENOENT The named node does not exist.

ENOMEM Not enough memory is available to create the necessary structures.

DESCRIPTION:

The first `cfg_children()` call after a `cfg_read()` returns information about the first node without children under the node returned by `cfg_read()`. Subsequent calls to `cfg_children()` without the intervening `cfg_read()` shall return information about the remaining nodes without children under that same node.

If `cfg_read()` has not yet been called for the configuration traversal stream represented by `cfgp`, `cfg_children()` returns a pointer to the first entry in a list of the nodes represented by the `pathnames` argument to `cfg_open()`.

In either case, the list is NULL-terminated, ordered by the user-specified comparison function, if any, and linked through the `cfg_link` field.

NOTES:

The `_POSIX_CFG` feature flag is defined to indicate this service is available.

3.4.11 cfg_mark - Set Configuration Space Options

CALLING SEQUENCE:

```
#include <cfg.h>

int cfg_mark(
  CFG          *cfgp,
  CFGENT       *f,
  int           options
);
```

STATUS CODES:

A successful call to `cfg_mark()` returns a value of zero and an unsuccessful call returns the `errno`.

EINVAL The specified combination of the `cfgp` and `f` arguments is not supported by the implementation.

EINVAL The specified value of the `options` argument is invalid.

DESCRIPTION:

The `cfg_mark()` function modifies the subsequent behavior of the `cfg` functions with regard to the node referenced by the structure pointed to by the argument `f` or the configuration space referenced by the structure pointed to by the argument `cfgp`.

Exactly one of the `f` argument and the `cfgp` argument is NULL.

The value of the `options` argument is exactly one of the flags specified in the following list:

CFG_AGAIN If the `cfgp` argument is non-NULL, or the `f` argument is NULL, or the structure referenced by `f` is not the one most recently returned by `cfg_read()`, `cfg_mark()` returns an error. Otherwise, the next call to the `cfg_read()` function returns the structure referenced by `f` with the `cfg_info` field reinitialized. Subsequent behavior of the `cfg` functions are based on the reinitialized value of `cfg_info`.

CFG_SKIP If the `cfgp` argument is non-NULL, or the `f` argument is NULL, or the structure referenced by `f` is not one of those specified as accessible, or the structure referenced by `f` is not for a node of type pre-order node, `cfg_mark()` returns an error. Otherwise, no more structures for the node referenced by `f` or its descendants are returned by the `cfg_read()` function.

CFG_FOLLOW If the `cfgp` argument is non-NULL, or the `f` argument is NULL, or the structure referenced by `f` is not one of those specified as accessible, or the structure referenced by `f` is not for a node of type symbolic link, `cfg_mark()` returns an error. Otherwise, the next call to the cfg_read() function returns the structure referenced by f with the cfg_info field reset to reflect the target of the symbolic link instead of the symbolic link itself. If the target of the link is node with children, the pre-order return, followed by the return of structures referencing all of its descendants, followed by a post-order return, shall be done.

If the target of the symbolic link does not exist, the fields of the structure by cfg_read() shall be unmodified, except that the cfg_info field shall be reset to CFG_SLNONE.

NOTES:

The _POSIX_CFG feature flag is defined to indicate this service is available.

3.4.12 cfg_close - Close a Configuration Space

CALLING SEQUENCE:

```
#include <cfg.h>

int cfg_close(
  CFG          *cfgp
);
```

STATUS CODES:

A successful call to `cfg_close()` returns a value of zero and an unsuccessful call returns the `errno`.

EBADF The `cfgp` argument does not refer to an open configuration space traversal stream.

DESCRIPTION:

The `cfg_close()` function closes a configuration space transversal stream represented by the CFG structure pointed at by the `cfgp` argument. All system resources allocated for this configuration space traversal stream should be deallocated. Upon return, the value of `cfgp` need not point to an accessible object of type CFG.

NOTES:

The `_POSIX_CFG` feature flag is defined to indicate this service is available.

3.4.13 cfg_readdir - Reads a directory

CALLING SEQUENCE:

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

struct dirent *cfg_readdir(
  DIR   *dirp
);
```

STATUS CODES:

EBADF Invalid file descriptor

DESCRIPTION:

The `cfg_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 `cfg_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 `cfg_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.

This function is not defined in the POSIX specification. It is an extension provided by this implementation.

3.4.14 cfg_umask - Sets a file creation mask.

CALLING SEQUENCE:

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

mode_t cfg_umask(
  mode_t cmask
);
```

STATUS CODES:

DESCRIPTION:

The `cfg_umask()` function sets the process node 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.

The file creation mask is inherited across `fork()` and `exec()` calls. This makes it possible to alter the default permission bits of created files.

NOTES: None

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

3.4.15 cfg_chmod - Changes file mode.

CALLING SEQUENCE:

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

int cfg_chmod(
  const char *path,
  mode_t      mode
);
```

STATUS CODES:

A successful call to `cfg_chmod()` returns a value of zero and an unsuccessful call returns the `errno`.

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 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 node and the calling process does not have the appropriate privileges, `cfg_chmod()` returns -1 and sets `errno` to `EPERM`.

NOTES:

3.4.16 cfg_chown - Changes the owner and/or group of a file.

CALLING SEQUENCE:

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

int cfg_chown(
  const char *path,
  uid_t       owner,
  gid_t       group
);
```

STATUS CODES:

A successful call to `cfg_chown()` returns a value of zero and an unsuccessful call returns the `errno`.

EACCES	Search permission is denied for a directory in a file's path prefix
EINVAL	Invalid argument
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 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 `cfg_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.

4 Administration Interface Manager

4.1 Introduction

The administration interface manager provides a portable interface for some system administrative functions. The capabilities in this manager are defined in the POSIX 1003.1h/D3 proposed standard titled **Services for Reliable, Available, and Serviceable Systems**.

The directives provided by the administration interface manager are:

- `admin_shutdown` - Shutdown the system

4.2 Background

4.2.1 admin_args Structure

```
put structure here
```

admin_type
This field ...

ADMIN_AUTOBOOT
The default, causing the system to reboot in its usual fashion. The `admin_data` field points to an implementation defined string that specifies the system image to reboot.

ADMIN_HALT
The system is simply halted; no reboot takes place.

ADMIN_FAST
The system does no send SIGTERM to active processes before halting.

ADMIN_IMMEDIATE
The system does not perform any of the normal shutdown procedures.

ADMIN_ALTSYSTEM
The system reboots using the `admin_data` string as a specification of the system to be booted.

ADMIN_ALTCONFIG
The system reboots using the `admin_data` string as a specification of the initial implicit configuration space.

ADMIN_SYSDUMP
Dump kernal memory before rebooting.

ADMIN_INIT
An option allowing the specification of an alternate initial program to be run when the system reboots.

admin_data
This field ...

4.3 Operations

4.3.1 Shutting Down the System

4.4 Directives

This section details the administration interface 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 admin_shutdown - Shutdown the system

CALLING SEQUENCE:

```
#include <admin.h>

int admin_shutdown(
  struct admin_args   *args[],
  size_t               nargs
);
```

STATUS CODES:

EINVAL An invalid argument was passed to the function call.

EPERM The caller does not have appropriate permission for shutting down the system.

DESCRIPTION:

The `admin_shutdown` function restarts the system. The `args` argument specifies alternate or optional behavior for the `admin_shutdown` function. The `admin_type` member of each element of the `args` array specifies the optional behavior to be performed. There are some `admin_types` values that may provoke unspecified behavior. The `nargs` argument specifies the length of the `args` array.

NOTES:

The `_POSIX_ADMIN` feature flag is defined to indicate this service is available.

5 Stack Bounds Checker

5.1 Introduction

The stack bounds checker is an RTEMS support component that determines if a task has overflowed its run-time stack. The routines provided by the stack bounds checker manager are:

- `Stack_check_Initialize` - Initialize the Stack Bounds Checker
- `Stack_check_Dump_usage` - Report Task Stack Usage

5.2 Background

5.2.1 Task Stack

Each task in a system has a fixed size stack associated with it. This stack is allocated when the task is created. As the task executes, the stack is used to contain parameters, return addresses, saved registers, and local variables. The amount of stack space required by a task is dependent on the exact set of routines used. The peak stack usage reflects the worst case of subroutine pushing information on the stack. For example, if a subroutine allocates a local buffer of 1024 bytes, then this data must be accounted for in the stack of every task that invokes that routine.

Recursive routines make calculating peak stack usage difficult, if not impossible. Each call to the recursive routine consumes n bytes of stack space. If the routine recursives 1000 times, then `1000 *` n bytes of stack space are required.

5.2.2 Execution

The stack bounds checker operates as a set of task extensions. At task creation time, the task's stack is filled with a pattern to indicate the stack is unused. As the task executes, it will overwrite this pattern in memory. At each task switch, the stack bounds checker's task switch extension is executed. This extension checks that the last `n` bytes of the task's stack have not been overwritten. If they have, then a blown stack error is reported.

The number of bytes checked for an overwrite is processor family dependent. The minimum stack frame per subroutine call varies widely between processor families. On CISC families like the Motorola MC68xxx and Intel ix86, all that is needed is a return address. On more complex RISC processors, the minimum stack frame per subroutine call may include space to save a significant number of registers.

Another processor dependent feature that must be taken into account by the stack bounds checker is the direction that the stack grows. On some processor families, the stack grows up or to higher addresses as the task executes. On other families, it grows down to lower addresses. The stack bounds checker implementation uses the stack description definitions provided by every RTEMS port to get for this information.

5.3 Operations

5.3.1 Initializing the Stack Bounds Checker

The stack checker is initialized automatically when its task create extension runs for the first time. When this occurs, the `Stack_check_Initialize` is invoked.

The application must include the stack bounds checker extension set in its set of Initial Extensions. This set of extensions is defined as `STACK_CHECKER_EXTENSION`. If using `<confdefs.h>` for Configuration Table generation, then all that is necessary is to define the macro `STACK_CHECKER_ON` before including `<confdefs.h>` as shown below:

```
#define STACK_CHECKER_ON
  ...
#include <confdefs.h>
```

5.3.2 Reporting Task Stack Usage

The application may dynamically report the stack usage for every task in the system by calling the `Stack_check_Dump_usage` routine. This routine prints a table with the peak usage and stack size of every task in the system. The following is an example of the report generated:

```
    ID      NAME       LOW         HIGH     AVAILABLE     USED
0x04010001  IDLE  0x003e8a60  0x003e9667       2952        200
0x08010002  TA1   0x003e5750  0x003e7b57       9096       1168
0x08010003  TA2   0x003e31c8  0x003e55cf       9096       1168
0x08010004  TA3   0x003e0c40  0x003e3047       9096       1104
0xffffffff  INTR  0x003ecfc0  0x003effbf      12160        128
```

Notice the last time. The task id is 0xffffffff and its name is "INTR". This is not actually a task, it is the interrupt stack.

5.3.3 When a Task Overflows the Stack

When the stack bounds checker determines that a stack overflow has occurred, it will attempt to print a message identifying the task and then shut the system down. If the stack overflow has caused corruption, then it is possible that the message can not be printed.

The following is an example of the output generated:

```
BLOWN STACK!!! Offending task(0x3eb360): id=0x08010002; name=0x54413120
  stack covers range 0x003e5750 - 0x003e7b57 (9224 bytes)
  Damaged pattern begins at 0x003e5758 and is 128 bytes long
```

The above includes the task id and a pointer to the task control block as well as enough information so one can look at the task's stack and see what was happening.

5.4 Routines

This section details the stack bounds checker's routines. A subsection is dedicated to each of routines and describes the calling sequence, related constants, usage, and status codes.

5.4.1 Stack_check_Initialize - Initialize the Stack Bounds Checker

CALLING SEQUENCE:

```
void Stack_check_Initialize( void );
```

STATUS CODES: NONE

DESCRIPTION:

Initialize the stack bounds checker.

NOTES:

This is performed automatically the first time the stack bounds checker task create extension executes.

5.4.2 Stack_check_Dump_usage - Report Task Stack Usage

CALLING SEQUENCE:

```
void Stack_check_Dump_usage( void );
```

STATUS CODES: NONE

DESCRIPTION:

This routine prints a table with the peak stack usage and stack space allocation of every task in the system.

NOTES:

NONE

6 Rate Monotonic Period Statistics

6.1 Introduction

The rate monotonic period statistics manager is an RTEMS support component that maintains statistics on the execution characteristics of each task using a period. The routines provided by the rate monotonic period statistics manager are:

- `Period_usage_Initialize` - Initialize the Period Statistics
- `Period_usage_Reset` - Reset the Period Statistics
- `Period_usage_Update` - Update the Statistics for this Period
- `Period_usage_Dump` - Report Period Statistics Usage

6.2 Background

6.3 Period Statistics

This manager maintains a set of statistics on each period. The following is a list of the information kept:

- `id` is the id of the period.
- `count` is the total number of periods executed.
- `missed_count` is the number of periods that were missed.
- `min_cpu_time` is the minimum amount of CPU execution time consumed on any execution of the periodic loop.
- `max_cpu_time` is the maximum amount of CPU execution time consumed on any execution of the periodic loop.
- `total_cpu_time` is the total amount of CPU execution time consumed by executions of the periodic loop.
- `min_wall_time` is the minimum amount of wall time that passed on any execution of the periodic loop.
- `max_wall_time` is the maximum amount of wall time that passed on any execution of the periodic loop.
- `total_wall_time` is the total amount of wall time that passed during executions of the periodic loop.

The above information is inexpensive to maintain and can provide very useful insights into the execution characteristics of a periodic task loop.

6.3.1 Analysis of the Reported Information

The period statistics reported must be analyzed by the user in terms of what the applications is. For example, in an application where priorities are assigned by the Rate Monotonic

Algorithm, it would be very undesirable for high priority (i.e. frequency) tasks to miss their period. Similarly, in nearly any application, if a task were supposed to execute its periodic loop every 10 milliseconds and it averaged 11 milliseconds, then application requirements are not being met.

The information reported can be used to determine the "hot spots" in the application. Given a period's id, the user can determine the length of that period. From that information and the CPU usage, the user can calculate the percentage of CPU time consumed by that periodic task. For example, a task executing for 20 milliseconds every 200 milliseconds is consuming 10 percent of the processor's execution time. This is usually enough to make it a good candidate for optimization.

However, execution time alone is not enough to gauge the value of optimizing a particular task. It is more important to optimize a task executing 2 millisecond every 10 milliseconds (20 percent of the CPU) than one executing 10 milliseconds every 100 (10 percent of the CPU). As a general rule of thumb, the higher frequency at which a task executes, the more important it is to optimize that task.

6.4 Operations

6.4.1 Initializing the Period Statistics

The period statistics manager must be explicitly initialized before any calls to this manager. This is done by calling the `Period_usage_Initialize` service.

6.4.2 Updating Period Statistics

It is the responsibility of each period task loop to update the statistics on each execution of its loop. The following is an example of a simple periodic task that uses the period statistics manager:

```
rtems_task Periodic_task()
{
  rtems_name        name;
  rtems_id          period;
  rtems_status_code status;

  name = rtems_build_name( 'P', 'E', 'R', 'D' );

  (void) rate_monotonic_create( name, &period );

  while ( 1 ) {
    if ( rate_monotonic_period( period, 100 ) == TIMEOUT )
      break;

    /* Perform some periodic actions */

    /* Report statistics */
    Period_usage_Update( period_id );
  }

  /* missed period so delete period and SELF */

  (void) rate_monotonic_delete( period );
  (void) task_delete( SELF );
}
```

6.4.3 Reporting Period Statistics

The application may dynamically report the period usage for every period in the system by calling the `Period_usage_Dump` routine. This routine prints a table with the following information per period:

- period id
- id of the task that owns the period
- number of periods executed
- number of periods missed
- minimum/maximum/average cpu use per period
- minimum/maximum/average wall time per period

The following is an example of the report generated:

```
Period information by period
   ID      OWNER   PERIODS  MISSED    CPU TIME    WALL TIME
0x28010001  TA1       502     0       0/1/ 1.00    0/0/0.00
0x28010002  TA2       502     0       0/1/ 1.00    0/0/0.00
0x28010003  TA3       502     0       0/1/ 1.00    0/0/0.00
0x28010004  TA4       502     0       0/1/ 1.00    0/0/0.00
0x28010005  TA5        10     0       0/1/ 0.90    0/0/0.00
```

6.5 Routines

This section details the rate monotonic period statistics manager's routines. A subsection is dedicated to each of this manager's routines and describes the calling sequence, related constants, usage, and status codes.

6.5.1 Period_usage_Initialize - Initialize the Period Statistics

CALLING SEQUENCE:

```
void Period_usage_Initialize( void );
```

STATUS CODES: NONE

DESCRIPTION:

This routine allocates the table used to contain the period statistics. This table is then initialized by calling the `Period_usage_Reset` service.

NOTES:

This routine invokes the `malloc` routine to dynamically allocate memory.

6.5.2 Period_usage_Reset - Reset the Period Statistics

CALLING SEQUENCE:

```
void Period_usage_Reset( void );
```

STATUS CODES: NONE

DESCRIPTION:

This routine re-initializes the period statistics table to its default state which is when zero period executions have occurred.

NOTES:

NONE

6.5.3 Period_usage_Update - Update the Statistics for this Period

CALLING SEQUENCE:

```
void Period_usage_Update(
  rtems_id   id
);
```

STATUS CODES: NONE

DESCRIPTION:

The `Period_usage_Update` routine must be invoked at the "bottom" of each periodic loop iteration to update the statistics.

NOTES:

NONE

6.5.4 Period_usage_Dump - Report Period Statistics Usage

CALLING SEQUENCE:

```
void Period_usage_Dump( void );
```

STATUS CODES: NONE

DESCRIPTION:

This routine prints out a table detailing the period statistics for all periods in the system.

NOTES:

NONE

7 CPU Usage Statistics

7.1 Introduction

The CPU usage statistics manager is an RTEMS support component that provides a convenient way to manipulate the CPU usage information associated with each task The routines provided by the CPU usage statistics manager are:

- `CPU_usage_Dump` - Report CPU Usage Statistics
- `CPU_usage_Reset` - Reset CPU Usage Statistics

7.2 Background

7.3 Operations

7.4 Report CPU Usage Statistics

7.4.1 Reporting Period Statistics

The application may dynamically report the CPU usage for every task in the system by calling the `CPU_usage_Dump` routine. This routine prints a table with the following information per task:

- task id
- task name
- number of clock ticks executed
- percentage of time consumed by this task

The following is an example of the report generated:

```
CPU Usage by thread
   ID         NAME         TICKS    PERCENT
0x04010001   IDLE            0      0.000
0x08010002   TA1          1203      0.748
0x08010003   TA2           203      0.126
0x08010004   TA3           202      0.126

Ticks since last reset = 1600

Total Units = 1608
```

Notice that the "Total Units" is greater than the ticks per reset. This is an artifact of the way in which RTEMS keeps track of CPU usage. When a task is context switched into the CPU, the number of clock ticks it has executed is incremented. While the task is executing, this number is incremented on each clock tick. Otherwise, if a task begins and completes execution between successive clock ticks, there would be no way to tell that it executed at all.

Another thing to keep in mind when looking at idle time, is that many systems – especially during debug – have a task providing some type of debug interface. It is usually fine to think of the total idle time as being the sum of the IDLE task and a debug task that will not be included in a production build of an application.

7.5 Reset CPU Usage Statistics

Invoking the `CPU_usage_Reset` routine resets the CPU usage statistics for all tasks in the system.

7.6 Directives

This section details the CPU usage statistics 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.

7.6.1 CPU_usage_Dump - Report CPU Usage Statistics

CALLING SEQUENCE:

```
void CPU_usage_Dump( void );
```

STATUS CODES: NONE

DESCRIPTION:

This routine prints out a table detailing the CPU usage statistics for all tasks in the system.

NOTES:

NONE

7.6.2 CPU_usage_Reset - Reset CPU Usage Statistics

CALLING SEQUENCE:

```
void CPU_usage_Reset( void );
```

STATUS CODES: NONE

DESCRIPTION:

This routine re-initializes the CPU usage statistics for all tasks in the system to their initial state. The initial state is that a task has not executed and thus has consumed no CPU time. default state which is when zero period executions have occurred.

NOTES:

NONE

8 Error Reporting Support

8.1 Introduction

These error reporting facilities are an RTEMS support component that provide convenient facilities for handling error conditions in an RTEMS application. of each task using a period. The services provided by the error reporting support component are:

- `rtems_error` - Report an Error
- `rtems_panic` - Report an Error and Panic
- `rtems_status_text` - ASCII Version of RTEMS Status

8.2 Background

8.2.1 Error Handling in an Embedded System

Error handling in an embedded system is a difficult problem. If the error is severe, then the only recourse is to shut the system down in a safe manner. Other errors can be detected and compensated for. The error reporting routines in this support component – `rtems_error` and `rtems_panic` assume that if the error is severe enough, then the system should be shutdown. If a simple shutdown with some basic diagnostic information is not sufficient, then these routines should not be used in that particular system. In this case, use the `rtems_status_text` routine to construct an application specific error reporting routine.

8.3 Operations

8.3.1 Reporting an Error

The `rtems_error` and `rtems_panic` routines can be used to print some diagnostic information and shut the system down. The `rtems_error` routine is invoked with a user specified error level indicator. This error indicator is used to determine if the system should be shutdown after reporting this error.

8.4 Routines

This section details the error reporting support compenent's routine. A subsection is dedicated to each of this manager's routines and describes the calling sequence, related constants, usage, and status codes.

8.4.1 rtems_status_text - ASCII Version of RTEMS Status

CALLING SEQUENCE:

```
const char *rtems_status_text(
  rtems_status_code status
);
```

STATUS CODES:

Returns a pointer to a constant string that describes the given RTEMS status code.

DESCRIPTION:

This routine returns a pointer to a string that describes the RTEMS status code specified by `status`.

NOTES:

NONE

8.4.2 rtems_error - Report an Error

CALLING SEQUENCE:

```
int rtems_error(
  int         error_code,
  const char *printf_format,
  ...
);
```

STATUS CODES:

Returns the number of characters written.

DESCRIPTION:

This routine prints the requested information as specified by the `printf_format` parameter and the zero or more optional arguments following that parameter. The `error_code` parameter is an error number with either `RTEMS_ERROR_PANIC` or `RTEMS_ERROR_ABORT` bitwise or'ed with it. If the `RTEMS_ERROR_PANIC` bit is set, then then the system is system is shutdown via a call to `_exit`. If the `RTEMS_ERROR_ABORT` bit is set, then then the system is system is shutdown via a call to `abort`.

NOTES:

NONE

8.4.3 rtems_panic - Report an Error and Panic

CALLING SEQUENCE:

```
int rtems_panic(
   const char *printf_format,
   ...
);
```

STATUS CODES:

Returns the number of characters written.

DESCRIPTION:

This routine is a wrapper for the `rtems_error` routine with an implied error level of `RTEMS_ERROR_PANIC`. See `rtems_error` for more information.

NOTES:

NONE

9 Monitor Task

9.1 Introduction

The monitor task is a simple interactive shell that allows the user to make inquries about he state of various system objects. The routines provided by the monitor task manager are:

- `rtems_monitor_init` - Initialize the Monitor Task
- `rtems_monitor_wakeup` - Wakeup the Monitor Task

9.2 Background

There is no background information.

9.3 Operations

9.3.1 Initializing the Monitor

The monitor is initialized by calling `rtems_monitor_init`. When initialized, the monitor is created as an independent task. An example of initializing the monitor is shown below:

```
#include <rtems/monitor.h>
        ...
rtems_monitor_init(0);
```

The "0" parameter to the `rtems_monitor_init` routine causes the monitor to immediately enter command mode. This parameter is a bitfield. If the monitor is to suspend itself on startup, then the `RTEMS_MONITOR_SUSPEND` bit should be set.

9.4 Routines

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

9.4.1 rtems_monitor_init - Initialize the Monitor Task

CALLING SEQUENCE:

```
void rtems_monitor_init(
  unsigned32 monitor_flags
);
```

STATUS CODES: NONE

DESCRIPTION:

This routine initializes the RTEMS monitor task. The `monitor_flags` parameter indicates how the server task is to start. This parameter is a bitfield and has the following constants associated with it:

- **RTEMS_MONITOR_SUSPEND** - suspend monitor on startup
- **RTEMS_MONITOR_GLOBAL** - monitor should be global

If the `RTEMS_MONITOR_SUSPEND` bit is set, then the monitor task will suspend itself after it is initialized. A subsequent call to `rtems_monitor_wakeup` will be required to activate it.

NOTES:

The monitor task is created with priority 1. If there are application tasks at priority 1, then there may be times when the monitor task is not executing.

9.4.2 rtems_monitor_wakeup - Wakeup the Monitor Task

CALLING SEQUENCE:

```
void rtems_monitor_wakeup( void );
```

STATUS CODES: NONE

DESCRIPTION:

This routine is used to activate the monitor task if it is suspended.

NOTES:

NONE

9.5 Monitor Interactive Commands

The following commands are supported by the monitor task:

- `help` - Obtain Help
- `pause` - Pause Monitor for a Specified Number of Ticks
- `exit` - Invoke a Fatal RTEMS Error
- `symbol` - Show Entries from Symbol Table
- `continue` - Put Monitor to Sleep Waiting for Explicit Wakeup
- `config` - Show System Configuration
- `itask` - List Init Tasks
- `mpci` - List MPCI Config
- `task` - Show Task Information
- `queue` - Show Message Queue Information
- `extension` - User Extensions
- `driver` - Show Information About Named Drivers
- `dname` - Show Information About Named Drivers
- `object` - Generic Object Information
- `node` - Specify Default Node for Commands That Take IDs

9.5.1 help - Obtain Help

The `help` command prints out the list of commands. If invoked with a command name as the first argument, detailed help information on that command is printed.

9.5.2 pause - Pause Monitor for a Specified Number of Ticks

The `pause` command cause the monitor task to suspend itself for the specified number of ticks. If this command is invoked with no arguments, then the task is suspended for 1 clock tick.

9.5.3 exit - Invoke a Fatal RTEMS Error

The `exit` command invokes `rtems_error_occurred` directive with the specified error code. If this command is invoked with no arguments, then the `rtems_error_occurred` directive is invoked with an arbitrary error code.

9.5.4 symbol - Show Entries from Symbol Table

The `symbol` command lists the specified entries in the symbol table. If this command is invoked with no arguments, then all the symbols in the symbol table are printed.

9.5.5 continue - Put Monitor to Sleep Waiting for Explicit Wakeup

The `continue` command suspends the monitor task with no timeout.

9.5.6 config - Show System Configuration

The `config` command prints the system configuration.

9.5.7 itask - List Init Tasks

The `itask` command lists the tasks in the initialization tasks table.

9.5.8 mpci - List MPCI Config

The `mpci` command shows the MPCI configuration information

9.5.9 task - Show Task Information

The `task` command prints out information about one or more tasks in the system. If invoked with no arguments, then information on all the tasks in the system is printed.

9.5.10 queue - Show Message Queue Information

The `queue` command prints out information about one or more message queues in the system. If invoked with no arguments, then information on all the message queues in the system is printed.

9.5.11 extension - User Extensions

The `extension` command prints out information about the user extensions.

9.5.12 driver - Show Information About Named Drivers

The `driver` command prints information about the device driver table.

9.5.13 dname - Show Information About Named Drivers

The `dname` command prints information about the named device drivers.

9.5.14 object - Generic Object Information

The `object` command prints information about RTEMS objects.

9.5.15 node - Specify Default Node for Commands That Take IDs

The `node` command sets the default node for commands that look at object ID ranges.

Command and Variable Index

There are currently no Command and Variable Index entries.

Concept Index

There are currently no Concept Index entries.

Table of Contents