RTEMS Classic API Guide

Release 5.0.0 (master)

©Copyright 2017, RTEMS Project (built 21st December 2017)

CONTENTS

COPYRIGHT (c) 1988 - 2015.
On-Line Applications Research Corporation (OAR).
COPYRIGHT (c) 2016-2017.
RTEMS Foundation, The RTEMS Documentation Project

Licenses:

Creative Commons Attribution-ShareAlike 4.0 International Public License
https://creativecommons.org/licenses/by-sa/4.0/legalcode

The authors have used their best efforts in preparing this material. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. No warranty of any kind, expressed or implied, with regard to the software or the material contained in this document is provided. No liability arising out of the application or use of any product described in this document is assumed. The authors reserve the right to revise this material and to make changes from time to time in the content hereof without obligation to notify anyone of such revision or changes.

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

RTEMS Online Resources

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

CHAPTER

ONE

KEY CONCEPTS

1.1 Introduction

The facilities provided by RTEMS are built upon a foundation of very powerful concepts. These concepts must be understood before the application developer can efficiently utilize RTEMS. The purpose of this chapter is to familiarize one with these concepts.

1.2 Objects

RTEMS provides directives which can be used to dynamically create, delete, and manipulate a set of predefined object types. These types include tasks, message queues, semaphores, memory regions, memory partitions, timers, ports, and rate monotonic periods. The object-oriented nature of RTEMS encourages the creation of modular applications built upon re-usable "building block" routines.

All objects are created on the local node as required by the application and have an RTEMS assigned ID. All objects have a user-assigned name. Although a relationship exists between an object's name and its RTEMS assigned ID, the name and ID are not identical. Object names are completely arbitrary and selected by the user as a meaningful "tag" which may commonly reflect the object's use in the application. Conversely, object IDs are designed to facilitate efficient object manipulation by the executive.

1.2.1 Object Names

An object name is an unsigned thirty-two bit entity associated with the object by the user. The data type `rtems_name` is used to store object names.

Although not required by RTEMS, object names are often composed of four ASCII characters which help identify that object. For example, a task which causes a light to blink might be called "LITE". The `rtems_build_name` routine is provided to build an object name from four ASCII characters. The following example illustrates this:

```
rtems_name my_name;
my_name = rtems_build_name( 'L', 'I', 'T', 'E' );
```

However, it is not required that the application use ASCII characters to build object names. For example, if an application requires one-hundred tasks, it would be difficult to assign meaningful ASCII names to each task. A more convenient approach would be to name them the binary values one through one-hundred, respectively.

RTEMS provides a helper routine, `rtems_object_get_name`, which can be used to obtain the name of any RTEMS object using just its ID. This routine attempts to convert the name into a printable string.

The following example illustrates the use of this method to print an object name:

```
#include <rtems.h>
#include <rtems/bspIo.h>
void print_name(rtems_id id)
{
    char  buffer[10];   /* name assumed to be 10 characters or less */
    char *result;
    result = rtems_object_get_name( id, sizeof(buffer), buffer );
    printk( "ID=0x%08x name=%s\n", id, ((result) ? result : "no name") );
}
```

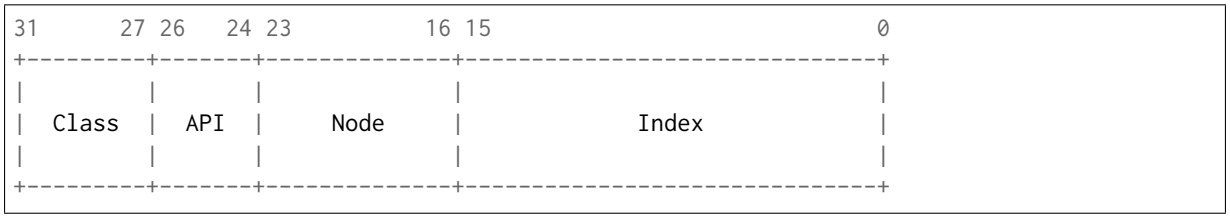
1.2.2 Object IDs

An object ID is a unique unsigned integer value which uniquely identifies an object instance. Object IDs are passed as arguments to many directives in RTEMS and RTEMS translates the ID

to an internal object pointer. The efficient manipulation of object IDs is critical to the performance of RTEMS services. Because of this, there are two object Id formats defined. Each target architecture specifies which format it will use. There is a thirty-two bit format which is used for most of the supported architectures and supports multiprocessor configurations. There is also a simpler sixteen bit format which is appropriate for smaller target architectures and does not support multiprocessor configurations.

1.2.2.1 Thirty-Two Object ID Format

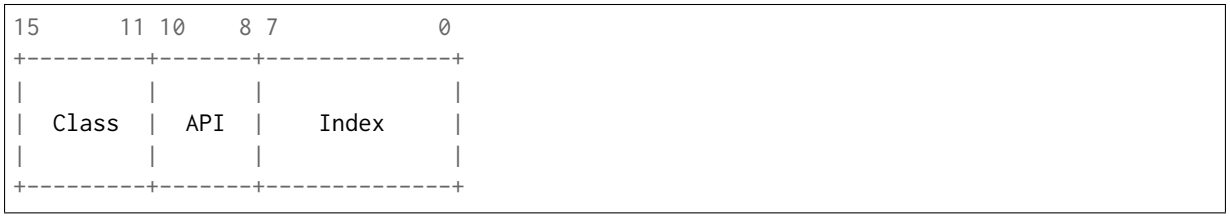
The thirty-two bit format for an object ID is composed of four parts: API, object class, node, and index. The data type `rtems_id` is used to store object IDs.



The most significant five bits are the object class. The next three bits indicate the API to which the object class belongs. The next eight bits (16-23) are the number of the node on which this object was created. The node number is always one (1) in a single processor system. The least significant sixteen bits form an identifier within a particular object type. This identifier, called the object index, ranges in value from 1 to the maximum number of objects configured for this object type.

1.2.2.2 Sixteen Bit Object ID Format

The sixteen bit format for an object ID is composed of three parts: API, object class, and index. The data type `rtems_id` is used to store object IDs.



The sixteen-bit format is designed to be as similar as possible to the thrity-two bit format. The differences are limited to the eliminatation of the node field and reduction of the index field from sixteen-bits to 8-bits. Thus the sixteen bit format only supports up to 255 object instances per API/Class combination and single processor systems. As this format is typically utilized by sixteen-bit processors with limited address space, this is more than enough object instances.

1.2.3 Object ID Description

The components of an object ID make it possible to quickly locate any object in even the most complicated multiprocessor system. Object ID's are associated with an object by RTEMS when the object is created and the corresponding ID is returned by the appropriate object create directive. The object ID is required as input to all directives involving objects, except those which create an object or obtain the ID of an object.

The object identification directives can be used to dynamically obtain a particular object's ID given its name. This mapping is accomplished by searching the name table associated with this object type. If the name is non-unique, then the ID associated with the first occurrence of the name will be returned to the application. Since object IDs are returned when the object is created, the object identification directives are not necessary in a properly designed single processor application.

In addition, services are provided to portably examine the subcomponents of an RTEMS ID. These services are described in detail later in this manual but are prototyped as follows:

```
uint32_t rtems_object_id_get_api( rtems_id );
uint32_t rtems_object_id_get_class( rtems_id );
uint32_t rtems_object_id_get_node( rtems_id );
uint32_t rtems_object_id_get_index( rtems_id );
```

An object control block is a data structure defined by RTEMS which contains the information necessary to manage a particular object type. For efficiency reasons, the format of each object type's control block is different. However, many of the fields are similar in function. The number of each type of control block is application dependent and determined by the values specified in the user's Configuration Table. An object control block is allocated at object create time and freed when the object is deleted. With the exception of user extension routines, object control blocks are not directly manipulated by user applications.

1.3 Communication and Synchronization

In real-time multitasking applications, the ability for cooperating execution threads to communicate and synchronize with each other is imperative. A real-time executive should provide an application with the following capabilities:

- Data transfer between cooperating tasks
- Data transfer between tasks and ISRs
- Synchronization of cooperating tasks
- Synchronization of tasks and ISRs

Most RTEMS managers can be used to provide some form of communication and/or synchronization. However, managers dedicated specifically to communication and synchronization provide well established mechanisms which directly map to the application's varying needs. This level of flexibility allows the application designer to match the features of a particular manager with the complexity of communication and synchronization required. The following managers were specifically designed for communication and synchronization:

- Semaphore
- Message Queue
- Event
- Signal

The semaphore manager supports mutual exclusion involving the synchronization of access to one or more shared user resources. Binary semaphores may utilize the optional priority inheritance algorithm to avoid the problem of priority inversion. The message manager supports both communication and synchronization, while the event manager primarily provides a high performance synchronization mechanism. The signal manager supports only asynchronous communication and is typically used for exception handling.

1.4 Locking Protocols

RTEMS supports the four locking protocols

- *Chapter 1 Section 4.2 - Immediate Ceiling Priority Protocol (ICPP)*,
- *Chapter 1 Section 4.3 - Priority Inheritance Protocol*,
- *Chapter 1 Section 4.4 - Multiprocessor Resource Sharing Protocol (MrsP)*, and
- *Chapter 1 Section 4.5 - O(m) Independence-Preserving Protocol (OMIP)*

for synchronization objects providing mutual-exclusion (mutex). The OMIP is only available in SMP configurations and replaces the priority inheritance protocol in this case. One aim of the locking protocols is to avoid priority inversion.

Since RTEMS 5.1, priority updates due to the locking protocols take place immediately and are propagated recursively. The mutex owner and wait for mutex relationships define a directed acyclic graph (DAG). The run-time of the mutex obtain, release and timeout operations depend on the complexity of this resource dependency graph.

1.4.1 Priority Inversion

Priority inversion is a form of indefinite postponement which is common in multitasking, preemptive executives with shared resources. Priority inversion occurs when a high priority tasks requests access to shared resource which is currently allocated to a low priority task. The high priority task must block until the low priority task releases the resource. This problem is exacerbated when the low priority task is prevented from executing by one or more medium priority tasks. Because the low priority task is not executing, it cannot complete its interaction with the resource and release that resource. The high priority task is effectively prevented from executing by lower priority tasks.

1.4.2 Immediate Ceiling Priority Protocol (ICPP)

Each mutex using the Immediate Ceiling Priority Protocol (ICPP) has a ceiling priority. The priority of the mutex owner is immediately raised to the ceiling priority of the mutex. In case the thread owning the mutex releases the mutex, then the normal priority of the thread is restored. This locking protocol is beneficial for schedulability analysis, see also *[BW01]*.

This protocol avoids the possibility of changing the priority of the mutex owner multiple times since the ceiling priority must be set to the one of highest priority thread which will ever attempt to acquire that mutex. This requires an overall knowledge of the application as a whole. The need to identify the highest priority thread which will attempt to obtain a particular mutex can be a difficult task in a large, complicated system. Although the priority ceiling protocol is more efficient than the priority inheritance protocol with respect to the maximum number of thread priority changes which may occur while a thread owns a particular mutex, the priority inheritance protocol is more forgiving in that it does not require this apriori information.

1.4.3 Priority Inheritance Protocol

The priority of the mutex owner is raised to the highest priority of all threads that currently wait for ownership of this mutex *[SRL90]*. Since RTEMS 5.1, priority updates due to the priority

inheritance protocol take place immediately and are propagated recursively.

1.4.4 Multiprocessor Resource Sharing Protocol (MrsP)

The Multiprocessor Resource Sharing Protocol (MrsP) is a generalization of the priority ceiling protocol to clustered scheduling *[BW13]*. One of the design goals of MrsP is to enable an effective schedulability analysis using the sporadic task model. Each mutex using the MrsP has a ceiling priority for each scheduler instance. The priority of the mutex owner is immediately raised to the ceiling priority of the mutex defined for its home scheduler instance. In case the thread owning the mutex releases the mutex, then the normal priority of the thread is restored. Threads that wait for mutex ownership are not blocked with respect to the scheduler and instead perform a busy wait. The MrsP uses temporary thread migrations to foreign scheduler instances in case of a preemption of the mutex owner. This locking protocol is available since RTEMS 4.11. It was re-implemented in RTEMS 5.1 to overcome some shortcomings of the original implementation *[CBHM15]*.

1.4.5 O(m) Independence-Preserving Protocol (OMIP)

The $O(m)$ Independence-Preserving Protocol (OMIP) is a generalization of the priority inheritance protocol to clustered scheduling which avoids the non-preemptive sections present with priority boosting *[Bra13]*. The m denotes the number of processors in the system. Similar to the uni-processor priority inheritance protocol, the OMIP mutexes do not need any external configuration data, e.g. a ceiling priority. This makes them a good choice for general purpose libraries that need internal locking. The complex part of the implementation is contained in the thread queues and shared with the MrsP support. This locking protocol is available since RTEMS 5.1.

1.5 Thread Queues

In case more than one *thread* may wait on a synchronization object, e.g. a semaphore or a message queue, then the waiting threads are added to a data structure called the thread queue. Thread queues are named task wait queues in the Classic API. There are two thread queuing disciplines available which define the order of the threads on a particular thread queue. Threads can wait in FIFO or priority order.

In uni-processor configurations, the priority queuing discipline just orders the threads according to their current priority and in FIFO order in case of equal priorities. However, in SMP configurations, the situation is a bit more difficult due to the support for clustered scheduling. It makes no sense to compare the priority values of two different scheduler instances. Thus, it is impossible to simply use one plain priority queue for threads of different clusters. Two levels of queues can be used as one way to solve the problem. The top-level queue provides FIFO ordering and contains priority queues. Each priority queue is associated with a scheduler instance and contains only threads of this scheduler instance. Threads are enqueued in the priority queues corresponding to their scheduler instances. To dequeue a thread, the highest priority thread of the first priority queue is selected. Once this is done, the first priority queue is appended to the top-level FIFO queue. This guarantees fairness with respect to the scheduler instances.

Such a two-level queue needs a considerable amount of memory if fast enqueue and dequeue operations are desired. Providing this storage per thread queue would waste a lot of memory in typical applications. Instead, each thread has a queue attached which resides in a dedicated memory space independent of other memory used for the thread (this approach was borrowed from FreeBSD). In case a thread needs to block, there are two options

- the object already has a queue, then the thread enqueues itself to this already present queue and the queue of the thread is added to a list of free queues for this object, or
- otherwise, the queue of the thread is given to the object and the thread enqueues itself to this queue.

In case the thread is dequeued, there are two options

- the thread is the last thread in the queue, then it removes this queue from the object and reclaims it for its own purpose, or
- otherwise, the thread removes one queue from the free list of the object and reclaims it for its own purpose.

Since there are usually more objects than threads, this actually reduces the memory demands. In addition the objects only contain a pointer to the queue structure. This helps to hide implementation details. Inter-cluster priority queues are available since RTEMS 5.1.

A doubly-linked list (chain) is used to implement the FIFO queues yielding a $O(1)$ worst-case time complexity for enqueue and dequeue operations.

A red-black tree is used to implement the priority queues yielding a $O(log(n))$ worst-case time complexity for enqueue and dequeue operations with n being the count of threads already on the queue.

1.6 Time

The development of responsive real-time applications requires an understanding of how RTEMS maintains and supports time-related operations. The basic unit of time in RTEMS is known as a *clock tick* or simply *tick*. The tick interval is defined by the application configuration option CONFIGURE_MICROSECONDS_PER_TICK. The tick interval defines the basic resolution of all interval and calendar time operations. Obviously, the directives which use intervals or wall time cannot operate without some external mechanism which provides a periodic clock tick. This clock tick is provided by the clock driver. The tick precision and stability depends on the clock driver and interrupt latency. Most clock drivers provide a timecounter to measure the time with a higher resolution than the tick.

By tracking time in units of ticks, RTEMS is capable of supporting interval timing functions such as task delays, timeouts, timeslicing, the delayed execution of timer service routines, and the rate monotonic scheduling of tasks. An interval is defined as a number of ticks relative to the current time. For example, when a task delays for an interval of ten ticks, it is implied that the task will not execute until ten clock ticks have occurred. All intervals are specified using data type `rtems_interval`.

A characteristic of interval timing is that the actual interval period may be a fraction of a tick less than the interval requested. This occurs because the time at which the delay timer is set up occurs at some time between two clock ticks. Therefore, the first countdown tick occurs in less than the complete time interval for a tick. This can be a problem if the tick resolution is large.

The rate monotonic scheduling algorithm is a hard real-time scheduling methodology. This methodology provides rules which allows one to guarantee that a set of independent periodic tasks will always meet their deadlines even under transient overload conditions. The rate monotonic manager provides directives built upon the Clock Manager's interval timer support routines.

Interval timing is not sufficient for the many applications which require that time be kept in wall time or true calendar form. Consequently, RTEMS maintains the current date and time. This allows selected time operations to be scheduled at an actual calendar date and time. For example, a task could request to delay until midnight on New Year's Eve before lowering the ball at Times Square. The data type `rtems_time_of_day` is used to specify calendar time in RTEMS services. See Time and Date Data Structures.

1.7 Timer and Timeouts

Timer and timeout services are a standard component of an operating system. The use cases fall roughly into two categories:

- Timeouts – used to detect if some operations need more time than expected. Since the unexpected happens hopefully rarely, timeout timers are usually removed before they expire. The critical operations are insert and removal. For example, they are important for the performance of a network stack.
- Timers – used to carry out some work in the future. They usually expire and need a high resolution. An example use case is a time driven scheduler, e.g. rate-monotonic or EDF.

In RTEMS versions prior to 5.1 the timer and timeout support was implemented by means of delta chains. This implementation was unfit for SMP systems due to several reasons. The new implementation present since RTEMS 5.1 uses a red-black tree with the expiration time as the key. This leads to $O(log(n))$ worst-case insert and removal operations for n active timer or timeouts. Each processor provides its own timer and timeout service point so that it scales well with the processor count of the system. For each operation it is sufficient to acquire and release a dedicated SMP lock only once. The drawback is that a 64-bit integer type is required internally for the intervals to avoid a potential overflow of the key values.

An alternative to the red-black tree based implementation would be the use of a timer wheel based algorithm *[VL87]* which is used in Linux and FreeBSD *[VC95]* for example. A timer wheel based algorithm offers $O(1)$ worst-case time complexity for insert and removal operations. The drawback is that the run-time of the clock tick procedure is unpredictable due to the use of a hash table or cascading.

The red-black tree approach was selected for RTEMS, since it offers a more predictable run-time behaviour. However, this sacrifices the constant insert and removal operations offered by the timer wheel algorithms. See also *[GN06]*. The implementation can re-use the red-black tree support already used in other areas, e.g. for the thread priority queues. Less code is a good thing for size, testing and verification.

1.8 Memory Management

RTEMS memory management facilities can be grouped into two classes: dynamic memory allocation and address translation. Dynamic memory allocation is required by applications whose memory requirements vary through the application's course of execution. Address translation is needed by applications which share memory with another CPU or an intelligent Input/Output processor. The following RTEMS managers provide facilities to manage memory:

- Region
- Partition
- Dual Ported Memory

RTEMS memory management features allow an application to create simple memory pools of fixed size buffers and/or more complex memory pools of variable size segments. The partition manager provides directives to manage and maintain pools of fixed size entities such as resource control blocks. Alternatively, the region manager provides a more general purpose memory allocation scheme that supports variable size blocks of memory which are dynamically obtained and freed by the application. The dual-ported memory manager provides executive support for address translation between internal and external dual-ported RAM address space.

CHAPTER

TWO

SELF-CONTAINED OBJECTS

2.1 Introduction

One of the original design goals of RTEMS was the support for heterogeneous computing based on message passing. This was realized by synchronization objects with an architecture-independent identifier provided by the system during object creation (a 32-bit unsigned integer used as a bitfield) and a user-defined four character name. This approach in the so called Classic API has some weaknesses:

- Dynamic memory (the workspace) is used to allocate object pools. This requires a complex configuration with heavy use of the C pre-processor. The unlimited objects support optionally expands and shrinks the object pool. Dynamic memory is strongly discouraged by some coding standards, e.g. MISRA C:2012 *[BBB+13]*.
- Objects are created via function calls which return an object identifier. The object operations use this identifier and map it internally to an object representation.
- The object identifier is only known at run-time. This hinders compiler optimizations and static analysis.
- The objects reside in a table, e.g. they are suspect to false sharing of cache lines *[Dre07]*.
- The object operations use a rich set of options and attributes. For each object operation these parameters must be evaluated and validated at run-time to figure out what to do exactly for this operation.

For applications that use fine grained locking the mapping of the identifier to the object representation and the parameter evaluation are a significant overhead that may degrade the performance dramatically. An example is the new network stack (libbsd) which uses hundreds of locks in a basic setup. Another example is the OpenMP support (libgomp).

To overcome these issues new self-contained synchronization objects are available since RTEMS 4.11. Self-contained synchronization objects encapsulate all their state in exactly one data structure. The user must provide the storage space for this structure and nothing more. The user is responsible for the object life-cycle. Initialization and destruction of self-contained synchronization objects cannot fail provided all function parameters are valid. In particular, a not enough memory error cannot happen. It is possible to statically initialize self-contained synchronization objects. This allows an efficient use of static analysis tools.

Several header files define self-contained synchronization objects. The Newlib `<sys/lock.h>` header file provides

- mutexes,
- recursive mutexes,
- condition variables,
- counting semaphores,
- binary semaphores, and
- Futex synchronization *[FRK02]*.

They are used internally in Newlib (e.g. for FILE objects), for the C++11 threads and the OpenMP support (libgomp). The Newlib provided self-contained synchronization objects focus on performance. There are no error checks to catch software errors, e.g. invalid parameters. The application configuration is significantly simplified, since it is no longer necessary to

account for lock objects used by Newlib and GCC. The Newlib defined self-contained synchronization objects can be a statically initialized and reside in the `.bss` section. Destruction is a no-operation.

The header file `<pthread.h>` provides

- POSIX barriers (`pthread_barrier_t`),
- POSIX condition variables (`pthread_cond_t`),
- POSIX mutexes (`pthread_mutex_t`),
- POSIX reader/writer locks (`pthread_rwlock_t`), and
- POSIX spinlocks (`pthread_spinlock_t`)

as self-contained synchronization objects. The POSIX synchronization objects are used for example by the Ada run-time support. The header file `<semaphore.h>` provides self-contained

- POSIX unnamed semaphores (`sem_t` initialized via `sem_init()`).

2.2 RTEMS Thread API

To give RTEMS users access to self-contained synchronization objects an API is necessary. One option would be to simply use the POSIX threads API (pthreads), C11 threads or C++11 threads. However, these standard APIs lack for example binary semaphores which are important for task/interrupt synchronization. The timed operations use in general time values specified by seconds and nanoseconds. Setting up the time values in seconds (time_t has 64 bits) and nanoseconds is burdened with a high overhead compared to time values in clock ticks for relative timeouts. The POSIX API mutexes can be configured for various protocols and options, this adds a run-time overhead. There are a variety of error conditions. This is a problem in combination with some coding standards, e.g. MISRA C:2012. APIs used by Linux (e.g. <linux/mutex.h>) or the FreeBSD kernel (e.g. MUTEX(9)) are better suited as a template for high-performance synchronization objects. The goal of the *RTEMS Thread API* is to offer the highest performance with the lowest space-overhead on RTEMS. It should be suitable for device drivers.

2.3 Mutual Exclusion

The rtems_mutex and rtems_recursive_mutex objects provide mutual-exclusion synchronization using the *Chapter 1 Section 4.3 - Priority Inheritance Protocol* in uni-processor configurations or the *Chapter 1 Section 4.5 - O(m) Independence-Preserving Protocol (OMIP)* in SMP configurations. Recursive locking should be used with care *[Wil12]*. The storage space for these object must be provided by the user. There are no defined comparison or assignment operators for these type. Only the object itself may be used for performing synchronization. The result of referring to copies of the object in calls to

- rtems_mutex_lock(),
- rtems_recursive_mutex_lock(),
- rtems_mutex_unlock(),
- rtems_recursive_mutex_unlock(),
- rtems_mutex_set_name(),
- rtems_recursive_mutex_set_name(),
- rtems_mutex_get_name(),
- rtems_recursive_mutex_get_name(),
- rtems_mutex_destroy(), and
- rtems_recursive_mutex_destroy()

is undefined. Objects of the type rtems_mutex must be initialized via

- RTEMS_MUTEX_INITIALIZER(), or
- rtems_mutex_init().

They must be destroyed via

- rtems_mutex_destroy().

is undefined. Objects of the type rtems_recursive_mutex must be initialized via

- RTEMS_RECURSIVE_MUTEX_INITIALIZER(), or
- rtems_recursive_mutex_init().

They must be destroyed via

- rtems_recursive_mutex_destroy().

2.3.1 Static mutex initialization

CALLING SEQUENCE:

```
rtems_mutex mutex = RTEMS_MUTEX_INITIALIZER(
  name
);

rtems_recursive_mutex mutex = RTEMS_RECURSIVE_MUTEX_INITIALIZER(
  name
);
```

DESCRIPTION:

An initializer for static initialization. It is equivalent to a call to `rtems_mutex_init()` or `rtems_recursive_mutex_init()` respectively.

NOTES:

Global mutexes with a `name` of `NULL` may reside in the `.bss` section.

2.3.2 Run-time mutex initialization

CALLING SEQUENCE:

```
void rtems_mutex_init(
  rtems_mutex *mutex,
  const char  *name
);

void rtems_recursive_mutex_init(
  rtems_recursive_mutex *mutex,
  const char            *name
);
```

DESCRIPTION:

Initializes the `mutex` with the specified `name`.

NOTES:

The `name` must be persistent throughout the life-time of the mutex. A `name` of `NULL` is valid. The mutex is unlocked after initialization.

2.3.3 Lock the mutex

CALLING SEQUENCE:

```
void rtems_mutex_lock(
  rtems_mutex *mutex
);

void rtems_recursive_mutex_lock(
  rtems_recursive_mutex *mutex
);
```

DESCRIPTION:

Locks the `mutex`.

NOTES:

This function must be called from thread context with interrupts enabled. In case the mutex is currently locked by another thread, then the thread is blocked until it becomes the mutex owner. Threads wait in priority order.

A recursive lock happens in case the mutex owner tries to lock the mutex again. The result of recursively locking a mutex depends on the mutex variant. For a normal (non-recursive) mutex (`rtems_mutex`) the result is unpredictable. It could block the owner indefinetly or lead to a fatal deadlock error. A recursive mutex (`rtems_recursive_mutex`) can be locked recursively by the mutex owner.

Each mutex lock operation must have a corresponding unlock operation.

2.3.4 Unlock the mutex

CALLING SEQUENCE:

```
void rtems_mutex_unlock(
  rtems_mutex *mutex
);

void rtems_recursive_mutex_unlock(
  rtems_recursive_mutex *mutex
);
```

DESCRIPTION:
Unlocks the `mutex`.

NOTES:
This function must be called from thread context with interrupts enabled. In case the currently executing thread is not the owner of the `mutex`, then the result is unpredictable.

Exactly the outer-most unlock will make a recursive mutex available to other threads.

2.3.5 Set mutex name

CALLING SEQUENCE:

```
void rtems_mutex_set_name(
  rtems_mutex *mutex,
  const char  *name
);

void rtems_recursive_mutex_set_name(
  rtems_recursive_mutex *mutex,
  const char            *name
);
```

DESCRIPTION:

Sets the `mutex` name to `name`.

NOTES:

The `name` must be persistent throughout the life-time of the mutex. A `name` of `NULL` is valid.

2.3.6 Get mutex name

CALLING SEQUENCE:

```
const char *rtems_mutex_get_name(
  const rtems_mutex *mutex
);

const char *rtems_recursive_mutex_get_name(
  const rtems_recursive_mutex *mutex
);
```

DESCRIPTION:
Returns the `mutex` name.

NOTES:
The name may be `NULL`.

2.3.7 Mutex destruction

CALLING SEQUENCE:

```
void rtems_mutex_destroy(
  rtems_mutex *mutex
);

void rtems_recursive_mutex_destroy(
  rtems_recursive_mutex *mutex
);
```

DESCRIPTION:
Destroys the `mutex`.

NOTES:
In case the mutex is locked or still in use, then the result is unpredictable.

2.4 Condition Variables

The `rtems_condition_variable` object provides a condition variable synchronization object. The storage space for this object must be provided by the user. There are no defined comparison or assignment operators for this type. Only the object itself may be used for performing synchronization. The result of referring to copies of the object in calls to

- `rtems_condition_variable_wait()`,
- `rtems_condition_variable_signal()`,
- `rtems_condition_variable_broadcast()`,
- `rtems_condition_variable_set_name()`,
- `rtems_condition_variable_get_name()`, and
- `rtems_condition_variable_destroy()`

is undefined. Objects of this type must be initialized via

- `RTEMS_CONDITION_VARIABLE_INITIALIZER()`, or
- `rtems_condition_variable_init()`.

They must be destroyed via

- `rtems_condition_variable_destroy()`.

2.4.1 Static condition variable initialization

CALLING SEQUENCE:

```
rtems_condition_variable condition_variable = RTEMS_CONDITION_VARIABLE_INITIALIZER(
  name
);
```

DESCRIPTION:

An initializer for static initialization. It is equivalent to a call to `rtems_condition_variable_init()`.

NOTES:

Global condition variables with a `name` of `NULL` may reside in the `.bss` section.

2.4.2 Run-time condition variable initialization

CALLING SEQUENCE:

```
void rtems_condition_variable_init(
  rtems_condition_variable *condition_variable,
  const char               *name
);
```

DESCRIPTION:
Initializes the `condition_variable` with the specified `name`.

NOTES:
The `name` must be persistent throughout the life-time of the condition variable. A `name` of `NULL` is valid.

2.4.3 Wait for condition signal

CALLING SEQUENCE:

```
void rtems_condition_variable_wait(
  rtems_condition_variable *condition_variable,
  rtems_mutex              *mutex
);
```

DESCRIPTION:

Atomically waits for a condition signal and unlocks the mutex. Once the condition is signalled to the thread it wakes up and locks the mutex again.

NOTES:

This function must be called from thread context with interrupts enabled. Threads wait in priority order.

2.4.4 Signals a condition change

CALLING SEQUENCE:

```
void rtems_condition_variable_signal(
  rtems_condition_variable *condition_variable
);
```

DESCRIPTION:

Signals a condition change to the highest priority waiting thread. If no threads wait currently on this condition variable, then nothing happens.

2.4.5 Broadcasts a condition change

CALLING SEQUENCE:

```
void rtems_condition_variable_broadcast(
  rtems_condition_variable *condition_variable
);
```

DESCRIPTION:

Signals a condition change to all waiting thread. If no threads wait currently on this condition variable, then nothing happens.

2.4.6 Set condition variable name

CALLING SEQUENCE:

```
void rtems_condition_variable_set_name(
  rtems_condition_variable *condition_variable,
  const char               *name
);
```

DESCRIPTION:
Sets the `condition_variable` name to `name`.

NOTES:
The `name` must be persistent throughout the life-time of the condition variable. A `name` of `NULL` is valid.

2.4.7 Get condition variable name

CALLING SEQUENCE:

```
const char *rtems_condition_variable_get_name(
  const rtems_condition_variable *condition_variable
);
```

DESCRIPTION:
Returns the `condition_variable` name.

NOTES:
The name may be `NULL`.

2.4.8 Condition variable destruction

CALLING SEQUENCE:

```
void rtems_condition_variable_destroy(
  rtems_condition_variable *condition_variable
);
```

DESCRIPTION:
Destroys the `condition_variable`.

NOTES:
In case the condition variable still in use, then the result is unpredictable.

2.5 Counting Semaphores

The rtems_counting_semaphore object provides a counting semaphore synchronization object. The storage space for this object must be provided by the user. There are no defined comparison or assignment operators for this type. Only the object itself may be used for performing synchronization. The result of referring to copies of the object in calls to

- rtems_counting_semaphore_wait(),
- rtems_counting_semaphore_post(),
- rtems_counting_semaphore_set_name(),
- rtems_counting_semaphore_get_name(), and
- rtems_counting_semaphore_destroy()

is undefined. Objects of this type must be initialized via

- RTEMS_COUNTING_SEMAPHORE_INITIALIZER(), or
- rtems_counting_semaphore_init().

They must be destroyed via

- rtems_counting_semaphore_destroy().

2.5.1 Static counting semaphore initialization

CALLING SEQUENCE:

```
rtems_counting_semaphore counting_semaphore = RTEMS_COUNTING_SEMAPHORE_INITIALIZER(
  name,
  value
);
```

DESCRIPTION:

An initializer for static initialization. It is equivalent to a call to `rtems_counting_semaphore_init()`.

NOTES:

Global counting semaphores with a `name` of `NULL` may reside in the `.bss` section.

2.5.2 Run-time counting semaphore initialization

CALLING SEQUENCE:

```
void rtems_counting_semaphore_init(
  rtems_counting_semaphore *counting_semaphore,
  const char               *name,
  unsigned int              value
);
```

DESCRIPTION:

Initializes the `counting_semaphore` with the specified `name` and `value`. The initial value is set to `value`.

NOTES:

The `name` must be persistent throughout the life-time of the counting semaphore. A `name` of `NULL` is valid.

2.5.3 Wait for a counting semaphore

CALLING SEQUENCE:

```
void rtems_counting_semaphore_wait(
  rtems_counting_semaphore *counting_semaphore
);
```

DESCRIPTION:

Waits for the `counting_semaphore`. In case the current semaphore value is positive, then the value is decremented and the function returns immediately, otherwise the thread is blocked waiting for a semaphore post.

NOTES:

This function must be called from thread context with interrupts enabled. Threads wait in priority order.

2.5.4 Post a counting semaphore

CALLING SEQUENCE:

```
void rtems_counting_semaphore_post(
  rtems_counting_semaphore *counting_semaphore
);
```

DESCRIPTION:

Posts the `counting_semaphore`. In case at least one thread is waiting on the counting semaphore, then the highest priority thread is woken up, otherwise the current value is incremented.

NOTES:

This function may be called from interrupt context. In case it is called from thread context, then interrupts must be enabled.

2.5.5 Set counting semaphore name

CALLING SEQUENCE:

```
void rtems_counting_semaphore_set_name(
  rtems_counting_semaphore *counting_semaphore,
  const char               *name
);
```

DESCRIPTION:
Sets the `counting_semaphore` name to `name`.

NOTES:
The `name` must be persistent throughout the life-time of the counting semaphore. A `name` of `NULL` is valid.

2.5.6 Get counting semaphore name

CALLING SEQUENCE:

```
const char *rtems_counting_semaphore_get_name(
  const rtems_counting_semaphore *counting_semaphore
);
```

DESCRIPTION:
Returns the `counting_semaphore` name.

NOTES:
The name may be `NULL`.

2.5.7 Counting semaphore destruction

CALLING SEQUENCE:

```
void rtems_counting_semaphore_destroy(
  rtems_counting_semaphore *counting_semaphore
);
```

DESCRIPTION:
Destroys the `counting_semaphore`.

NOTES:
In case the counting semaphore still in use, then the result is unpredictable.

2.6 Binary Semaphores

The `rtems_binary_semaphore` object provides a binary semaphore synchronization object. The storage space for this object must be provided by the user. There are no defined comparison or assignment operators for this type. Only the object itself may be used for performing synchronization. The result of referring to copies of the object in calls to

- `rtems_binary_semaphore_wait()`,
- `rtems_binary_semaphore_wait_timed_ticks()`,
- `rtems_binary_semaphore_try_wait()`,
- `rtems_binary_semaphore_post()`,
- `rtems_binary_semaphore_set_name()`,
- `rtems_binary_semaphore_get_name()`, and
- `rtems_binary_semaphore_destroy()`

is undefined. Objects of this type must be initialized via

- `RTEMS_BINARY_SEMAPHORE_INITIALIZER()`, or
- `rtems_binary_semaphore_init()`.

They must be destroyed via

- `rtems_binary_semaphore_destroy()`.

2.6.1 Static binary semaphore initialization

CALLING SEQUENCE:

```
rtems_binary_semaphore binary_semaphore = RTEMS_BINARY_SEMAPHORE_INITIALIZER(
  name
);
```

DESCRIPTION:

An initializer for static initialization. It is equivalent to a call to `rtems_binary_semaphore_init()`.

NOTES:

Global binary semaphores with a `name` of `NULL` may reside in the `.bss` section.

2.6.2 Run-time binary semaphore initialization

CALLING SEQUENCE:

```
void rtems_binary_semaphore_init(
  rtems_binary_semaphore *binary_semaphore,
  const char             *name
);
```

DESCRIPTION:
Initializes the `binary_semaphore` with the specified `name`. The initial value is set to zero.

NOTES:
The `name` must be persistent throughout the life-time of the binary semaphore. A `name` of `NULL` is valid.

2.6.3 Wait for a binary semaphore

CALLING SEQUENCE:

```
void rtems_binary_semaphore_wait(
  rtems_binary_semaphore *binary_semaphore
);
```

DESCRIPTION:

Waits for the `binary_semaphore`. In case the current semaphore value is one, then the value is set to zero and the function returns immediately, otherwise the thread is blocked waiting for a semaphore post.

NOTES:

This function must be called from thread context with interrupts enabled. Threads wait in priority order.

2.6.4 Wait for a binary semaphore with timeout in ticks

CALLING SEQUENCE:

```
int rtems_binary_semaphore_wait_timed_ticks(
  rtems_binary_semaphore *binary_semaphore,
  uint32_t               ticks
);
```

DIRECTIVE STATUS CODES:

`0`	The semaphore wait was successful.
`ETIMEDOUT`	The semaphore wait timed out.

DESCRIPTION:

Waits for the `binary_semaphore` with a timeout in `ticks`. In case the current semaphore value is one, then the value is set to zero and the function returns immediately with a return value of `0`, otherwise the thread is blocked waiting for a semaphore post. The time waiting for a semaphore post is limited by `ticks`. A `ticks` value of zero specifies an infinite timeout.

NOTES:

This function must be called from thread context with interrupts enabled. Threads wait in priority order.

2.6.5 Tries to wait for a binary semaphore

CALLING SEQUENCE:

```
int rtems_binary_semaphore_try_wait(
  rtems_binary_semaphore *binary_semaphore
);
```

DIRECTIVE STATUS CODES:

`0`	The semaphore wait was successful.
`EAGAIN`	The semaphore wait failed.

DESCRIPTION:
Tries to wait for the `binary_semaphore`. In case the current semaphore value is one, then the value is set to zero and the function returns immediately with a return value of `0`, otherwise it returns immediately with a return value of `EAGAIN`.

NOTES:
This function may be called from interrupt context. In case it is called from thread context, then interrupts must be enabled.

2.6.6 Post a binary semaphore

CALLING SEQUENCE:

```
void rtems_binary_semaphore_post(
  rtems_binary_semaphore *binary_semaphore
);
```

DESCRIPTION:

Posts the `binary_semaphore`. In case at least one thread is waiting on the binary semaphore, then the highest priority thread is woken up, otherwise the current value is set to one.

NOTES:

This function may be called from interrupt context. In case it is called from thread context, then interrupts must be enabled.

2.6.7 Set binary semaphore name

CALLING SEQUENCE:

```
void rtems_binary_semaphore_set_name(
  rtems_binary_semaphore *binary_semaphore,
  const char             *name
);
```

DESCRIPTION:
Sets the `binary_semaphore` name to `name`.

NOTES:
The `name` must be persistent throughout the life-time of the binary semaphore. A `name` of `NULL` is valid.

2.6.8 Get binary semaphore name

CALLING SEQUENCE:

```
const char *rtems_binary_semaphore_get_name(
  const rtems_binary_semaphore *binary_semaphore
);
```

DESCRIPTION:
Returns the `binary_semaphore` name.

NOTES:
The name may be `NULL`.

2.6.9 Binary semaphore destruction

CALLING SEQUENCE:

```
void rtems_binary_semaphore_destroy(
  rtems_binary_semaphore *binary_semaphore
);
```

DESCRIPTION:
Destroys the `binary_semaphore`.

NOTES:
In case the binary semaphore still in use, then the result is unpredictable.

2.7 Threads

Warning: The self-contained threads support is work in progress. In contrast to the synchronization objects the self-contained thread support is not just an API glue layer to already existing implementations.

The rtems_thread object provides a thread of execution.

CALLING SEQUENCE:

```
RTEMS_THREAD_INITIALIZER(
  name,
  thread_size,
  priority,
  flags,
  entry,
  arg
);

void rtems_thread_start(
  rtems_thread *thread,
  const char   *name,
  size_t        thread_size,
  uint32_t      priority,
  uint32_t      flags,
  void       ( *entry )( void * ),
  void         *arg
);

void rtems_thread_restart(
  rtems_thread *thread,
  void         *arg
) RTEMS_NO_RETURN;

void rtems_thread_event_send(
  rtems_thread *thread,
  uint32_t      events
);

uint32_t rtems_thread_event_poll(
  rtems_thread *thread,
  uint32_t      events_of_interest
);

uint32_t rtems_thread_event_wait_all(
  rtems_thread *thread,
  uint32_t      events_of_interest
);

uint32_t rtems_thread_event_wait_any(
  rtems_thread *thread,
  uint32_t      events_of_interest
);

```

```
45 void rtems_thread_destroy(
46   rtems_thread *thread
47 );
48
49 void rtems_thread_destroy_self(
50   void
51 ) RTEMS_NO_RETURN;
```

CHAPTER

THREE

GLOSSARY

active
A term used to describe an object which has been created by an application.

aperiodic task
A task which must execute only at irregular intervals and has only a soft deadline.

API
An acronym for Application Programming Interface.

application
In this document, software which makes use of RTEMS.

ASR
see Asynchronous Signal Routine.

asynchronous
Not related in order or timing to other occurrences in the system.

Asynchronous Signal Routine
Similar to a hardware interrupt except that it is associated with a task and is run in the context of a task. The directives provided by the signal manager are used to service signals.

atomic operations
Atomic operations are defined in terms of *C11*.

awakened
A term used to describe a task that has been unblocked and may be scheduled to the CPU.

big endian
A data representation scheme in which the bytes composing a numeric value are arranged such that the most significant byte is at the lowest address.

bit-mapped
A data encoding scheme in which each bit in a variable is used to represent something different. This makes for compact data representation.

block
A physically contiguous area of memory.

blocked task
The task state entered by a task which has been previously started and cannot continue execution until the reason for waiting has been satisfied. Blocked tasks are not an element of the set of ready tasks of a scheduler instance.

Board Support Package

BSP
A collection of device initialization and control routines specific to a particular type of board or collection of boards.

broadcast
To simultaneously send a message to a logical set of destinations.

buffer
A fixed length block of memory allocated from a partition.

C++11
The standard ISO/IEC 14882:2011.

C11
The standard ISO/IEC 9899:2011.

calling convention
The processor and compiler dependent rules which define the mechanism used to invoke subroutines in a high-level language. These rules define the passing of arguments, the call and return mechanism, and the register set which must be preserved.

Central Processing Unit
This term is equivalent to the terms processor and microprocessor.

chain
A data structure which allows for efficient dynamic addition and removal of elements. It differs from an array in that it is not limited to a predefined size.

cluster
We have clustered scheduling in case the set of processors of a system is partitioned into non-empty pairwise disjoint subsets. These subsets are calledclusters. Clusters with a cardinality of one are partitions. Each cluster is owned by exactly one scheduler instance.

coalesce
The process of merging adjacent holes into a single larger hole. Sometimes this process is referred to as garbage collection.

Configuration Table
A table which contains information used to tailor RTEMS for a particular application.

context
All of the processor registers and operating system data structures associated with a task.

context switch
Alternate term for task switch. Taking control of the processor from one task and transferring it to another task.

control block
A data structure used by the executive to define and control an object.

core
When used in this manual, this term refers to the internal executive utility functions. In the interest of application portability, the core of the executive should not be used directly by applications.

CPU
An acronym for Central Processing Unit.

critical section
A section of code which must be executed indivisibly.

CRT
An acronym for Cathode Ray Tube. Normally used in reference to the man-machine interface.

deadline
A fixed time limit by which a task must have completed a set of actions. Beyond this point, the results are of reduced value and may even be considered useless or harmful.

device
A peripheral used by the application that requires special operation software. See also device driver.

device driver
Control software for special peripheral devices used by the application.

Device Driver Table
A table which contains the entry points for each of the configured device drivers.

directives
RTEMS' provided routines that provide support mechanisms for real-time applications.

dispatch
The act of loading a task's context onto the CPU and transferring control of the CPU to that task.

dormant
The state entered by a task after it is created and before it has been started.

dual-ported
A term used to describe memory which can be accessed at two different addresses.

embedded
An application that is delivered as a hidden part of a larger system. For example, the software in a fuel-injection control system is an embedded application found in many late-model automobiles.

entry point
The address at which a function or task begins to execute. In C, the entry point of a function is the function's name.

envelope
A buffer provided by the MPCI layer to RTEMS which is used to pass messages between nodes in a multiprocessor system. It typically contains routing information needed by the MPCI. The contents of an envelope are referred to as a packet.

events
A method for task communication and synchronization. The directives provided by the event manager are used to service events.

exception
A synonym for interrupt.

executing task
The task state entered by a task after it has been given control of the processor. In SMP

configurations, a task may be registered as executing on more than one processor for short time frames during task migration. Blocked tasks can be executing until they issue a thread dispatch.

executive
In this document, this term is used to referred to RTEMS. Commonly, an executive is a small real-time operating system used in embedded systems.

exported
An object known by all nodes in a multiprocessor system. An object created with the GLOBAL attribute will be exported.

external address
The address used to access dual-ported memory by all the nodes in a system which do not own the memory.

FIFO
An acronym for First In First Out.

First In First Out
A discipline for manipulating entries in a data structure.

floating point coprocessor
A component used in computer systems to enhance performance in mathematically intensive situations. It is typically viewed as a logical extension of the primary processor.

freed
A resource that has been released by the application to RTEMS.

global
An object that has been created with the GLOBAL attribute and exported to all nodes in a multiprocessor system.

handler
The equivalent of a manager, except that it is internal to RTEMS and forms part of the core. A handler is a collection of routines which provide a related set of functions. For example, there is a handler used by RTEMS to manage all objects.

hard real-time system
A real-time system in which a missed deadline causes the worked performed to have no value or to result in a catastrophic effect on the integrity of the system.

heap
A data structure used to dynamically allocate and deallocate variable sized blocks of memory.

heir task
A task is an heir if it is registered as an heir in a processor of the system. A task can be the heir on at most one processor in the system. In case the executing and heir tasks differ on a processor and a thread dispatch is marked as necessary, then the next thread dispatch will make the heir task the executing task.

heterogeneous
A multiprocessor computer system composed of dissimilar processors.

homogeneous
A multiprocessor computer system composed of a single type of processor.

I/O
An acronym for Input/Output.

ID
An RTEMS assigned identification tag used to access an active object.

IDLE task
A special low priority task which assumes control of the CPU when no other task is able to execute.

interface
A specification of the methodology used to connect multiple independent subsystems.

internal address
The address used to access dual-ported memory by the node which owns the memory.

interrupt
A hardware facility that causes the CPU to suspend execution, save its status, and transfer control to a specific location.

interrupt level
A mask used to by the CPU to determine which pending interrupts should be serviced. If a pending interrupt is below the current interrupt level, then the CPU does not recognize that interrupt.

Interrupt Service Routine
An ISR is invoked by the CPU to process a pending interrupt.

ISR
An acronym for Interrupt Service Routine.

kernel
In this document, this term is used as a synonym for executive.

list
A data structure which allows for dynamic addition and removal of entries. It is not statically limited to a particular size.

little endian
A data representation scheme in which the bytes composing a numeric value are arranged such that the least significant byte is at the lowest address.

local
An object which was created with the LOCAL attribute and is accessible only on the node it was created and resides upon. In a single processor configuration, all objects are local.

local operation
The manipulation of an object which resides on the same node as the calling task.

logical address
An address used by an application. In a system without memory management, logical addresses will equal physical addresses.

loosely-coupled
A multiprocessor configuration where shared memory is not used for communication.

major number
The index of a device driver in the Device Driver Table.

manager
A group of related RTEMS' directives which provide access and control over resources.

MCS
An acronym for Mellor-Crummey Scott.

memory pool
Used interchangeably with heap.

message
A sixteen byte entity used to communicate between tasks. Messages are sent to message queues and stored in message buffers.

message buffer
A block of memory used to store messages.

message queue
An RTEMS object used to synchronize and communicate between tasks by transporting messages between sending and receiving tasks.

Message Queue Control Block
A data structure associated with each message queue used by RTEMS to manage that message queue.

minor number
A numeric value passed to a device driver, the exact usage of which is driver dependent.

mode
An entry in a task's control block that is used to determine if the task allows preemption, timeslicing, processing of signals, and the interrupt disable level used by the task.

MPCI
An acronym for Multiprocessor Communications Interface Layer.

multiprocessing
The simultaneous execution of two or more processes by a multiple processor computer system.

multiprocessor
A computer with multiple CPUs available for executing applications.

Multiprocessor Communications Interface Layer
A set of user-provided routines which enable the nodes in a multiprocessor system to communicate with one another.

Multiprocessor Configuration Table
The data structure defining the characteristics of the multiprocessor target system with which RTEMS will communicate.

multitasking
The alternation of execution amongst a group of processes on a single CPU. A scheduling algorithm is used to determine which process executes at which time.

mutual exclusion
A term used to describe the act of preventing other tasks from accessing a resource simultaneously.

nested
A term used to describe an ASR that occurs during another ASR or an ISR that occurs during another ISR.

node
A term used to reference a processor running RTEMS in a multiprocessor system.

non-existent
The state occupied by an uncreated or deleted task.

NUMA
An acronym for Non-Uniform Memory Access.

numeric coprocessor
A component used in computer systems to enhance performance in mathematically intensive situations. It is typically viewed as a logical extension of the primary processor.

object
In this document, this term is used to refer collectively to tasks, timers, message queues, partitions, regions, semaphores, ports, and rate monotonic periods. All RTEMS objects have IDs and user-assigned names.

object-oriented
A term used to describe systems with common mechanisms for utilizing a variety of entities. Object-oriented systems shield the application from implementation details.

operating system
The software which controls all the computer's resources and provides the base upon which application programs can be written.

overhead
The portion of the CPUs processing power consumed by the operating system.

packet
A buffer which contains the messages passed between nodes in a multiprocessor system. A packet is the contents of an envelope.

partition
An RTEMS object which is used to allocate and deallocate fixed size blocks of memory from an dynamically specified area of memory.

partition
Clusters with a cardinality of one are partitions.

Partition Control Block
A data structure associated with each partition used by RTEMS to manage that partition.

pending
A term used to describe a task blocked waiting for an event, message, semaphore, or signal.

periodic task
A task which must execute at regular intervals and comply with a hard deadline.

physical address
The actual hardware address of a resource.

poll
A mechanism used to determine if an event has occurred by periodically checking for a particular status. Typical events include arrival of data, completion of an action, and errors.

pool
A collection from which resources are allocated.

portability
A term used to describe the ease with which software can be rehosted on another computer.

posting
The act of sending an event, message, semaphore, or signal to a task.

preempt
The act of forcing a task to relinquish the processor and dispatching to another task.

priority
A mechanism used to represent the relative importance of an element in a set of items. RTEMS uses priority to determine which task should execute.

priority boosting
A simple approach to extend the priority inheritance protocol for clustered scheduling is priority boosting. In case a mutex is owned by a task of another cluster, then the priority of the owner task is raised to an artificially high priority, the pseudo-interrupt priority.

priority inheritance
An algorithm that calls for the lower priority task holding a resource to have its priority increased to that of the highest priority task blocked waiting for that resource. This avoids the problem of priority inversion.

priority inversion
A form of indefinite postponement which occurs when a high priority tasks requests access to shared resource currently allocated to low priority task. The high priority task must block until the low priority task releases the resource.

processor utilization
The percentage of processor time used by a task or a set of tasks.

proxy
An RTEMS control structure used to represent, on a remote node, a task which must block as part of a remote operation.

Proxy Control Block
A data structure associated with each proxy used by RTEMS to manage that proxy.

PTCB
An acronym for Partition Control Block.

PXCB
An acronym for Proxy Control Block.

QCB
An acronym for Message Queue Control Block.

quantum
The application defined unit of time in which the processor is allocated.

queue
Alternate term for message queue.

ready task
A task occupies this state when it is available to be given control of a processor. A ready task has no processor assigned. The scheduler decided that other tasks are currently more important. A task that is ready to execute and has a processor assigned is called scheduled.

real-time
A term used to describe systems which are characterized by requiring deterministic response times to external stimuli. The external stimuli require that the response occur at a precise time or the response is incorrect.

reentrant
A term used to describe routines which do not modify themselves or global variables.

region
An RTEMS object which is used to allocate and deallocate variable size blocks of memory from a dynamically specified area of memory.

Region Control Block
A data structure associated with each region used by RTEMS to manage that region.

registers
Registers are locations physically located within a component, typically used for device control or general purpose storage.

remote
Any object that does not reside on the local node.

remote operation
The manipulation of an object which does not reside on the same node as the calling task.

resource
A hardware or software entity to which access must be controlled.

resume
Removing a task from the suspend state. If the task's state is ready following a call to the `rtems_task_resume` directive, then the task is available for scheduling.

return code
Also known as error code or return value.

return code
A value returned by RTEMS directives to indicate the completion status of the directive.

RNCB
An acronym for Region Control Block.

round-robin
A task scheduling discipline in which tasks of equal priority are executed in the order in which they are made ready.

RS-232
A standard for serial communications.

running
The state of a rate monotonic timer while it is being used to delineate a period. The timer exits this state by either expiring or being canceled.

schedulable
A set of tasks which can be guaranteed to meet their deadlines based upon a specific scheduling algorithm.

schedule
The process of choosing which task should next enter the executing state.

scheduled task
A task is scheduled if it is allowed to execute and has a processor assigned. Such a task executes currently on a processor or is about to start execution. A task about to start execution it is an heir task on exactly one processor in the system.

scheduler
A scheduler or scheduling algorithm allocates processors to a subset of its set of ready tasks. So it manages access to the processor resource. Various algorithms exist to choose the tasks allowed to use a processor out of the set of ready tasks. One method is to assign each task a priority number and assign the tasks with the lowest priority number to one processor of the set of processors owned by a scheduler instance.

scheduler instance
A scheduler instance is a scheduling algorithm with a corresponding context to store its internal state. Each processor in the system is owned by at most one scheduler instance. The processor to scheduler instance assignment is determined at application configuration time. See Configuring a System.

segments
Variable sized memory blocks allocated from a region.

semaphore
An RTEMS object which is used to synchronize tasks and provide mutually exclusive access to resources.

Semaphore Control Block
A data structure associated with each semaphore used by RTEMS to manage that semaphore.

shared memory
Memory which is accessible by multiple nodes in a multiprocessor system.

signal
An RTEMS provided mechanism to communicate asynchronously with a task. Upon reception of a signal, the ASR of the receiving task will be invoked.

signal set
A thirty-two bit entity which is used to represent a task's collection of pending signals and the signals sent to a task.

SMCB
An acronym for Semaphore Control Block.

SMP
An acronym for Symmetric Multiprocessing.

SMP barriers
The SMP barriers ensure that a defined set of independent threads of execution on a set of processors reaches a common synchronization point in time. They are implemented using atomic operations. Currently a sense barrier is used in RTEMS.

SMP locks
The SMP locks ensure mutual exclusion on the lowest level and are a replacement for the sections of disabled interrupts. Interrupts are usually disabled while holding an SMP lock. They are implemented using atomic operations. Currently a ticket lock is used in RTEMS.

soft real-time system
A real-time system in which a missed deadline does not compromise the integrity of the system.

sporadic task
A task which executes at irregular intervals and must comply with a hard deadline. A minimum period of time between successive iterations of the task can be guaranteed.

stack
A data structure that is managed using a Last In First Out (LIFO) discipline. Each task has a stack associated with it which is used to store return information and local variables.

status code
Also known as error code or return value.

suspend
A term used to describe a task that is not competing for the CPU because it has had a `rtems_task_suspend` directive.

synchronous
Related in order or timing to other occurrences in the system.

system call
In this document, this is used as an alternate term for directive.

target
The system on which the application will ultimately execute.

TAS
An acronym for Test-And-Set.

task

thread
A logically complete thread of execution. It consists normally of a set of registers and a stack. The scheduler assigns processors to a subset of the ready tasks. The terms task and thread are synonym in RTEMS. The term task is used throughout the Classic API, however, internally in the operating system implementation and the POSIX API the term thread is used.

Task Control Block
A data structure associated with each task used by RTEMS to manage that task.

task migration
Task migration happens in case a task stops execution on one processor and resumes execution on another processor.

task processor affinity
The set of processors on which a task is allowed to execute.

task switch
Alternate terminology for context switch. Taking control of the processor from one task and given to another.

TCB
An acronym for Task Control Block.

thread dispatch
The thread dispatch transfers control of the processor from the currently executing thread to the heir thread of the processor.

tick
The basic unit of time used by RTEMS. It is a user-configurable number of microseconds. The

current tick expires when a clock tick directive is invoked.

tightly-coupled
A multiprocessor configuration system which communicates via shared memory.

timeout
An argument provided to a number of directives which determines the maximum length of time an application task is willing to wait to acquire the resource if it is not immediately available.

timer
An RTEMS object used to invoke subprograms at a later time.

Timer Control Block
A data structure associated with each timer used by RTEMS to manage that timer.

timeslice
The application defined unit of time in which the processor is allocated.

timeslicing
A task scheduling discipline in which tasks of equal priority are executed for a specific period of time before being preempted by another task.

TLS
An acronym for Thread-Local Storage *[Dre13]*. TLS is available in *C11* and *C++11*. The support for TLS depends on the CPU port *[RTE]*.

TMCB
An acronym for Timer Control Block.

transient overload
A temporary rise in system activity which may cause deadlines to be missed. Rate Monotonic Scheduling can be used to determine if all deadlines will be met under transient overload.

TTAS
An acronym for Test and Test-And-Set.

User Extension Table
A table which contains the entry points for each user extensions.

user extensions
Software routines provided by the application to enhance the functionality of RTEMS.

User Initialization Tasks Table
A table which contains the information needed to create and start each of the user initialization tasks.

user-provided

user-supplied
These terms are used to designate any software routines which must be written by the application designer.

vector
Memory pointers used by the processor to fetch the address of routines which will handle various exceptions and interrupts.

wait queue
The list of tasks blocked pending the release of a particular resource. Message queues, regions, and semaphores have a wait queue associated with them.

yield
When a task voluntarily releases control of the processor.

CHAPTER

FOUR

INDEX

BIBLIOGRAPHY

[RTE] RTEMS CPU Architecture Supplement. URL: https://docs.rtems.org/branches/master/cpu-supplement.pdf.

[BBB+13] Dave Banham, Andrew Banks, Mark Bradbury, Paul Burden, Mark Dawson-Butterworth, Mike Hennell, Chris Hills, Steve Montgomery, Chris Tapp, and Liz Whiting. *MISRA C:2012 Guidelines for the Use of the C Language in Critical Systems*. MISRA Limited, March 2013. ISBN 978-1906400101.

[Boe12] Hans-J. Boehm. Can Seqlocks Get Along With Programming Language Memory Models? Technical Report, HP Laboratories, June 2012. HPL-2012-68. URL: http://www.hpl.hp.com/techreports/2012/HPL-2012-68.pdf.

[Bra11] Björn B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011. URL: http://www.cs.unc.edu/~bbb/diss/brandenburg-diss.pdf.

[Bra13] Björn B. Brandenburg. A Fully Preemptive Multiprocessor Semaphore Protocol for Latency-Sensitive Real-Time Applications. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS 2013)*, 292–302. 2013. URL: http://www.mpi-sws.org/~bbb/papers/pdf/ecrts13b.pdf.

[Bur91] A. Burns. Scheduling hard real-time systems: a review. *Software Engineering Journal*, 6:116–128, 1991.

[BW01] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Addison-Wesley, November 2001. ISBN 978-0321417459.

[BW13] A. Burns and A. J. Wellings. A Schedulability Compatible Multiprocessor Resource Sharing Protocol - MrsP. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS 2013)*. 2013. URL: http://www-users.cs.york.ac.uk/~burns/MRSPpaper.pdf.

[CBHM15] Sebastiano Catellani, Luca Bonato, Sebastian Huber, and Enrico Mezzetti. Challenges in the Implementation of MrsP. In *Reliable Software Technologies - Ada-Europe 2015*, 179–195. 2015.

[CvdBC16] Kuan-Hsun Chen, Georg von der Brüggen, and Jian-Jia Chen. Overrun Handling for Mixed-Criticality Support in RTEMS. In *Mixed Criticality Systems - WMC 2016*, 13–14. 2016. URL: http://ls12-www.cs.tu-dortmund.de/daes/media/documents/publications/downloads/2016-wmc.pdf.

[CMV14] Davide Compagnin, Enrico Mezzetti, and Tullio Vardanega. Putting RUN into practice: implementation and evaluation. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS 2014)*. 2014.

[Dre07] Ulrich Drepper. *What Every Programmer Should Know About Memory*. 2007. URL: http://www.akkadia.org/drepper/cpumemory.pdf.

[Dre13] Ulrich Drepper. *ELF Handling For Thread-Local Storage*. 2013. URL: http://www.akkadia.org/drepper/tls.pdf.

[FRK02] Hubertus Franke, Rusty Russel, and Matthew Kirkwood. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In *Proceedings of the Ottawa Linux Symposium 2002*, 479–495. 2002. URL: https://www.kernel.org/doc/ols/2002/ols2002-pages-479-495.pdf.

[GN06] Thomas Gleixner and Douglas Niehaus. Hrtimers and Beyond: Transforming the Linux Time Subsystems. In *Proceedings of the Linux Symposium*, 333–346. 2006. URL: https://www.kernel.org/doc/ols/2006/ols2006v1-pages-333-346.pdf.

[GCB13] Arpan Gujarati, Felipe Cerqueira, and Björn B. Brandenburg. Schedulability Analysis of the Linux Push and Pull Scheduler with Arbitrary Processor Affinities. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS 2013)*. 2013. URL: https://people.mpi-sws.org/~bbb/papers/pdf/ecrts13a-rev1.pdf.

[LSD89] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Real-Time Systems Symposium*, 166–171. 1989.

[LL73] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20:46–61, 1973.

[LLF+16] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The Linux Scheduler: a Decade of Wasted Cores. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. 2016. URL: https://hal.archives-ouvertes.fr/hal-01295194/document.

[SG90] Lui Sha and J. B. Goodenough. Real-time scheduling theory and Ada. *Computer*, 23:53–62, 1990.

[SRL90] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39:1175–1185, 1990.

[VC95] G. Varghese and A. Costello. Redesigning the BSD callout and timer facilities. Technical Report, Washington University in St. Louis, November 1995. WUCS-95-23. URL: http://web.mit.edu/afs.new/sipb/user/daveg/ATHENA/Info/wucs-95-23.ps.

[VL87] G. Varghese and T. Lauck. Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*. 1987. URL: http://www.cs.columbia.edu/~nahum/w6998/papers/sosp87-timing-wheels.pdf.

[Wil12] Anthony Williams. *C++ Concurrency in Action – Practical Multithreading*. Manning Publications Co, 2012. ISBN 978-1933988771.

INDEX