rtmk User and Developer Documentation

A free real-time microkernel
For version 0.2, 4 February 2002

Johan Rydberg, jrydberg@rtmk.org

Table of Contents

1 Introduction 1

1.1 Basic kernel functionality 1

2 Kernel Interface 2

2.1 Tasks - The execution environment 2
2.1.1 The running task 2
2.1.2 Creating tasks 2
2.1.3 Task information 3
2.1.4 Task's special ports 3
2.2 Threads - the basic execution unit. 4
2.2.1 The executing thread 4
2.2.2 Controling threads 4
2.2.3 Reply ports 4
2.2.4 Special ports 5
2.2.5 Thread states 5
2.2.6 Setting priority 6
2.3 Ports - The communication channel 6
2.3.1 Allocating ports 6
2.3.2 Destroying ports 7
2.3.3 Migration control 7
2.3.4 Sending and receiving messages 7
2.4 Virtual memory management 8
2.4.1 Locking memory 8
2.4.2 Mapping a memory object 8
2.4.3 Copying memory between tasks 8
2.5 Exception handling 9
2.6 Kernel error codes 9

3 Message-based communicaton 11

3.1 Basic concepts 11

4 Intel 80386 Dependent Features 12

4.1 Application Binary Interface Related 12
4.2 Machine-dependent thread states 12
4.3 Booting the kernel 12

5 Function Index 13

1 Introduction

'`rtmk`' is a communication-oriented operating system kernel providing:

- multiple tasks, each with a large, paged and protected virtual memory space,
- multiple threads of execution within each task, with a flexible scheduling facility,
- flexible sharing of memory between tasks, and
- message-based interprocess communication.

1.1 Basic kernel functionality

The '`rtmk`' microkernel supports the following basic abstractions:

- A `task` is an execution environment and is the basic unit of resource allocation. A task includes a paged virtual address space and protected access to system resourecs.
- A `thread` is the basic unit of execution. It consists of all processor state (e.g., hardware registers) necessary for independent execution. A thread executes in the virtual memory and port rights context of a single task. The conventional notion of a process is, in rtmk, represented by a task with a single thread of control.
- A `port` is a simple communication channel – implemented as a message queue managed and protected by the kernel.
- A `message` is a typed collection of data objects used in communications between threads. A message can be of any size and contain inline data, pointers to data or capabilities for ports.

Message-passing is the primary mean of communication among tasks. The `rtmk` kernel functions can be divided into the following groups:

- basic message primitives and support facilities,
- port management facilities,
- task and thread creation and management facilities, and
- virtual memory management facilities.

2 Kernel Interface

2.1 Tasks - The execution environment

A *task* is an execution environment and is the basic unit of resource allocation. A task includes a paged virtual address space and protected access to system resources.

The size of the virtual address space is architecture dependent. The Intel 80386 port of '`rtmk`' provides a 3 GB address space to the user. The kernel uses the first 1 GB of the address space, this memory is not visible to the user application.

2.1.1 The running task

A thread can always get the name of the send right of the task that it is currently executing in, by simply calling '`task_self`'.

`rtmk_port` **task_self** (`void`) Task function

Return send rights to the task that the current executing thread is running in. References to the task is not increased by this function, so there is no need to deallocate after usage.

2.1.2 Creating tasks

When creating a new task the user application can choose to fork of the parent tasks address space or create a new, empty, address space of the child task. A newly created task is `NOT` suspended.

`kern_return_t` **task_create** (`rtmk_port_t` *task*, `bool` *fork_p*, `rtmk_port_t` **child_taskp*) Task function

Creates a new task, where *task* will act as the parent task. If *fork_p* is `true`, the address space of *task* will be forked, taking region inheritance flags in account. Send right to the new task is returned in *child_taskp*. The kernel always hold receive right for a task.

`kern_return_t` **task_terminate** (`rtmk_port_t` *task*) Task function

Try to terminate *task*. When this function returns, all execution of threads in *task* have been stopped and the task have been terminated. If *task* is the current running task (i.e. we are terminating our self), this function will never return.

`kern_return_t` **task_suspend** (`rtmk_port_t` *task*) Task function

Suspend execution of all threads that belong to *task*, until they are resumed.

`kern_return_t` **task_resume** (`rtmk_port_t` *task*) Task function

Resume execution of all threads (except those who is individually suspended) that belong to *task*.

2.1.3 Task information

An application with a send right to a `task` can always retrieve information about that task. '`thread_info`' returns a structure containing basic information about the task and the number of resources it is holding.

`kern_return_t` **task_threads** (`rtmk_port_t` *task*, `rtmk_port_t` **threadsp*, `int` **countp*) Task function

This function returns an array, *threadsp*, with **countp* entries containing send rights to all threads in *task*. The array is '`out-of-line`' memory, so it has to be deallocated using `vm_deallocate` after it has been used.

`kern_return_t` **task_info** (`rtmk_port_t` *task*, `struct task_info` **infop*, `int` **countp*) Task function

Retrieve information about *task* and store it in **infop*. On call, **countp* must hold the value of '`TASK_INFO_COUNT`'.

`kern_return_t` **task_names** (`rtmk_port_t` *task*, `rtmk_port_name_t` **rightsp*, `int` **rights_countp*, `rtmk_port_type_t` **types*, `int` **types_countp*) Task function

Retrieve two arrays that hold information about names and types of all rights that *task* holds. Arrays must be deallocated after they have been used.

2.1.4 Task's special ports

Each task controls a set of special ports that are used for several purposes. Each slot in the port set contains a send right to a port that can be retrieved by a someone that holds send rights to the task. Available slots:

`TASK_SPECIAL_PORT_KERNEL`
: Represents task to the outside world. This is the port that is returned by '`task_self`'.

`TASK_SPECIAL_PORT_BOOTSTRAP`
: Slot can be used to identify '`bootstrap port`' that is assigned to the particular task. The kernel does not use the bootstrap port internally, but applications can use it when forking of children.

`TASK_SPECIAL_PORT_EXCEPTION`
: Exception messages for task are sent to this port. See Section 2.5 [Exceptions], page 9.

There are some slots reserved for the future, and some that are free to be used by applications.

`kern_return_t` **task_special_port_set** (`rtmk_port_t` *task*, `int` *slot*, `rtmk_port_t` *port*) Task function

Set control port in *task* to *port* at *slot* in control port array. (??? write something else here) `kern_return_t` **task_special_port_get** (`rtmk_port_t` *task*, `int` *slot*, `rtmk_port_t` **portp*) — Task function

Get send rights to port *slot* in *task*'s control port set. Right is returned in *portp*.

2.2 Threads - the basic execution unit.

A *thread* is the basic unit of execution. It consists of all processor state (e.g., hardware registers) necessary for independent execution, and scheduling information.

At any given time a thread executes in the virtual memory and port rights context of ONE single task. But threads can migrate to other tasks, using *full migrated RPC*.

The conventional notion of a *process* is, in '`rtmk`', represented by a task with a single thread of control.

2.2.1 The executing thread

A thread can always get the name of the send right of itself, the thread that it is currently executing, by simply calling '`thread_self`'.

`rtmk_port_t` **thread_self** (`void`) — Thread function

Return send rights to the current executing thread.

2.2.2 Controling threads

When a thread is created, it is assigned to a task. This is the task that the thread will begin to execute in, it's so called *home task*.

`kern_return_t` **thread_create** (`rtmk_port_t` *task*, `rtmk_port_t` **threadp*) — Thread function

Create thread that will execute in *task*. New thread is suspended.

`kern_return_t` **thread_terminate** (`rtmk_port_t` *thread*) — Thread function

Terminate *thread*.

`kern_return_t` **thread_suspend** (`rtmk_port_t` *thread*) — Thread function

Suspend execution of *thread*.

`kern_return_t` **thread_resume** (`rtmk_port_t` *thread*) — Thread function

Resume execution of *thread* if suspend count drops to zero.

2.2.3 Reply ports

To perform a *RPC* the thread needs a reply port to receive the reply on. To allocate this port using `port_allocate` would cause to much overhead. The `thread_reply_port` system call return right name to a newly allocated port, that can be used for receiving replies.

`rtmk_port_t` **thread_reply_port** (`void`) — Thread function

Allocate a port that can be used a receive port of replies.

2.2.4 Special ports

Each thread, just like tasks, controls a set of special ports. Each slot in the port set contains a send right to a port that can be retrieved by a someone that holds send rights to the task. Available slots:

`THREAD_SPECIAL_PORT_KERNEL`
: Represents thread to the outside world. This is the port that is returned by '`thread_self`'.

`THREAD_SPECIAL_PORT_EXCEPTION`
: Exception messages for task are sent to this port. See Section 2.5 [Exceptions], page 9.

There are some slots reserved for the future, and some that are free to be used by applications.

`kern_return_t` **thread_special_port_set** (`rtmk_port_t` *thread*, `int` *slot*, `rtmk_port_t` *port*) — Thread function

Set control port in *thread* to *port* at *slot* in control port array. (??? write something else here)

`kern_return_t` **thread_special_port_get** (`rtmk_port_t` *thread*, `int` *slot*, `rtmk_port_t` **portp*) — Thread function

Get send rights to port *slot* in *thread*'s control port set. Right is returned in *portp*.

2.2.5 Thread states

The `thread_state_get` and `thread_state_set` function are used to retrieve or set information about a particular thread. The *flavor* argument specifies what state/status we want. Available flavors:

`THREAD_STATE_FLAVOR_TIMING`
: Timing information about thread. The '`thread_state_timing`' structure holds both user- and system-timing. **countp* should be `THREAD_STATE_FLAVOR_TIMING_COUNT`. This flavor is **read only**.

`kern_return_t` **thread_state_get** (`rtmk_port_t` *thread*, `int` *flavor*, `void` **state*, `int` **countp*) — Thread function

Get state specified with *flavor* from *thread*. State is returned in *state*. **countp* should be the size of the state. See above.

`kern_return_t` **thread_state_set** (`rtmk_port_t` *thread*, `int` *flavor*, `void` **state*, `int` *count*) — Thread function

Set state specified with *flavor* from *thread*. *state* holds the state. *count* should be the size of the state. See above.

2.2.6 Setting priority

The *rtmk* microkernel provides three different scheduling policies and a 0-127 priority range per policy. These are set per-thread.

`THREAD_POLICY_TIMESHARE`
: The default scheduling policy. The threads are scheduled using a credit-based time sharing algorithm.

`THREAD_POLICY_RR`
: Threads are scheduled in a round-robin maner.

`THREAD_POLICY_FIFO`
`THREAD_POLICY_FCFS`
: A *first come, first served* scheduling algorithm. Threads are only preempted by higher-priority threads.

`kern_return_t` **thread_priority_set** (`rtmk_port_t` *thread*, `int` *policy*, `int` *priority*) — Thread function

Set scheduling policy and priority for *thread* to *policy* and *priority*. If *policy* is an unknown scheduling policy, or if *priority* is out of range, `KERN_INVALID_ARGUMENT` is returned.

2.3 Ports - The communication channel

A *port* is a simple communication channel – implemented as a message queue managed and protected by the kernel.

A *port set* is a collection of ports that have a single protected message queue, which enables *M:N* communication with a single server.

2.3.1 Allocating ports

Ports and port sets are allocated with the same functions.

`kern_return_t` **port_allocate** (`rtmk_port_t` *task*, `rtmk_port_right_t` *flavor*, `rtmk_port_t` **portp*) — Ports function

Allocate receive right to a new port in *task*'s protected name space. *flavor* specifies what type of port right we should allocate, either `RTMK_PORT_RIGHT_RECEIVE` or `RTMK_PORT_RIGHT_PORT_SET`.

`kern_return_t` **port_allocate_named** (`rtmk_port_t` *task*, `rtmk_port_right_t` *flavor*, `rtmk_port_t` *port_name*) — Ports function

Same things as '`port_allocate`' except that we don't let the kernel choose our right name. Instead we insist on the name *port_name*.

`kern_return_t` **port_move_member** (`rtmk_port_t` *task*, `rtmk_port_t` *member*, `rtmk_port_t` *pset*) — Ports function

Insert *member* into port set specified by *pset*. If *pset* is *NULL*, *member* is removed from any port set it was a member of.

2.3.2 Destroying ports

`kern_return_t` **port_deallocate** (`rtmk_port_t` *task*, `rtmk_port_t` *port_name*) — Ports function

Deallocate a reference to *port_name*. If reference count drops to zero, the right is removed from *task*'s protected name space.

`kern_return_t` **port_destroy** (`rtmk_port_t` *task*, `rtmk_port_t` *port_name*) — Ports function

Destroy *port_name*. *task* must hold receive right to it, which can be either a port or a port set. After this, the port is considered dead and no more messages can be sent to it.

2.3.3 Migration control

It is possible to forbid and permit threads from migrating into the targets context. Threads that tries to migrate through a migrate inhibited target will block until migration is re-enabled.

`kern_return_t` **port_inhibit** (`rtmk_port_t` *task*, `rtmk_port_t` *port_name*) — Ports function

Inhibit migration to port or port set specified by *port_name*.

`kern_return_t` **port_exhibit** (`rtmk_port_t` *task*, `rtmk_port_t` *port_name*) — Ports function

Enable threads to migrate into *task*'s context through *port_name*.

2.3.4 Sending and receiving messages

`kern_return_t` **msg_send** (`struct rtmk_msg_header` **msgh*, `rtmk_msg_timeout_t` *timeout*) — Ports function

Send message to `msgh->msgh_remote_port`. *msgh* is pointer to typed data. If *timeout* is zero, we can block forever.

`kern_return_t` **msg_receive** (`struct rtmk_msg_header` **msgh*, `rtmk_msg_timeout_t` *timeout*) — Ports function

Receive message from local port specified in message header *msgh*. If *timeout* is zero, we can block forever.

`kern_return_t` **msg_rpc** (`struct rtmk_msg_header` **msgh*, `rtmk_msg_size_t` *recv_size*, `rtmk_msg_timeout_t` *timeout*) — Ports function

Perform a full RPC from information in *msgh*. *recv_size* is length of receive buffer. If *timeout* is zero, we can block forever.

`kern_return_t` **msg_migrate** (`struct rtmk_msg_header` **msgh*, `rtmk_msg_size_t` *recv_size*) — Ports function

Perform a full RPC with thread migration (the fast path). *recv_size* is length of receive buffer.

2.4 Virtual memory management

`kern_return_t` **vm_allocate** (`rtmk_port_t` *task*, `vm_size_t` *size*, `vm_offset_t` **offsetp*, `int` *anywhere_p*) — VM function

Allocate anonymous region of *size* bytes in *task*'s address space. If *anywhere_p* is true the kernel chooses offset into address space, othersize **offsetp* specifies location. Offset is returned in *offsetp*.

`kern_return_t` **vm_deallocate** (`rtmk_port_t` *task*, `vm_offset_t` *offset*, `vm_size_t` *size*) — VM function

Deallocate region [*offset*, *offset+size*) of *task*'s address space.

`kern_return_t` **vm_protect** (`rtmk_port_t` *task*, `vm_offset_t` *offset*, `vm_size_t` *size*, `vm_prot_t` *protection*) — VM function

Lower protection level of region [*offset*, *offset+size*) to *protection*. If *protection* is higher than maximum protection, `KERN_INVALID_ARGUMENT` is returned.

2.4.1 Locking memory

For some applications it is neccesarry, to ensure real-time, to lock certain regions of the address space in memory. Locked memory will never be swaped out.

`kern_return_t` **vm_wire** (`rtmk_port_t` *task*, `vm_offset_t` *offset*, `vm_size_t` *size*, `int` *wired_p*) — VM function

Lock region [*offset*, *offset+size*) into memory if *wired_p* is true. If user tries to lock a region into memory, and some pages were swaped-out, those are brought in before this function returns.

2.4.2 Mapping a memory object

`kern_return_t` **vm_map** (`rtmk_port_t` *task*, `rtmk_port_t` *memory_object*, `vm_offset_t` **offsetp*, `vm_size_t` *size*, `int` *anywhere_p*, `vm_prot_t` *prot*, `vm_inherit_t` *inherit*) — VM function

Map *size* bytes of *memory_object* into *task*'s address space. Kernel chooses offset into address space if *anywhere_p* is true, othersize **offsetp* specifies location. Offset is returned in *offsetp*.

2.4.3 Copying memory between tasks

Sometime it is neccessary to copy memory between different address spaces. This can be done by three functions; '`vm_write`', '`vm_read`' and '`vm_copy`'.

??? wip!

2.5 Exception handling

When a thread causes and exception, due to for example a divide by zero, an exception message is send to the threads exception port. If the thread do not have an assigned port, it send it to the tasks port, of which the thread belongs to.

The message is in the form of a RPC, defined as following:

```
(define-method exception_raise (returns REMS_MSG_TYPE_INTEGER32)
  (arguments (out exc_port  RTMK_MSG_TYPE_COPY_SEND)
             (out thread    RTMK_MSG_TYPE_COPY_SEND)
             (out task      RTMK_MSG_TYPE_COPY_SEND)
             (out exception RTMK_MSG_TYPE_INTEGER32)
             (out code      RTMK_MSG_TYPE_INTEGER32)
             (out subcode   RTMK_MSG_TYPE_INTEGER32)
             (out state     RTMK_MSG_TYPE_INTEGER8[]))
)
```

exc_port is the exception port that the message is sent to. *thread* is the thread that caused the exception, and the thread belongs to *task*. *exception* tells us what type of exception *threads* raised. The value of *code* and *subcode* is dependent on type of exception. Exception types:

`EXCEPTION_BAD_ACCESS`
: Could not access memory. *code* contains ‘`kern_return_t`’ describing error. *code* contains bad memory address.

`EXCEPTION_BAD_INSN`
: Instruction failed. *code* contains address of bad instruction.

`EXCEPTION_ARITHMETIC`
: Arithmetic error. Exact nature of exception is in *code*.

`EXCEPTION_SOFTWARE`
: Exception caused by software. The value of *code* and *subcode* is dependent on architecture.

`EXCEPTION_BREAKPOINT`
: Thread caused an breakpoint. The value of *code* and *subcode* is dependent on architecture. (??? is this correct?)

2.6 Kernel error codes

All kernel functions that returns a value of the `kern_return_t` type uses a set of standard error codes, that is listed here:

`KERN_SUCCESS`
: No error.

`KERN_INVALID_ADDRESS`
: Address specifed was not valid.

`KERN_NO_SPACE`
: No space in the virtual address space.

`KERN_INVALID_ARGUMENT`
: User passed an invalid argument to the kernel.

`KERN_FAILURE`
: General failure. Kernel can not specify what went wrong.

`KERN_RESOURCE_SHORTAGE`
: The kernel ran out of resources while trying to perform the action. Normally this means that there no more memory, and the page-out daemon does not work as it should.

`KERN_NOT_RECEIVER`
: The task does not have receive rights for a specified port.

`KERN_NO_ACCESS`
: The task have no access to specified resource.

`KERN_NOT_IN_SET`
: Port is not a member of the specified port set.

`KERN_NAME_EXISTS`
: The specified right name already exist.

`KERN_RIGHT_EXISTS`
: The specified right already existed.

`KERN_ABORTED`
: The system call was aborted.

`KERN_INVALID_NAME`
: The name that was specifed was invalid.

`KERN_INVALID_TASK`
: The task that was specifed was invalid.

`KERN_INVALID_HOST`
: The host that was specifed was invalid.

`KERN_INVALID_RIGHT`
: The right that was specified was invalid.

`KERN_INVALID_VALUE`
: The value was invalid.

3 Message-based communicaton

In *rtmk*, IPC is the central and most import kernel component. Instead of the operating system supporting IPC mechanisms, *rtmk* provides an IPC facility that supports mich of the operating system. There are several important goals in the design of rtmk IPC;

- Message passing must be the fundamental communcation mechanism.
- The amount of data in a single message may range from a few bytes to an entire address space. The kernel should enable large transfers without unneccesarry data copying.
- The kernel should provide secure communications and allow only authorized threads to send and receive messages.
- Communication and memory management are tightly coupled. The IPC subsystem uses the *copy-on-write* mechanism of the memory subsystem to effeciently transfer large amounts of data.
- The IPC mechanism should be suitable for applications based on the *client-server* model.
- The subsystem should be highly optimized and should create as little overhead as possible.

3.1 Basic concepts

The *rtmk* microkernel supplys two fundamental IPC abstractions; messages and ports. A *message* is a collection of typed data. A *port* is a protected queue of messages. A message can be send only to a port, not to a task or a thread. rtmk associates *send rights* and *receive rights* with each port. These rights are owned by tasks. A send right allow a task to send messages to the port; a receive right allows it to receive mesages sent to the port. Several tasks may own send rights to a single port, but only one task holds the receive rights. Thus a port allows many-to-one communication.

Each port has a reference count that monitors the number of rights to it. Each such right (a.k.a. *capability*) represent one name of that port. The names are integer, and the name space is local to each task. Thus two tasks may have different names for the same port. Conversely, the same port name may refer to different ports in different tasks.

Ports also represent kernel object. Hence each object, such as a task, thread, or host, is represented by a port. Rights to these ports represent object references and allow the holder to perform oprtations on that object. The kernel holds the receive rights to such ports.

??? WIP

4 Intel 80386 Dependent Features

The i386 version of 'rtmk' supports the 32-bit Intel architecture. Sometime in the future support for the 64-bit architecture will be added.

4.1 Application Binary Interface Related

The SVR4/i386 ABI (pages 3-31, 3-32) says that when the entry point runs, most registers' values are unspecified, except for:

`%edx` Contains a function pointer to be registered with 'atexit'. This is how the dynamic linker arranges to have DT_FINI functions called for shared libraries that have been loaded before this code runs.

`%esp` The stack contains the arguments and environment:

```
0(%esp)                 argc
4(%esp)                 argv[0]
...
(4*argc)(%esp)          NULL
(4*(argc+1))(%esp)      envp[0]
...
                        NULL
```

4.2 Machine-dependent thread states

Intel 80386 depdendent thread state flavors:

`THREAD_STATE_FLAVOR_I386_CPU`
The executing context (i.e., hardware registers) of the thread. The 'thread_state_i386_cpu' structure holds execution state. **countp* should be THREAD_STATE_FLAVOR_I386_CPU_COUNT. To set or read this state, the thread must be suspended.

`THREAD_STATE_FLAVOR_I386_LDT`
On the Intel 80386 architecture each thread have a LDT entry available for custom use. The 'thread_state_i386_ldt' structure holds LDT state. The segment number is 0x17.

4.3 Booting the kernel

The *rtmk* kernel uses the GNU GRUB bootloader to load the microkernel and the operating system kernel. Example of GRUB configuration file:

```
title rtmk + operating system kernel
  root (hd0,1)
  kernel /boot/rtmk
  module /boot/os-kernel --single-user --root=hd0a
```

You can find GNU GRUB at http://www.gnu.org/software/grub/grub.html.

5 Function Index

M

msg_migrate 7
msg_receive 7
msg_rpc 7
msg_send 7

P

port_allocate 6
port_allocate_named 6
port_deallocate 7
port_destroy 7
port_exhibit 7
port_inhibit 7
port_move_member 6

T

task_create 2
task_info 3
task_names 3
task_resume 2
task_self 2
task_special_port_get 4
task_special_port_set 3
task_suspend 2
task_terminate 2
task_threads 3
thread_create 4
thread_priority_set 6
thread_reply_port 4
thread_resume 4
thread_self 4
thread_special_port_get 5
thread_special_port_set 5
thread_state_get 5
thread_state_set 5
thread_suspend 4
thread_terminate 4

V

vm_allocate 8
vm_deallocate 8
vm_map 8
vm_protect 8
vm_wire 8