Getting Started with RTLinux

FSM Labs, Inc.

April 20, 2001

Contents

Chapter 1

Introduction

Welcome to the RTLinux Getting Started Guide! RTLinux is a hard realtime operating system that coexists with the Linux OS. With RTLinux, it is possible to create realtime POSIX.1b threads that will run at precisely specified moments of time. We have designed the Getting Started Guide with the assumption that the reader has had some programming experience, but has never used RTLinux.

The document is organized as follows. First, we present basic information needed to get started: sources of help, common programming errors, and an overview of the RTLinux design (Chapter 1). Next, we present the basic RTLinux API and will step you through your first "Hello World" program (Chapter 2). Third, we offer some of the more advanced API (Chapter 3), after which you'll find some special considerations and concepts (Chapter 4). Finally, in the appendices, you find some different ways of running RTLinux programs (Appendix A) and, most importantly, a complete listing of the RTLinux API, utilities, and important paths (Appendix B).

1.1 Sources of Help

The RTLinux white paper in `doc/design.pdf` explains the basic architecture in more detail and a summary of the design is presented in Section 1.3. As you progress in your use of RTLinux, you'll find yourself wanting more information. Fortunately, there are many sources of help. For the most up-to-date information, see the http://www.fsmlabs.com , http://www.-rtlinux.com and http://www.rtlinux.org websites.

If you are primarily interested in hard realtime control and not particularly interested in learning how to use RTLinux itself, take a look at FSM Labs RTiC-Lab at www.rtic-lab.org . RTiC-Lab is a front end to RTLinux that greatly simplifies hard realtime control implementation, monitoring and tuning.

If you are interested in running RTLinux on an industry standard PC-104 board or other type of minimal or embedded system, see FSMLabs MiniRTL project , found at www.rtlinux.-org/minirtl.html MiniRTL fits on a signle floppy disk and provides full RTLinux capabilities.

Some other documents you may find useful are (Note: All references to directories and files assume that RTLinux has been installed in its default location `/usr/rtlinux`).:

- The RTLinux Manual Project, available at -www.rtlinux.org/documents/documentation/RTLManual/RTLManual.html
- The Single UNIX specification, available at www.opengroup.org/onlinepubs/-7908799/index.html . (The Single UNIX spec is also installed in HTML format with the RTLinux distribution. (`susv2/index.html`)
- The LinuxThreads library documentation at http://pauillac.inria.fr/-˜xleroy/linuxthreads (included with glibc2). You can try running:

  ```
  man 3 pthread_create
  ```

  to see if it is installed on your system.
- "Getting Started With POSIX Threads" (by Thomas Wagner and Don Towsley), available at centaurus.cs.umass.edu/˜wagner/threads-_html/tutorial.html .
- "Pthreads Programming", available at www.oreilly.com/catalog/pthread by Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell.

- Other documents or books describing POSIX threads.

The RTLinux distribution itself contains documentation to help you along in your RTLinux projects:

- The `man` directory contains UNIX manual pages describing features and commands specific to RTLinux. You can modify the MANPATH environment variable so that these manual pages can be found with the man command. (Type `man man` for instructions on how to change the MANPATH variable globally.)

- The `html/MAN` directory contains the same manual pages, converted to HTML.

- The RTLinux Frequently Asked Questions (FAQ) file can be found under the top level directory of the RTLinux source tree.

- The `examples` directory contains programs which will give you first-hand experience with the RTLinux API.

If, after attempting all of the above, you still have questions, there is another rich source of information via the RTLinux mailing lists. You can subscribe/unsubscribe to these lists at www.rtlinux.org/mailing_lists.html . Of course, you may not be the first person with your question. To ease your search for answers, the lists are both browseable and searchable.

1.2 Before You Begin: A Warning

Realtime programs in RTLinux are executed in kernel space and have little or no protection against bugs in the user's code. Special care must be taken when programming realtime tasks because programming errors may bring the system down.

RTLinux supplies a debugger within its source tree under the directory `debugger`. *Use of the debugger is strongly recommended to reduce the risk of system crashes.*

Note also that by default RTLinux tasks do not have access to the computer's Floating Point Unit (FPU). You must explicitly set permissions for each of your RTLinux tasks that require the use of the FPU.

1.3 RTLinux Overview

This section is intended to give users a top-level understanding of RTLinux. It is not designed as an in-depth technical discussion of the system's architecture. Readers interested in the topic can start with Michael Barabanov's Master's Thesis. (A postscript version is available for download at www.rtlinux.org/documents/papers/thesis.ps).

The basic premise underlying the design of RTLinux is that it is not feasible to identify and eliminate all aspects of kernel operation that lead to unpredictability. These sources of unpredictability include the Linux scheduling algorithm (which is optimized to maximize throughput), device drivers, uninterrruptible system calls, the use of interrupt disabling and virtual memory operations. The best way to avoid these problems is to construct a small, predictable kernel separate from the Linux kernel, and to make it simple enough that operations can be measured and shown to have predictable execution. This has been the course taken by the developers of RTLinux. This approach has the added benefit of maintainability - prior to the development of RTLinux, every time new device drivers or other enhancements to Linux were needed, a study would have to be performed to determine that the change would not introduce unpredictability.

Figure 1.1 shows the basic Linux kernel without hard realtime support. You will see that the Linux kernel separates the hardware from user-level tasks. The kernel has the ability to suspend any user-level task, once that task has outrun the "slice of time" allotted to it by the CPU. Assume, for example, that a user task controls a robotic arm. The standard Linux kernel could potentially preempt the task and give the CPU to one which is less critical (e.g. one that boots up Netscape). Consequently, the arm will not meet strict timing requirements. Thus, in trying to be "fair" to all tasks, the kernel can prevent critical events from occurring.

Figure 1.2 shows a Linux kernel modified to support hard realtime. An additional layer of abstraction - termed a "virtual machine" in the literature - has been added between the standard Linux kernel and the computer hardware. As far as the standard Linux kernel is concedrned, this new layer appears to be actual hardware. More importantly, this new layer introduces its own fixed-priority scheduler. This scheduler assigns the lowest priority to the standard Linux kernel, which then runs as an independent task. Then it allows the user to both introduce and set priorities for any number of realtime tasks.

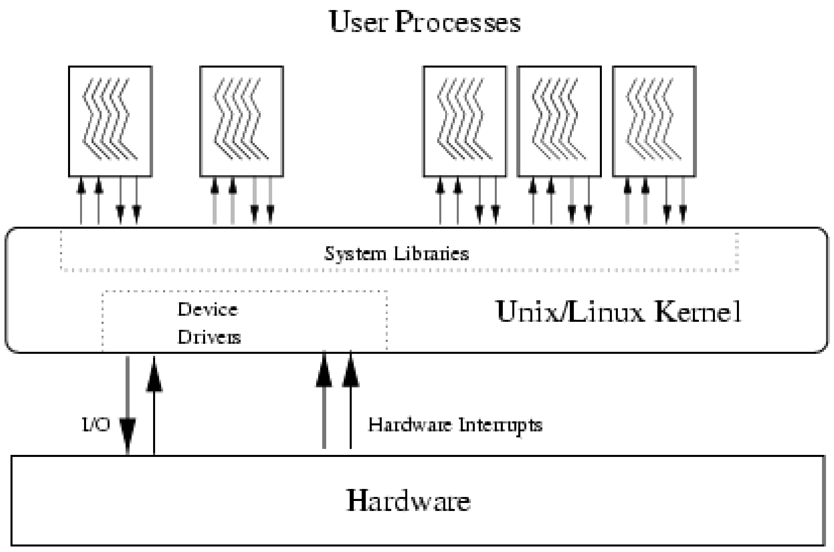

Figure 1.1: Detail of the bare Linux kernel

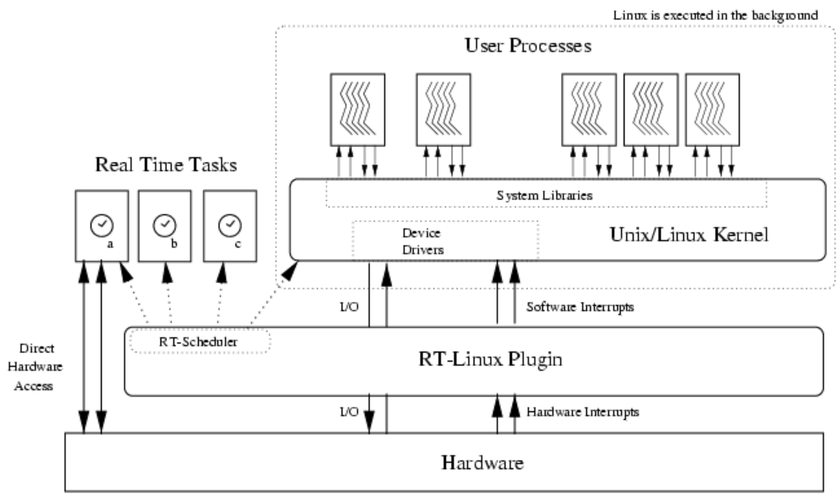

Figure 1.2: Detail of the RTLinux kernel

The abstraction layer introduced by RTLinux works by intercepting all hardware interrupts. Hardware interrupts not related to realtime activities are held and then passed to the Linux kernel as software interrupts when the RTLinux kernel is idle and the standard Linux kernel runs. Otherwise, the appropriate realtime interrupt service routine (ISR) is run. The RTLinux executive is itself nonpreemptible. Unpredictable delays within the RTLinux executive are eliminated by its small size and limited operations. Realtime tasks have two special attributes: they are "privileged" (that is, they have direct access to hardware), and they do not use virtual memory. Realtime tasks are written as special Linux modules that can be dynamically loaded into memory. They are are not expected to execute Linux system calls. The initialization code for a realtime tasks initializes the realtime task structure and informs RTLinux of its deadline, period, and release-time constraints. Non-periodic tasks are supported through the use of interrupts.

In contrast with some other approaches to realtime, RTLinux leaves the Linux kernel essentially untouched. Via a set of relatively simple modifications, it manages to convert the existing Linux kernel into a hard realtime environment without hindering future Linux development.

Chapter 2

The Basic API: Writing RTLinux Modules

This chapter Introduces critical concepts that must be grasped in order to successfully write RTLinux modules. It also presents the basic Application Programming Interface (API) used in all RTLinux programs. Then it steps the user through the creation of a basic "Hello World" programming example, which is intended to help the user in developing their very first RTLinux program.

2.1 Understanding an RTLinux Program

In the latest versions of RTLinux, programs are not created as standalone applications. Rather, they are modelled as modules which are loaded into the Linux kernel space. A Linux module is nothing but an object file, usually created with the `-c` flag argument to `gcc`. The module itself is created by compiling an ordinary C language file in which the `main ()` function is replaced by a pair of `init/cleanup` functions:

```
int init_module();
void cleanup_module();
```

As its name implies, the `init_module ()` function is called when the module is first loaded into the kernel. It should return 0 on success and a negative value on failure. Similarly, the `cleanup_module` is called when the module is unloaded.

For example, if we assume that a user has created a C file named `my_module.c`, the code can be converted into a module by typing the following:

```
gcc -c {SOME-FLAGS} my_module.c
```

This command creates a module file named `my_module.o`, which can now be inserted into the kernel. To insert the module into the kernel, we use the `insmod` command. To remove it, the `rmmod` command is used.

Documentation for both of these commands can be accessed by typing:

`man 8 insmod`*, and*
`man 8 rmmod`*.*

Here, the "8" forces the man command to look for the manual pages associated with system administration. From now on, we will refer to commands by their name and manual category. Using this format, these two commands would be referred to as `insmod (8)` *and* `rmmod (8)`*.*

For further information on running RTLinux programs, refer to Appendix A.

2.2 The Basic API

Now that we understand the general structure of modules, and how to load and unload them, we are ready to look at the RTLinux API.

2.2.1 Creating RTLinux POSIX Threads

A realtime application is usually composed of several "threads" of execution. Threads are light-weight processes which share a common address space. Conceptually, Linux kernel control threads are also RTLinux threads (with one for each CPU in the system). In RTLinux, all threads share the Linux kernel address space.

To create a new realtime thread, we use the `pthread_create(3)` function. This function must only be called from the Linux kernel thread (i.e., using `init_module()`):

```
#include <pthread.h>
int pthread_create(pthread_t * thread,
                    pthread_attr_t * attr,
                    void *(*start_routine)(void *),
                    void * arg);
```

The thread is created using the attributes specified in the "`attr`" thread attributes object. If `attr` is `NULL`, default attributes are used. For more detailed information, refer to the POSIX functions:

- `pthread_attr_init(3)`,
- `pthread_attr_setschedparam(3)`, and
- `pthread_attr_getschedparam(3)`

as well as these RTL-specific functions:

- `pthread_attr_getcpu_np(3)` , and
- `pthread_attr_setcpu_np(3)`

which are used to get and set general attributes for the scheduling parameters and the CPUs in which the thread is intended to run.

The ID of the newly created thread is stored in the location pointed to by "`thread`". The function pointed to by `start_routine` is taken to be the thread code. It is passed the "`arg`" argument.

To cancel a thread, use the POSIX function:

```
pthread_cancel(pthread thread);
```

You should join the thread in `cleanup_module` *with* `pthread_join()` *for its resources to be deallocated.*

You must make sure the thread is cancelled before calling `pthread_join()` *from* `cleanup_module()`. *Otherwise, Linux will hang waiting for the thread to finish. If unsure, use* `pthread_delete_np(3)` *instead of* `pthread_cancel()/pthread_join()`.

2.2.2 Time Facilities

RTLinux provides several clocks that can be used for timing functionality, such as as referencing for thread scheduling and obtaining timestamps. Here is the general timing API:

```
#include <rtl_time.h>

int clock_gettime(clockid_t clock_id, struct timespec *ts);
hrtime_t clock_gethrtime(clockid_t clock);

struct timespec {
   time_t tv_sec; /* seconds */
   long tv_nsec; /* nanoseconds */
};
```

To obtain the current clock reading, use the `clock_gettime(3)` function where `clock_id` is the clock to be read and `ts` is a structure which stores the value obtained.

The `hrtime_t` value is expressed as a single 64-bit number of nanoseconds. Thus, `clock_gethrtime(3)` is the same as `clock_gettime`, but returns the time as an `hrtime_t` rather than as a `timespec` structure.

2.2.3 Conversion Routines

Several routines exist for converting from one form of time reporting to the other:

```
#include <rtl_time.h>

hrtime_t timespec_to_ns(const struct timespec *ts);
struct timespec timespec_from_ns(hrtime_t t)
const struct timespec * hrt2ts(hrtime_tvalue);
```

These are especially useful macros for passing time values into `nanosleep`, `pthread_cond_timedwait` and the like.

Currently supported clocks are:

- CLOCK_MONOTONIC: This POSIX clock runs at a steady rate, and is never adjusted or reset.
- CLOCK_REALTIME: This is the standard POSIX realtime clock. Currently, it is the same as CLOCK_MONOTONIC. It is planned that in future versions of RTLinux this clock will give the world time.
- CLOCK_RTL_SCHED: The clock that the scheduler uses for task scheduling.

The following clocks are architecture-dependent. They are not normally found in user programs.

- CLOCK_8254: Used on non-SMP x86 machines for scheduling.
- CLOCK_APIC: Used on SMP x86 machines.
- CLOCK_APIC: corresponds to the local APIC clock of the processor that executes `clock_gettime`. You cannot read or set the APIC clock of other processors.

2.2.4 Scheduling Threads

RTLinux provides scheduling, which allows thread code to run at specific times. RTLinux uses a pure priority-driven scheduler, in which the highest priority (ready) thread is always chosen to run. If two threads have the same priority, which one is chosen is undefined. RTLinux uses the following scheduling API:

```
int pthread_setschedparam(pthread_t thread,
                           int policy,
                           const struct sched_param *param);
int pthread_make_periodic_np(pthread_t thread,
                              const struct itimerspec *its);
int pthread_wait_np(void);
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);

struct itimerspec {
   struct timespec it_interval; /* timer period */
   struct timespec it_value;    /* timer expiration */
};
```

Thread priority can be modified at thread creation time by using:

`pthread_attr_setschedparam(3)`

or afterwards by using

`pthread_setschedparam(3)` .

The policy argument is currently not used in RTLinux, but should be specified as `SCHED_FIFO` for compatibility with future versions. The structure `sched_param` contains the `sched_priority` member. Higher values correspond to higher priorities. Use:

- `sched_get_priority_max(3)` , and
- `sched_get_priority_min(3)`

to determine possible values of `sched_priority`.

To make a realtime thread execute periodically, users may use the *non-portable*[1] function:

`pthread_make_periodic_np(3)`

which marks the thread as periodic. Timing is specified by the `itimer` structure `its`. The `it_value` member of the passed struct `itimerspec` specifies the time of the first invocation; the `it_interval` is the thread period. Note that when setting up the period for task **T**, the period specified in the `itimer` structure can be 0. This means that task **T** will execute only once.

The actual execution timing is performed by use of the function:

[1]It is possible to have threads execute periodically within RTLinux by using the pure POSIX API. However, this scheme is quite lengthy. This particular function has been added, therefore, to reduce user development time.

```
pthread_wait_np(3)
```

This function suspends the execution of the calling thread until the time specified by:

```
pthread_make_periodic_np(3)
```

In the next section we'll put the API to practical use.

2.3 A Simpl “Hello World” RTLinux program

We'll now write a small program that uses all of the API that we've learned thus far. This program will execute two times per second, and during each iteration it will print the message:

```
I'm here, my arg is 0
```

2.3.1 Code Listing

Save the following code under the filename `hello.c`:

```
#include <rtl.h>
#include <time.h>
#include <pthread.h>

pthread_t thread;
void * start_routine(void *arg) {
   struct sched_param p;
   p . sched_priority = 1;
   pthread_setschedparam (pthread_self(), SCHED_FIFO, &p);
   pthread_make_periodic_np (pthread_self(), gethrtime(),
                             500000000);

   while (1) {
     pthread_wait_np();
     rtl_printf("I'm here; my arg is %x\n", (unsigned) arg);
   }
   return 0;
}

int init_module(void) {
   return pthread_create (&thread, NULL, start_routine, 0);
}

void cleanup_module(void) {
   pthread_cancel (thread);
   pthread_join (thread, NULL);
}
```

❗ *This program can be found in* `examples/hello`.

Now, let's analyze the code.

2.3.2 Dissecting "Hello World"

In our program, the

```
init_module()
```

function begins the entire process by creating our execution thread – embodied in the function `start_routine()` – with an argument of 0 passed to `start_routine()`.

`start_routine` has three components: initialization, run-time and termination – best understood as the blocks before, during and after the `while()` loop, respectively.

Upon the first call to the newly-created thread `start_routine()`, the initialization section tells the scheduler to assign this thread a scheduling priority of 1 (one) with the call to `p.sched_priority`. Next, the thread sets the scheduler's behavior to be SCHED_FIFO for all subsequent executions with the call to `pthread_setschedparam`. Finally, by calling the function:

```
pthread_make_periodic_np()
```

the thread tells the scheduler to periodically execute this thread at a frequency of 2Hz (500 microseconds). This marks the end of the initialization section for the thread.

The `while()` loop begins with a call to the function:

```
pthread_wait_np()
```

which blocks all further execution of the thread until the scheduler calls it again. Once the thread is called again, it executes the rest of the contents inside the while loop, until it encounters another call to:

```
pthread_wait_np()
```

Because we haven't included any way to exit the loop, this thread will continue to execute forever at a rate of 2Hz. The only way to stop the program is by removing it from the kernel with the `rmmod(8)` command.

2.3.3 Compiling and Executing “Hello World”

In order to execute our program, we must first do the following:

1. *Compile the source code and create a module.* We can normally accomplish this by using the Linux GCC compiler directly from the command line. To simplify things, however, we'll create a Makefile. Then we'll only need to type “make” to compile our code.

2. *Locate and copy the* `rtl.mk` *file.* The `rtl.mk` file is an include file which contains all the flags needed to compile our code. For simplicity, we'll copy it from the RTLinux source tree and place it alongside of our `hello.c` file.

3. *Insert the module into the running RTLinux kernel.* The resulting object binary must be “plugged in” to the kernel, where it will be executed by RTLinux.

Let's look at these steps in some detail.

We begin by creating the Makefile that will be used to compile our `hello.c` program. Type the following into a file called `Makefile` and put it in the same directory as your `hello.c` program:

```
hello.o: hello.c
     gcc $(CFLAGS) hello.c
```

If you haven't already done so, locate the file `rtl.mk` and copy it into the same directory as your `hello.c` and `Makefile` files. The `rtl.mk` file can usually be found at `/usr/include/rtlinux/rtl.mk`.

```
cp /usr/include/rtlinux/rtl.mk .
```

(Note the trailing dot (.).)

Now, type the following:

```
make -f rtl.mk hello.o
```

This compiles the `hello.c` program and produces an object file named `hello.o`.

We now need to load the RTLinux modules. There are several ways to do this. The easiest is to use the `rtlinux(1)` command (as root):

```
rtlinux start hello
```

You can check the status of your modules by typing the command:

```
rtlinux status hello
```

For more information about the usage of the `rtlinux(1)` command, refer to its man page, or type:

```
rtlinux help
```

You should now be able to see your `hello.o` program printing its message twice per second. Depending on the configuration of your machine, you should either be able to see it directly in your console, or by typing:

```
dmesg
```

To stop the program, we need to remove it from the kernel. To do so, type:

```
rtlinux stop hello
```

For other ways on running RTLinux programs, refer to Appendix A.

Congratulations, you have now successfully created and run your very first RTLinux program!

Chapter 3

The Advanced API: Getting More Out of Your RTLinux Modules

RTLinux has a rich assortment of functions which can be used to solve most realtime application problems. This chapter describes some of the more advanced concepts.

3.1 Using Floating Point Operations in RT-Linux POSIX Threads

The use of floating-point operations in RTL POSIX threads is prohibited by default. The RTL-specific function `pthread_setfp_np(3)` is used to change the status of floating-point operations.

```
int pthread_setfp_np (pthread_tthread, int flag);
```

To enable FP operations in the thread, set the flag to 1. To disable FP operations, pass 0.

The `examples/fp` directory contains several examples of tasks which use floating point and the math library.

3.2 RTLinux Inter-Process Communication (IPC)

The general philosophy of RTLinux requires the realtime component of an application to be lightweight, small and simple. Applications should be split in such a way that, as long as timing restrictions are met, most of the work is done in user space. This approach makes for easier debugging and better understanding of the realtime part of the system. Consequently, communication mechanisms are necessary to interface RTLinux tasks and Linux.

RTLinux provides several mechanisms which allow communication between realtime threads and user space Linux processes. The most important are realtime FIFOs and shared memory.

3.2.1 Using Real-Time FIFOs

Realtime FIFOs are First-In-First-Out queues that can be read from and written to by Linux processes and RTLinux threads. FIFOs are uni-directional – you can use a pair of FIFOs for bi-directional data exchange. To use the FIFOs, the `system/rtl_posixio.o` and `fifos/rtl_fifo.o` Linux modules must be loaded in the kernel.

RT-FIFOs are Linux character devices with the major number of 150. Device entries in `/dev` are created during system installation. The device file names are `/dev/rtf0`, `/dev/rtf1`, etc., through `/dev/rtf63` (the maximum number of RT-FIFOs in the system is configurable during system compilation).

Before a realtime FIFO can be used, it must be initialized:

```
#include <rtl_fifo.h>
int rtf_create(unsigned int fifo, int size);
int rtf_destroy(unsigned int fifo);
```

`rtf_create` allocates the buffer of the specified size for the fifo buffer. The `fifo argument corresponds to the minor number of the device. rtf_destroy deallocates the FIFO.`

❗ *These functions must only be called from the Linux kernel thread (i.e., from* `init_module()`*).*

After the FIFO is created, the following calls can be used to access it from RTLinux threads: open(2) , read(2) , write(2) and close(2) . Support for other STDIO functions is planned for future releases.

Current implementation requires the FIFOs to be opened in non-blocking mode (O_NONBLOCK) by RTL threads.

You can also use the RTLinux-specific functions rtf_put (3) and rtf_get (3) .

Linux processes can use UNIX file IO functions without restriction. See the examples/measurement/rt_process.c example program for a practical application of RT-FIFOs.

3.2.2 Using Shared Memory

For shared memory, you can use the excellent mbuff driver by Tomasz Motylewski (motyl@chemie.unibas.ch. It is included with the RTLinux distribution and is installed in the drivers/mbuff directory. A manual is included with the package. Here, we'll just briefly describe the basic mode of operation.

First, the mbuff.o module must be loaded in the kernel. Two functions are used to allocate blocks of shared memory, connect to them and eventually deallocate them.

```
#include <mbuff.h>
void * mbuff_alloc(const char *name, int size);
void mbuff_free(const char *name, void * mbuf);
```

The first time mbuff_alloc is called with a given name, a shared memory block of the specified size is allocated. The reference count for this block is set to 1. On success, the pointer to the newly allocated block is returned. NULL is returned on failure. If the block with the specified name already exists, this function returns a pointer that can be used to access this block and increases the reference count.

mbuff_free deassociates mbuff from the specified buffer. The reference count is decreased by 1. When it reaches 0, the buffer is deallocated.

These functions are available for use in both Linux processes and the Linux kernel threads.

mbuff_alloc *and* mbuff_free *cannot be used from real-time threads. You should call them from* init_module *and* cleanup_module *only.*

3.2.3 Waking and Suspending RTLinux Threads

Interrupt-driven RTLinux threads can be created using the thread wakeup and suspend functions:

```
int pthread_wakeup_np(pthread_t thread);
int pthread_suspend_np(void);
```

The general idea is that a threaded task can be either awakened or suspended from within an interrupt service routine.

An interrupt-driven thread calls pthread_suspend_np(pthread_self()) and blocks. Later, the interrupt handler calls pthread_wakeup_np(3) for this thread. The thread will run until the next call to pthread_suspend_
. An example can be found in examples/sound/irqthread.c.

Another way to implement interrupt-driven threads is to use semaphores. See examples/measurements/irqsema.c for examples of this method.

3.2.4 Mutual Exclusion

Mutual exclusion refers to the concept of allowing only one task at a time (out of many) to read from or write to a shared resource. Without mutual exclusion, the integrity of the data found in that shared resource could become compromised. Refer to the appendix for further information on mutual exclusion.

RTLinux supports the POSIX pthread_mutex_ family of functions (include/rtl_mutex.h). Currently the following functions are available:

- pthread_mutexattr_getpshared(3)
- pthread_mutexattr_setpshared(3)
- pthread_mutexattr_init(3)
- pthread_mutexattr_destroy(3)
- pthread_mutexattr_settype(3)
- pthread_mutexattr_gettype(3)
- pthread_mutex_init(3)
- pthread_mutex_destroy(3)
- pthread_mutex_lock(3)
- pthread_mutex_trylock(3)
- pthread_mutex_unlock(3)

The supported mutex types include:

- PTHREAD_MUTEX_NORMAL (default POSIX mutexes) and
- PTHREAD_MUTEX_SPINLOCK (spinlocks)

See examples/mutex for a test program. POSIX semaphores are also supported. An example using POSIX semaphores can be found in examples/-mutex/sema_test.c.

3.3 Accessing Physical Memory and I/O Ports from RTLinux Threads

These capabilities are essential for programming hardware devices in the computer. RTLinux, just like ordinary Linux, supports the /dev/mem device (**man 4 mem**) for accessing physical memory from RTLinux threads. The rtl_posixio.o module must be loaded. The program opens /dev/mem, *mmaps* it, and then proceeds to read and write the mapped area. See examples/mmap for an example.

In a module, you can call `mmap` *from Linux mode only (i.e., from* `init_module()`*). Calling* `mmap` *from RT-threads will fail.*

Another way to access physical memory is via Linux's **ioremap** call:

```
char *ptr = ioremap(PHYS_AREA_ADDRESS, PHYS_AREA_LENGTH);
...
ptr[i] = x;
```

IO port access functions (specifically for x86 architecture) are as follows:

- Output a byte to a port:

  ```
  #include <asm/io.h>
  void outb(unsigned int value, unsigned short port)
  void outb_p(unsigned int value, unsigned short port)
  ```

- Output a word to a port:

  ```
  #include <asm/io.h>
  void outw(unsigned int value, unsigned short port)
  void outw_p(unsigned int value, unsigned short port)
  ```

- Read a byte from a port:

  ```
  #include <asm/io.h>
  char inb(unsigned short port)
  char inb_p(unsigned short port)
  ```

- Read a word from a port:

  ```
  #include <asm/io.h>
  short inw(unsigned short port)
  short inw_p(unsigned short port)
  ```

Functions with the ''_p'' suffix (e.g., outb_p) provide a small delay after reading or writing to the port. This delay is needed for some slow ISA devices on fast machines. (See also the Linux I/O port programming mini-HOWTO).

Check out examples/sound to see how some of these functions are used to program the PC realtime clock and the speaker.

3.4 Soft and Hard Interrupts

There are two types of interrupts in RTLinux: hard and soft.

Soft interrupts are normal Linux kernel interrupts. They have the advantage that some Linux kernel functions can be called from them safely. However, for many tasks they do not provide hard realtime performance; they may be delayed for considerable periods of time.

Hard interrupts (or realtime interrupts), on the other hand, have much lower latency. However, just as with realtime threads, only a very limited set of kernel functions may be called from the hard interrupt handlers.

3.4.1 Hard Interrupts

The two functions:

- rtl_request_irq(3) and
- rtl_free_irq(3)

are used for installing and uninstalling hard interrupt handlers for specific interrupts. The manual pages describe their operation in detail.

```
#include <rtl_core.h>
int rtl_request_irq(unsigned int irq,
                     unsigned int (*handler) (unsigned int,
                     struct pt_regs *));
int rtl_free_irq(unsigned int irq);
```

3.4.2 Soft interrupts

```
int rtl_get_soft_irq(
         void (*handler)(int, void *, struct pt_regs *),
         const char * devname);
void rtl_global_pend_irq(int ix);
void rtl_free_soft_irq(unsigned int irq);
```

The rtl_get_soft_irq(3) function allocates a virtual irq number and installs the handler function for it. This virtual interrupt can later be triggered using rtl_global_pend_irq(3) . rtl_global_pend_irq is safe to use from realtime threads and realtime interrupts. rtl_free_soft frees the allocated virtual interrupt.

Note that soft interrupts are used in the RTLinux FIFO implementation (fifos/rtl_fifo.c).

Chapter 4

Special Topics

You may never find yourself needing to know any of the following. Then again, you might.

4.1 Symmetric Multi-Processing Considerations

From the point of view of thread scheduling, RTLinux implements a separate UNIX process for each active CPU in the system. In general, thread control functions can only be used for threads running on the local CPU. Notable exceptions are:

- int pthread_wakeup_np(pthread_t thread) : wake up suspended thread
- int pthread_cancel (pthread_t thread) : cancel thread
- int pthread_join(pthread_t thread) : wait for thread to finish
- int pthread_delete_np (pthread_t thread) : kill the thread

By default, a thread is created to run on the current CPU. To assign a thread to a particular CPU, use the pthread_attr_setcpu_np(3) function to set the CPU pthread attribute. See examples/mutex/-mutex.c.

4.2 RTLinux Serial Driver (rt_com)

rt_com(3) is a driver for 8250 and 16550 families of UARTs commonly used in PCs (COM1, COM2, etc.). The available API is as follows:

```
#include <rt_com.h>
#include <rt_comP.h>
void rt_com_write(unsigned int com, char *pointer, int cnt);
int rt_com_read(unsigned int com, char *pointer, int cnt);
int rt_com_setup(unsigned int com, unsigned int baud,
                 unsigned int parity, unsigned int stopbits,
                 unsigned int wordlength);



#define RT_COM_CNT n
struct rt_com_struct
{
   int magic;                 // unused
   int baud-base;             // base-rate; 11520
                              // (BASE_BAUD in rt_comP.h;
                              // for standard ports.
   int port;                  // port number
   int irq;                   // interrupt number (IRQ)
                              //    for the port
   int flag;                  // flags set for this port
   void (*isr)(void)          // address of the interrupt
                              //    service routine
   int type;                  //
   int ier;                   // a copy of the IER register
   struct rt_buf_struct ibuf; // address of the port input
                              //    buffer
   struct rt_buf_struct obuf; // address of the port output
                              //    buffer
} rt_com_table [RT_COM_CNT];
```

where

- rt_com_write(3) - writes cnt characters from buffer ptr to the realtime serial port com.

- rt_com_read(3) - attempts to read cnt characters to buffer ptr from the realtime serial port com.
- rt_com_setup(3) - is used to dynamically change the parameters of each realtime serial port.

rt_com is a Linux module. The user must specify relevant serial port information via entries in rt_com_setup. In addition, the user must specify -- via entries in the rt_com_table (located in rt_com.h) -- the following:

- Number of serial ports available (n)
- Serial ports and relevant parameters for each, and
- An ISR to be executed when the port irq fires.

When rt_com (3) is installed with either insmod(8), modprobe(8) or rtlinux(1) , its init_module() function (in rt_com.c) requests the port device memory, registers the ISR and sets various default values for each port entry in rt_com_table.

4.3 Interfacing RTLinux Components to Linux

RTLinux threads, sharing a common address space with the Linux kernel, can in principle call Linux kernel functions. This is usually *not* a safe thing to do, however, because RTLinux threads may run even while Linux has interrupts disabled. Only functions that do not modify Linux kernel data structures (e.g., vsprintf) should be called from RTLinux threads.

RTLinux provides two delayed execution mechanisms to overcome this limitation: soft interrupts and task queues.

The RTLinux white paper discusses this topic in more detail.

4.4 Writing RTLinux Schedulers

Most users will never be required to write a scheduler. Future versions of RTLinux are expected to have a fully customizable scheduler, but in the meantime, here are some points to help the rest of you along:

- The scheduler is implemented in the scheduler/rtl_sched.c file
- The scheduler's architecture-dependent files are located in include/arch-i386 and scheduler/i386
- The scheduling decision is taken in the rtl_schedule() function. Thus, by modifying this function, it is possible to change the scheduling policy.

Further questions in this area may be addressed directly to the FSM Labs Crew.

Appendix A

Running RTLinux Programs

Your RTLinux distribution comes complete with several examples in the examples/ sub-directory. These examples are useful, not only for testing your brand new RTLinux distribution, but for helping get you started writing your own RTLinux programs.

A.1 General

Before you will be able to run any RTLinux programs, you must first insert the RTLinux scheduler and support modules in the modules into the Linux kernel. Use any of the following:

- **rtlinux(1)** script, the preferred method,
- **insmod(8)**,
- **modprobe(8)**, or
- the insrtl script file that has been supplied for you in the scripts directory.

For more information on Linux modules and how to manipulate them, see the Linux Kernel-HOWTO .

The following sections describe each of these methods in more detail.

A.2 Examples

A.2.1 Using `rtlinux`

Beginning with RTLinux 3.0-pre9, users can load and remove user modules by using the rtlinux(1) command. To insert, remove, and obtain status information about RTLinux modules, use the following commands:

```
rtlinux start my_program
rtlinux stop my_program
rtlinux staus my_program
```

For further information on the the rtlinux(1) script, type either:

```
man 1 rtlinux
```

or

```
rtlinux help.
```

A.2.2 Using `modprobe`

all the RTLinux modules, type the following:

```
modprobe -a rtl rtl_time rtl_sched rtl_posixio rtl_fifo
```

❗ *Using modprobe requires that modules be installed in* `/lib/modules/`*`kernel_version`*.

A.2.3 Using `insmod` and `rmmod`

Suppose we have the appropriately named my_program.o. Assuming that all the appropriate RTLinux modules have already been loaded, all that's left to do is to load this module into the kernel:

```
insmod my_program.o
```

To stop the program, all we need do is type:

```
rmmod my_program
```

Appendix B

The RTLinux API at a Glance

Some paths to be aware of:

- RTLinux is installed in the directory /usr/rtlinux-xxx, where xxx is the version number. To simplify future development, a symbolic link has been created as /usr/rtlinux which points to /usr/rtlinux-xxx. Users are encouraged to specify their paths via this symbolic link to maintain future compatibility with new RTLinux versions.

- /usr/rtlinux/include contains all the include files necessary for development projects.

- /usr/rtlinux/examples contains the RTLinux example programs, which illustrate the use of much of the API.

- /usr/doc/rtlinux/man contains the manual pages for RTLinux.

- /usr/rtlinux/modules contains the core RTLinux modules.

- /usr/rtlinux/bin contains RTLinux scripts and utilities.

The following sections provide a listing of the various utilities and APIs available in RTLinux.

B.1 Getting Around

There are several manual pages which give overviews on the technology and the APIs.

- rtl_v1 (3) : RTLinux facilities for RTLinux v1.x.

The RTLinux V1 API is presented exclusively for backwards compatibility. It is no longer recommended for new projects. Users are strongly discouraged from starting any new projects with this API.

- rtf (4) : realtime fifo devices
- rtl_index (4) : A comprehensive list of RTLinux functions.
- rtlinux (4) : A general roadmap and description to RTLinux

B.2 Scripts and Utilities

The following utilities are designed to make your programming job easier.

- rtl-config (1) : script used to get information about the installed version of RTLinux, cflags, include paths, and documentation paths.
- rtlinux (1) : SysV compatible script used to start RTLinux and load the user's RTLinux modules

B.3 Core RTLinux API

Here is the main RTLinux API. You are encouraged to use this API for all new projects.

- clock_gethrtime (3) : get high resolution time using the specified clock

- clock_gettime : clock and timer functions
- clock_settime : clock and timer functions
- gethrtime (3) : get high resolution time
- nanosleep : high resolution sleep
- pthread_attr_getcpu_np (3) : examine and change the CPU pthread attribute
- pthread_attr_getschedparam : dynamic thread scheduling parameters access
- pthread_attr_getdetachstate : get detachstate attributes
- pthread_attr_getstacksize : get stacksize attribute
- pthread_attr_init : initialize threads attribute object
- pthread_attr_setcpu_np (3) : examine and change the CPU pthread attribute
- pthread_attr_setdetachstate : set detachstate attributes
- pthread_attr_setfp_np (3) : set and get floating point enable attribute
- pthread_attr_setschedparam : dynamic thread scheduling parameters access
- pthread_attr_setstacksize : set stacksize attribute
- pthread_cancel (3) : stop and cancel a thread (not recommended)
- pthread_create (3) : create a thread
- pthread_condattr_destroy : destroy condition variable attributes object
- pthread_condattr_getpshared : get the process-shared condition variable attributes

- pthread_condattr_init : initialize condition variable attributes object
- pthread_condattr_setpshared : set the process-shared condition variable attributes
- pthread_cond_broadcast : broadcast a condition
- pthread_cond_destroy : destroy condition variable
- pthread_cond_init : initialize condition variable
- pthread_cond_signal : signal a condition
- pthread_cond_timedwait : wait on a condition variable
- pthread_cond_wait : wait on a condition variable
- pthread_delete_np (3) : delete a realtime thread
- pthread_exit : thread termination
- pthread_join (3) : terminate a thread
- pthread_kill (3) : send a signal to a thread
- pthread_linux (3) : get the thread identifier of the Linux thread
- pthread_make_periodic_np (3) : mark a realtime thread as periodic
- pthread_mutexattr_destroy(3) : Destroys a mutex attribute object.
- pthread_mutexattr_getprioceiling : get priority ceiling attribute of mutex attribute object.
- pthread_mutexattr_getpshared : obtains the process-shared setting of a mutex attribute object.
- pthread_mutexattr_gettype : get the mutex type
- pthread_mutexattr_init : initializes a mutex attribute object.

- pthread_mutexattr_setprioceiling : set priority ceiling attribute of mutex attribute object.
- pthread_mutexattr_setpshared : sets the process-shared attribute of a mutex attribute object
- pthread_mutexattr_settype : set the mutex type
- pthread_mutex_destroy : destroys a mutex
- pthread_mutex_init(3) : initializes a mutex with the attributes specified in the specified mutex attribute object.
- pthread_mutex_lock : locks an unlocked mutex. If the mutex is already locked, the calling thread blocks until the thread that currently holds the mutex releases it.
- pthread_mutex_trylock : tries to lock a mutex. If the mutex is already locked, the calling thread returns without wating for the mutex to be freed.
- pthread_mutex_unlock : unlocks a mutex.
- pthread_getschedparam : get schedparam attribute
- pthread_self : get calling thread's ID
- pthread_setcancelstate : set cancelability state
- pthread_setschedparam : set schedparam attribute
- pthread_setfp_np (3) : allow use of floating-point operations in a thread.
- pthread_suspend_np (3) : suspend execution of a realtime thread.
- pthread_wait_np (3) : suspend the current thread until the next period
- pthread_wakeup_np (3) : wake up a realtime thread.
- rt_com (3) : serial port driver for RTLinux

- `rt_com_read (3) :  read data in realtime from a serial por`
- `rt_com_setup (3) :  dynamically change the parameters of each realtime serial port.`
- `rt_com_table (3) :  an array of descriptions, one per serial port.`
- `rt_com_write (3) :  write data in realtime to a serial port`
- `rtf_create (3) :  create a realtime fifo`
- `rtf_create_handler (3) :  install a handler for realtime fifo data`
- `rtf_create_rt_handler (3) :  install a handler for realtime fifo data`
- `rtf_destroy (3) :  remove a realtime fifo created with  rtf_create(3)`
- `rtf_flush (3) :  empty a realtime FIFO`
- `rtf_get (3) :  read data from a realtime fifo`
- `rtf_link_user_ioctl (3) :  install an ioctl (3) handler for a realtime FIFO.`
- `rtf_put (3) :  write data to a realtime fifo`
- `rtf_make_user_pair (3) :  make a pair of RT-FIFOs act like a bidirectional FIFO`
- `rtl_allow_interrupts (3) :  control the CPU interrupt state`
- `rtl_free_irq (3) :  install and remove realtime interrupt handlers`
- `rtl_free_soft_irq (3) :  install and remove software interrupt handlers`
- `rtl_get_soft_irq (3) :  install and remove software interrupt handlers`

- `rtl_getcpuid (3) :  get the current processor id`
- `rtl_getschedclock (3) :  get the current scheduler clock`
- `rtl_global_pend_irq (3) :  schedule a Linux interrupt`
- `rtl_hard_disable_irq (3) :  interrupt control`
- `rtl_hard_enable_irq (3) :  interrupt control`
- `rtl_no_interrupts (3) :  control the CPU interrupt state`
- `rtl_printf (3) :  print formatted output`
- `rtl_request_irq (3) :  install and remove realtime interrupt handlers`
- `rtl_restore_interrupts (3) :  control the CPU interrupt state`
- `rtl_setclockmode (3) :  set the RTLinux clock mode`
- `rtl_stop_interrupts (3) :  control the CPU interrupt state`
- `rtlinux_sigaction (3) :  RTLinux v3 User-Level signal handling functions.`
- `rtlinux_signal (3) :  list of available RTLinux User-Level signals`
- `rtlinux_sigprocmask (3) :  RTLinux v3 User-Level signal handling functions.`
- `rtlinux_sigsetops (3) :  RTLinux User-Level signal set operations`
- `sched_get_priority_max :  get priority limits for the scheduling policy`
- `sched_get_priority_min :  get priority limits for the scheduling policy`
- `sem_init :  initialize POSIX semaphore`
- `sem_destroy :  destroy an unnamed POSIX semaphore`
- `sem_getvalue :  get the value of a sempahore`

- sem_post : unlock a semaphore
- sem_trywait : lock a semaphore
- sem_wait : lock a semaphore
- sigaction (2) : RTLinux POSIX signal handling functions
- sysconf : get configurable system variables
- time : clock and timer functions
- uname : get name of current system
- usleep : suspend execution for an interval

B.4 Version 1.x API: Not for New Projects

The v1 API is exclusively for older RTLinux projects. It is NOT recommended for use with new projects. This listing is for backward compatibility only:

- free_RTirq (3) : uninstall an interrupt handler
- request_RTirq (3) : install an interrupt handler
- rt_get_time (3) : get time in ticks
- rt_task_delete (3) : delete a realtime task
- rt_task_init (3) : create a realtime task
- rt_task_make_periodic (3) : mark a realtime task for execution.
- rt_task_suspend (3) : suspend execution of a realtime task.
- rt_task_wait (3) : suspend execution for the current period until the next period
- rt_task_wakeup (3) : allow a previously suspended realtime task to run.
- rt_use_fp (3) : set/remove permission for task to use floating point unit