RTAI: a Beginner's Guide

What is a "real time system"?

A real time system can be defined as a *"system capable of guaranteeing timing requirements of the processes under its control"*.

It must be *fast* and *predictable*. *Fast* means that it has a low latency, i.e. it responds to external, asynchronous events in a short time. The lower the latency, the better the system will respond to events which require immediate attention. *Predictable* means that it is able to determine task's completion time with certainty.

Typically a real time system represents the *computer controlling system* that manages and coordinates the activities of a controlled system, that can be viewed as the environment with which the computer interacts. The interaction is bidirectional, say through various sensors (environment -> computer) and actuators (computer -> environment), and is characterized by *timing correctness constraints*.

It is desirable that *time-critical* and *non time-critical activities* coexist in a real time system. Both are called tasks and a task with a timeliness requirement is called a *real time task*.

Typically real time tasks have the following types of requirements and/or constraints. - *Timing constraints*. The most common are either *periodic* or *aperiodic*. An aperiodic task has a deadline by which it must finish or start, or it may have a constraint on both start and finish times. A periodic task has to be repeated once per period. Most sensory processing is periodic, while aperiodic requirements can arise from dynamic events.

- *Resource requirements*. A real time task may require access to certain resources such as I/O devices, data structures, files and databases.
- *Communication requirements*. Tasks should be allowed to communicate with messages.
- *Concurrency constraints*. Tasks should be allowed concurrent access to common resources providing the consistency of the resource is not violated.

What is RTAI?

RTAI means *Real Time Application Interface.* Strictly speaking, it is not a real time operating system, such as VXworks or QNX. It is based on the Linux kernel, providing the ability to make it fully pre-emptable.

Linux is a standard time-sharing operating system which provides good average performance and highly sophisticated services. Like other OS, it offers to the applications at least the following services:

- hardware management layer dealing with event polling or processor/peripheral interrupts
- scheduler classes dealing with process activation, priorities, time slice
- communications means among applications.

Linux suffers from a lack of real time support. To obtain a timing correctness behaviour, it is necessary to make some changes in the kernel sources, i.e. in the interrupt handling and scheduling policies. In this way, you can have a real time platform, with low latency and high predicatbility requirements, within full non real time Linux environment (access to TCP/IP, graphical display and windowing systems, file and data base systems, etc.).

RTAI offers the same services of the Linux kernel core, adding the features of an industrial real time operating system. It consists basically of an *interrupt dispatcher:* RTAI mainly traps the peripherals

interrupts and if necessary re-routes them to Linux. It is not an intrusive modification of the kernel; it uses the concept of HAL (*hardware abstractionlayer*) to get information from Linux and to trap some fundamental functions. This HAL provides few dependencies to Linux Kernel. This leads to a simple adaptation in the Linux kernel, an easy RTAI port from version to version of Linux and an easier use of other operating systems instead of RTAI. RTAI considers Linux as a background task running when no real time activity occurs.

Installing RTAI

Please have a look at the README.INSTALL file in the RTAI distribution for detailed instructions how to install RTAI.

Kernel Modules

RTAI is very much module oriented. So to understand and be able to use RTAI is necessary to know the dynamically loadable modules for Linux.

The Linux kernel design is similar to that of classic Unix systems: it uses a monolithic architecture with file systems, device drivers, and other pieces statically linked into the kernel image to be used at boot time. The use of dynamic kernel modules allows you to write portions of the kernel as separate objects that can be loaded and unloaded on a running system.

A kernel module is simply an object file containing routines and/or data to load into a running kernel. When loaded, the module code resides in the kernel's address space and executes entirely within the context of the kernel. Technically, a module can be any set of routines, with the one restriction that two functions, init_module() and cleanup_module(), must be provided. The first is executed once the module is loaded, and the second, before the module is unloaded from the kernel.

The main functions to use to load /unload and inspect kernel modules are contained in the modutils package. They are:

- /sbin/insmod (insert a module into the running kernel)
- /sbin/rmmod (remove a module from the running kernel)
- /sbin/lsmod (inspect modules in the running codes)

Note that to manage with kernel modules you have to be root.

Once a module is loaded, it passes to form part of the operating system, hence it can use all the functions and access all variables and structures of the kernel. Similarly the global symbols created are made available or exported to other modules. If you don't want to share some symbol (variable or function), you have to declare it as static.

A module is built from a "C" source. Here is the simplest example of a kernel module (simple1.c).

```
---------  simple1.c ------------
#include
#include
#include
int var = 20;
int init_module(void)
{
    printk("\nVariable value: %d \n\n", var);
    return 0;
}
void cleanup_module(void)
{
    printk("\n Bye \n\n");
}
-------------------------------
```

This can be compiled with:

```
gcc -c -D__KERNEL__ -DMODULE -o simple1 simple1.c
```

Note that the kernel offers a different version of printf() called printk(); this works almost identically to the first except that it sends the output to a kernel ring buffer. At any istant you can examine the contents of the buffer using the command dmesg.

If you want to set the value of the variable var at installation, include the following macro: MODULE_PARM(var,"i"); after its declaration.

To avoid the gcc command line for all sources, let's use the Makefile facility. Here is an example for two sources, i.e. simple1.c and simple2.c.

```
---------  Makefile -----------
CFLAGS = -D__KERNEL__ -DMODULE -c
OBJS = simple1 simple2
CC = gcc
all:
clean:
        rm
-----------------------------
```

If you want to add a math function, say sin(), to our simple module you have to use the linker to get the sin() function into the kernel space. The Makefile becomes:

```
---------  Makefile -----------
CFLAGS = -D__KERNEL__ -DMODULE -c
OBJS = simple1 simple_with_sin
CC = gcc
all:
simple_with_sin.o: simple_with_sin.c
        -o $@ $<
simple_with_sin: simple_with_sin.o
        ld -r -static -o $@ $< -lm
clean:
        rm simple_with_sin.o
-----------------------------
```

If you want to do some real work, you have to learn how to read/write to hardware registers. It's very simple.

☐ Include the header file containing the definition of the macros #include

☐ Use inb() to read a byte from a 8-bits port and outb() to write a byte to a 8-bits port.

☐ Use inw() and outw() for 16-bits ports.

☐ If you include these macros and include the io.h file nothing will happen unless you use -O2 compiler flag; this option force the macros to be expanded.

Here is a simple code that write a value to the parallel port.

```
------------- simplepp.c -----------------
#include
#include
#include
#include
#define LPT 0x378
int var = 20;
MODULE_PARM(var,"i");
void write_lpt (unsigned char byte)
{
        outb(byte, LPT);
}
```

```
int init_module(void)
{
        printk("\n Variable Value: %d\n\n", var);
        write_lpt((unsigned char) var & 0xff);
        return 0;
}
void cleanup_module(void)
{
        write_lpt((unsigned char) 0);
        printk("\n Bye \n\n");
}
------------------------------------------
```

RTAI Modules.

To use RTAI, you have to load the modules that implement whatever RTAI capabilities you need. According to 1.3 release, available are the following core modules:

- rtai
- rtai_sched
- rtai_fifos
- rtai_shm
- lxrt
- rtai_pqueue
- rtai_pthread
- rtai_utils

Let's examine one by one.

1) It is *the really core module* and nothing about the real time services can be done without it. `rtai` initializes all of its control variables and structures, makes a copy of the `idt_table` and of the Linux irq handlers entry addresses and initializes the interrupts chips (ic) management specific functions. But when you install `rtai` with the usual `insmod rtai` command nothing happens, as `rtai` is a dormant module. You must specifically mount it when is needed by other modules calling `rt_mount_rtai()` to activate it. You must unmount it as well when it is not required anymore, by calling `rt_umount_rtai()`, which put `rtai` back into its bed to sleep. The mount call activates `rtai` and, even if you do not use any of its services Linux work toward the hardware is filtered by `rtai`. The most important thing happening when you mount RTAI is that from that very instant Linux is no more in power of disabling/enabling interrupts. From that point on `rtai` will assure that interrupt enables/disables will be consistent intra Linux but Linux could be preempted at any time by the higher authority of `rtai`, the only master of the hardware.

2) The *real time scheduler module*, which is in charge of distributing the CPU to different tasks present in the system, including Linux. The scheduling occurs when tasks perform certain system calls and on timer handler activation (each 8254 interrupt) (for an explanation of timers and interrupts see [RTAI Timers and Interrupts](#)). The scheduler makes it elected the first highest priority task in a READY state. RTAI considers the priority `0` as the highest priority and `0x3fffFfff` the lowest. Linux is given priority `0x7fffFfff`. Given a priority level, the first initialized task will be the first elected and will run to completion unless a task with a higher priority is elected or it terminates or the task calls a blocking system function. RTAI supports both periodic and oneshot mode for the real time scheduler.
You have three different schedulers:
- *UP*, only for uniprocessors
- *SMP*, for multiprocessors

-*MUP*, only for multiprocessors
The scheduler services are:
- *Task functions*
- *Timing functions*
- *Semaphore functions*
- *Mailbox functions*
- *Intertask communication functions*
All the functions can be used with any scheduler. Note that when you load `rtai_sched`, automatically rtai is mounted.

3) The module that implements the fifo services for RTAI. Many applications appear to benefit from a synergy between the real-time system side and the Linux side, for example for managing the data logging and displaying. Simple fifo buffers are used to do this; they are called *real time fifos.* The real-time task interface includes creation, destruction, reading and writing functions, performed by the `rtai_fifos` module. Linux user processes, on the other hand, see rt-fifos as ordinary character devices. Note that on the module side you always have only non blocking put/get, so that any different policy should be enforced by using appropriate user handler functions. Available are an old and a new (strongly recommended) fifo implementation. The last one is based on the mailboxes concepts, symmetrically usable from kernel modules and Linux processes. Even if fifos are strictly no more required in RTAI, because of the availability of LXRT (see below), fifos are kept for both compatibility reasons and because they are very useful tools to be used to communicate with interrupt handlers, since they do not require any scheduler to be installed. In this sense you can see this new implementation of fifos as a kind of universal form of device drivers, since once you have your interrupt handler installed you can use fifo services to do all the rest.

4) The RTAI specific module that allows sharing memory among different real time tasks and Linux processes, simoultaneously (it is another mechanism available to users, in addition to fifos). The services are symmetrical, i.e. the same calls can be used both in real time tasks, i.e. within the kernel, and Linux processes. The first allocation does a real allocation, any subsequent call to allocate with the same name from Linux processes just maps the area to the user space or return the related pointer to the already allocated space in kernel space. Analogously the freeing calls have just the effect of unmapping till the last is done, as that is the one the really frees allocated memory. Clearly cooperating users have to use the same name.

5) The LX(Linux)RT(RealTime) module, which implements services to make available any of the RTAI schedulers functions to Linux processes, so that a fully symmetric implementation of real time services is possible. To state it more clearly, that means that you can share memory, send messages, use semaphores and timings: Linux<->Linux, Linux<->RTAI and, naturally, RTAI<->RTAI.

6x) Posix RTAI modules. `rtai_pthread.o` provides hard real-time threads, where each thread is a RTAI task. All threads execute in the same address space and hence can work concurrently on shared data. `rtai_pqueue.o` provides kernel-safe message queues.

Timers and Interrupts

Correct timing and interrupt management represent the really challenge of a real time system, and hence of RTAI. But how can I get time from a PC? What is an interrupt and how can I manage it? From now on we will refer to Intel architecture (RTAI 1.3 runs on x86 machines).

Timers

UPs provide a specific chip to solve the problem of generating accurate time delays under software control. It is called 8254 and is a programmable interval timer/counter, that can be treated as an

array of four I/O ports in the system software (from 0x40 to 0x43). Three are indipendent 16-bit counters and the fourth is a control register for mode programming. The programmer configures the 8254 to match his/her requirements (select the mode) and programs one of the counters for the desired delay. After this delay, the 8254 will interrupt the CPU. Note that the counters are fully indipendent, so each counter may operate in a different mode. Linux programs the timer with mode 2 (rate generator, periodic pace) and loads the counter0 with the macro HZ defined in /usr/src/linux/include/asm/param.h (usually 100 hz). The counter 2 is used instead for beeping frequency.

RTAI provides both a periodic (mode 2 of the 8254) and a oneshot timer (mode 0), using the counter0 to load the initial count. In the oneshot mode the clock is reprogrammed every interrupt, while in the periodic one it is programmed only at beginning and then generates interrupts periodically. It is up to you which one to choose in relation to the application at hand. The periodic mode is much more efficient when you have one or many (but with a common period) tasks that take regular samples, while the onsehot mode is more flexible, because it allows for example to time several tasks with no large common divisor in their periods or to trigger off some external event. Note that in the RTAI oneshot mode the time is measured on the base of the CPU time stamp clock (TSC) and neither on the 8254 chip, which is used only to generate oneshot interrupts. This allows to reprogram the counter with only 2 I/O instructions, i.e. approximately 3 us. Since the TSC is not available on 486 machines for them RTAI uses a form of emulation of the "read time stamp clock" (rdtsc) assembler instruction based on counter2 of the 8254. So you can use RTAI also on such machines. Be warned that the oneshot timer on 486 is a performance overkill because of the need of reading the tsc, i.e. 8254 counter2 in this case, 2/3 times. MPs provides another facility to get time. In addition to 8254 chip, which is one per box, there is a LOCAL APIC per CPU.

Interrupts

[... not yet available]

RTAI example 1

This program is developed for the RTAI-1.4 version. You can find a *Makefile*, one kernel module (*rt_process.c*) and a Linux process (*scope.c*). By means of the scipt *Run* you can run the program.

The program simply generates a sine signal and displays the istant values on the screen.

------------- MAKEFILE --------------------

all: scope rt_process.o

LINUX_HOME = /usr/src/linux

RTAI_HOME = /home/rtai-1.4

INCLUDE = -I/include -I/include

MODFLAGS = -D__KERNEL__ -DMODULE -O2 -Wall

scope: scope.c

 gcc -o $@ $<

rt_process.o: rt_process.c

 gcc -c -o $@ $<

clean:

 rm -f rt_process.o scope

-------------- RT_PROCESS.C ------------------

```c
\#include <linux/module.h>
\#include <asm/io.h>
\#include <math.h>
\#include <rtai.h>
\#include <rtai_sched.h>
\#include <rtai_fifos.h>

\#define TICK_PERIOD 1000000
\#define TASK_PRIORITY 1
\#define STACK_SIZE 10000
\#define FIFO 0

static RT_TASK rt_task;

static void fun(int t)
{
 int counter = 0;
 float sin_value;
 while (1) {
 sin_value = sin(2*M_PI*1*rt_get_cpu_time_ns()/1E9);
 rtf_put(FIFO, &counter, sizeof(counter));
 rtf_put(FIFO, &sin_value, sizeof(sin_value));
 counter++;
 rt_task_wait_period();
 }
}
int init_module(void)
{
 RTIME tick_period;
 rt_set_periodic_mode();
 rt_task_init(&rt_task, fun, 1, STACK_SIZE, TASK_PRIORITY, 1, 0);
 rtf_create(FIFO, 8000);
 tick_period = start_rt_timer(nano2count(TICK_PERIOD));
 rt_task_make_periodic(&rt_task, rt_get_time() + tick_period,  tick_period);
 return 0;
}
```

```c
void cleanup_module(void)
{
stop_rt_timer();
rtf_destroy(FIFO);
rt_task_delete(&rt_task);
return;
}
```

-------------------- SCOPE.C ----------------------

```c
\#include <stdio.h>
\#include <unistd.h>
\#include <sys/types.h>
\#include <sys/mman.h>
\#include <sys/stat.h>
\#include <fcntl.h>
\#include <signal.h>

static int end;
static void endme(int dummy) { end=1; }

int main (void)
{
int fifo, counter;
float sin_value;
if ((fifo = open("/dev/rtf0", O_RDONLY)) < 0) {
fprintf(stderr, "Error opening /dev/rtf0\n");
exit(1);
}
signal(SIGINT, endme);
while (!end) {
read(fifo, &counter, sizeof(counter));
read(fifo, &sin_value, sizeof(sin_value));
printf(" Counter : %d Seno : %f \n", counter, sin_value);
}
return 0;
}
```

-------------------- RUN ----------------------

sync

insmod /home/rtai-1.4/modules/rtai.o

insmod /home/rtai-1.4/modules/rtai_sched.o

insmod /home/rtai-1.4/modules/rtai_shm.o

insmod /home/rtai-1.4/modules/rtai_fifos.o

insmod rt_process.o

./scope

rmmod rt_process

rmmod rtai_shm

rmmod rtai_fifos

rmmod rtai_sched

rmmod rtai

RTAI example 2

This example is like the first one, but it uses the shared memory, instead of fifos, to communicate between Linux and the RTAI-layer. The common data space is declared in the header file*parameters.h*.

------------------ MAKEFILE -------------------------

all: scope rt_process.o

LINUX_HOME = /usr/src/linux

RTAI_HOME = /home/rtai-1.4

INCLUDE = -I/include -I/include

MODFLAGS = -D__KERNEL__ -DMODULE -O2 -Wall

scope: scope.c

 gcc -o $@ $<

rt_process.o: rt_process.c parameters.h

 gcc -c -o $@ $<

clean:

 rm -f rt_process.o scope

-------------------- RT_PROCESS.C ----------------------

\#include <linux/module.h>

\#include <asm/io.h>

\#include <math.h>

```c
\#include <rtai.h>
\#include <rtai_shm.h>
\#include <rtai_sched.h>
\#include "parameters.h"

static RT_TASK rt_task;
static struct data_str *data;

static void fun(int t)
{
 unsigned int count = 0;
 float seno,coseno;
 while (1) {
 data->indx_counter = count;
 seno = sin(2*M_PI*1*rt_get_cpu_time_ns()/1E9);
 coseno = cos(2*M_PI*1*rt_get_cpu_time_ns()/1E9);
 data->sin_value = seno;
 data->cos_value = coseno;
 count++;
 rt_task_wait_period();
 }
}
int init_module(void)
{
 RTIME tick_period;
 rt_set_periodic_mode();
 rt_task_init(&rt_task, fun, 1, STACK_SIZE, TASK_PRIORITY, 1, 0);
 data = rtai_kmalloc(nam2num(SHMNAM), sizeof(struct data_str));
 tick_period = start_rt_timer(nano2count(TICK_PERIOD));
 rt_task_make_periodic(&rt_task, rt_get_time() + tick_period, tick_period);
 return 0;
}
void cleanup_module(void)
{
 stop_rt_timer();
 rt_task_delete(&rt_task);
 rtai_kfree(nam2num(SHMNAM));
```

```c
 return;
}



---------------------- SCOPE.C ---------------------------
\#include <stdio.h>
\#include <unistd.h>
\#include <sys/types.h>
\#include <sys/mman.h>
\#include <sys/stat.h>
\#include <fcntl.h>
\#include <signal.h>
\#include <rtai_shm.h>
\#include "parameters.h"

static int end;
static void endme(int dummy) { end=1; }

int main (void)
{
 struct data_str *data;
 signal(SIGINT, endme);
 data = rtai_malloc (nam2num(SHMNAM),1);
 while (!end) {
 printf(" Counter : %d Sine : %f Cosine : %f \n", data->indx_counter, data->sin_value,  data->cos_value);
 }
 rtai_free (nam2num(SHMNAM), &data);
 return 0;
}



---------------- PARAMETERS.H ----------------------
\#define TICK_PERIOD 1000000
\#define TASK_PRIORITY 1
\#define STACK_SIZE 10000
\#define SHMNAM "MIRSHM"
```

```c
struct data_str
{
 int indx_counter;
 float sin_value;
 float cos_value;
};
```

----------------------- RUN -----------------------

```
sync
insmod /home/rtai-1.4/modules/rtai.o
insmod /home/rtai-1.4/modules/rtai_sched.o
insmod /home/rtai-1.4/modules/rtai_shm.o
insmod /home/rtai-1.4/modules/rtai_fifos.o
insmod rt_process.o
./scope
rmmod rt_process
rmmod rtai_shm
rmmod rtai_fifos
rmmod rtai_sched
rmmod rtai
```