

## Real Time Linux II



**Abstract:** In this second issue devoted to RT-Linux I will try to offer a more practical view of RT-Linux. Before getting into details though, I will give a quick review of a very recent real time operating system named Linux KURT.



### Ismael Ripoll

**About the Author:** Ph.D. from the Polytechnic University of Valencia in 1996. Professor in operating systems at the Department of DISCA . Research interests include real-time scheduling and operating systems. Linux user since 1994. Hobbies: Trekking through the Pyrenees mountains, skiing, and home electronics.

[Write to the Author](#)

**Contents:**

[KU Real Time Linux \(KURT\)](#)

[What is Real Time good for?](#)

[Loadable Modules](#)

[Real Time Program](#)

[Communications between tasks](#)

[Conclusion](#)

[References](#)

### KU Real Time Linux (KURT)

Early this year (1998) was released a new real time operating system based on Linux. KURT is a soft real time operating system (soft or firm), i.e. the scheduler tries to satisfy the requested times of execution, but if any task finishes later than expected it is not a real tragedy and nothing dramatic happens. KURT's real time tasks can take advantage of all the utilities of Linux, in contrast to RT-Linux tasks. The improvements -- modifications -- made to the kernel are:

- Improve the resolution of the system clock. In Linux-i386, the interrupt frequency of the clock is 10ms (100 times per second), and it is using this time resolution that the kernel makes control decisions and measures time. KURT uses the same mechanism for managing time as RT-Linux. It programs the clock chip (8254) to generate interrupts on-demand, instead of periodically. This way time resolutions of microseconds can be achieved.
- The scheduler has been modified to include a new scheduling policy, SCHED\_KURT, besides those already implemented by the Linux kernel which are those defined by POSIX: SCHED\_FIFO, SCHED\_RR y SCHED\_OTHER.
- New system calls have been added in order to take advantage of the new real time functionality.

Real time tasks are modules dynamically loaded.

One of the most characteristic features of KURT is its scheduling policy. A decision was made to implement a cyclic scheduler. This type of schedulers use a table named *plan* that contains all the scheduled actions: activation moment, task to execute, duration of the task, etc.. The table is built during the system design phase. Later during run-time the work of the scheduler consists merely on reading sequentially the table following its instructions. When the end of the table is reached, the scheduler goes back to the beginning and continues executing tasks --

thus the name of cyclic scheduler. This type of scheduler has many advantages:

- The scheduler is very simple to implement
- It is efficient
- After a plan is built it the feasibility of the system can be determined immediately (some researchers maintain that this is the only method to warranty 100% the correct performance of a STR)

The main difficulty resides on generating the plan itself. Furthermore every time any of the task's parameters is modified it is necessary to rebuild the plan, also the amount of memory necessary to store it is often very large.

## What is Real Time good for?

Perhaps many people believe that real time techniques are only used in NASA, or with missiles or artifacts of that sort. Although this was true a number of years ago, nowadays the situation has changed drastically -- and it is going to change even more -- due to the increasing integration of information systems and electronics into the every day life of people. Among the every day situations where we find Real-Time is in the field of telecommunications and multimedia applications. For example, if we would like our computer to replay a sound file stored in the hard disk, a program would have to continuously (or better periodically) read, uncompress and send the sound data to the sound card. If at the same time we listen to the music we are working with an application, say a word processor or simply compiling the Linux kernel, it is certain that there will be periodic moments of silence while the processor handles other tasks. If instead of sound, we were reproducing video on our system, the result would be a playback with intermittent frozen images. These type of systems are known as soft Real-Time (violation of an execution period does not lead to a disastrous result, but it does degrade the services offered by the system).

RT-linux applications go beyond normal real time applications. With RT-Linux we can take total control of the PC (I say PC and not computer because for the moment there is no implementation of RT-Linux for any other architecture) as for the case of MSDOS. During a real time tasks it is possible to access all the ports of the PC, install interruption handlers, temporally disable interruptions, ... in other words we can "crash" the system as if it were a Windows system. This opportunity is however very attractive to those of us that enjoy attaching little electronic "gadgets" to the computer.

## Loadable Modules

To understand and be able to use RT-Linux is necessary to know the dynamically loadable modules for Linux. Matt Welsh has written a complete article where he explains in detail every issue concerning modules.

## *What are they?*

In most implementations of UNIX, the only way to access the hardware (ports, memory, interruptions, etc.) is through special files and having installed previously

the device drivers. Eventhough there are many good books that explain how to write device drivers, it is often a long and boring job, since it is necessary to write numerous functions to link the driver to the system.

**Modules are "fragments of the operating system" that can be inserted and extracted at run-time.** When a program made of several source files is compiled, first each file is compiled separately to generate an object file ".o", then all objects are link together resolving all references and generating a single executable. Let us suppose that the object file containing the function `main` can be run, and that the operating system were able to load in memory and link together the rest of the object files just when they were necessary. Well, the Kernel is able to do this with itself. When Linux starts up only the executable `vmlinux` is loaded in memory, it contains the indispensable elements of the kernel, later at run-time it can load and unload selectively whatever module is needed.

Modules is an optional feature in the Linux kernel, this feature must be requested during the compilation of the kernel. The kernels of all the distributions I know have been compiled with the modules option active.

It is even possible to create new modules and load them without having to recompile nor restart the system.

Once a module is loaded, it passes to form part of the operating system therefore:

- It can use all the functions and access all variables and and structures of the kernel.
- The code of the module is executed with the maximum level of privilege of the processor. On the i386 architecture it is executed at ring level 0, as a result it can have any type of access to the input/output and execute privileged instructions.
- The memory for both program and data is mapped directly to physical memory, over which it is not possible to do "paging" or how it is incorrectly known "swapping". Then it is impossible to generate a page fault during the execution of a module.

As we can see, a dynamically loaded module already has some of the characteristics of a real time program: it avoids delays by page faults and it has access to all the hardware resources.

## ***How to build and use them?***

A module is built from a "C" source. Here is an example of a minuscule module (to perform most of the following commands it is necessary to be super-user, root):

### **example1.c**

```
#define MODULE
#include <linux/module.h>
#include <linux/cons.h>
static int output=1;

int init_module(void) {
```

```

    printk("Output= %d\n",output);
    return 0;
}
void cleanup_module(void){
    printk("Adiós, Bye, Chao, Ovuvar, \n");
}

```

To compile it we use these parameters:

```
# gcc -I /usr/src/linux/include/linux -O2 -Wall
-D__KERNEL__ -c example1.c
```

The option `-c` tells gcc it must stop after generating the object file and skip the link phase. The final result is a file named `example1.o`.

The kernel lacks of standard output, therefore we can not use the function `printf()`... instead the kernel offers its own version of this function named `printk()`, this works almost identically to the first except that it sends the output to a kernel ring buffer. It is in this buffer that all the messages of the system end up, in fact these are the messages we see when stating up the system. At any instant we can examine the contents of the buffer using the command `dmseg` or directly inspecting the file `/proc/kmsg`.

Notice the function `main()` is absent and in its place we find the function `init_module()` that does not take any parameters. `cleanup_module()` is the last function to be called before unloading a module. Module loading is performed with the command `insmod`

```
# insmod example1.o
```

Right now we have installed the module `example1` and executed its `init_module()` function. To see the results type:

```
# dmesg | tail -1
Output= 1
```

The command `lsmod` lists the modules currently loaded on the kernel:

```
# lsmod
Module      Pages      Used by:
example1    1           0
sb          6           1
uart401     2 [sb]       1
sound       16 [sb uart401] 0 (autoclean)
```

And finally we use `rmmod` to unload a module:

```
# rmmod example1
# dmesg | tail -2
Output= 1
Adiós, Bye, Chao, Orvua,
```

The output of `dmesg` shows to us that the `cleanup_module()` function has been

executed.

We only need now to know how to pass parameters to a module. There is nothing more surprisingly simple. We can assign values to the global variables by passing parameters to `insmod`. For example:

```
# insmod ejemplo1.o output=4
# dmesg | tail -3
Output= 1
Adíos, Bye, Chao, Orvua,
Output= 4
```

Now we know all we need about modules, let us go back to RT-Linux.

## Our first Real Time Program

First remember that to use RT-Linux we had to prepare the Linux kernel to support Real Time modules -- we explained this operation in the [previous article](#).

There are two ways to make use of RT-Linux:

1. As a classic Real Time system with a scheduler based on fixed priorities.
2. As a bare PC, something similar to what can be done under DOS: capture interruptions and have total control over the computer.

This time I will discuss how to use RT-Linux as a system with fixed priorities. The example we are going to see does nothing "useful", it just sets in motion a real time task (a simple loop):

### example2.c

```
#define MODULE
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/version.h>
#include <linux/rt_sched.h>
RT_TASK task;

void fun(int computo) {
    int loop,x,limit;
    limit = 10;
    while(1){
        for (loop=0; loop<computo; loop++)
            for (x=1; x<limit; x++);

        rt_task_wait();
    }
}

int init_module(void) {

    RTIME now = rt_get_time();

    rt_task_init(&task,fun, 50 , 3000, 1);
    rt_task_make_periodic(&task,
        now+(RTIME)(RT_TICKS_PER_SEC*4000)/1000000,
        (RTIME)(RT_TICKS_PER_SEC * 100)/1000000);
    return 0;
}
```

```

}

void cleanup_module(void){
    rt_task_delete(&task);
}

```

Once again we compile this example with the following command:

```
# gcc -I /usr/src/linux/include/linux -O2 -Wall
-D__KERNEL__ -D__RT__ -c example2.c
```

Since the program is a module, the entry point is the function `init_module()`. The first thing it does is to read the current time and store it in a local variable; the function `rt_get_time()` returns the number `RT_TICKS_PER_SEC` elapsed since booted time (in the current implementation `RT_TICKS_PER_SEC` is 1.193.180, giving a resolution of 0.838 micro-seconds). With `rt_task_init()` the "task" structure is initialized but it is not launched yet. The main program of the recently created task is `fun()`, which is the second parameter. The following parameter is the data value passed to the new task when it starts execution, notice that `fun()` expects an `int` parameter. Next parameter is the size of the task's stack; since every task has its own thread of execution, each one needs its own stack. The last parameter is the priority; in this case, with only one task in the system, we can set any value we wish.

`rt_task_make_periodic()` transforms the task into a periodic task. It takes two time values, first the instant in absolute time when the task will be activated for the first time and the second value is the period between successive activation's from the first one.

The real time task (function `fun()`), is an infinite loop in which there are only two actions: a loop that only wastes time and then calls `rt_task_wait()`. `rt_task_wait()`, is a function that suspends the execution of the task that invoked it until the next activation time, moment when the execution will continue at the instruction right after `rt_task_wait()`. The reader should realize that a periodic task is not executed from the beginning at each activation time, instead the task must suspend execution by itself (after finishing its work) and wait for the next activation. This allows to write a task that only performs a number of initializations the first time is called.

To execute `example2` we must first install the module `rt_prio_sched`, because our program needs the functions `rt_task_make_periodic()`, `rt_task_delete()` and `rt_task_init()`. The function `rt_get_time()` is not contained within the module but it is located in the Linux kernel and therefore there is not need to install it in order to use it.

```
# modprobe rt_prio_sched
# insmod ./example2.o
```

Given that `rt_prio_sched` is a module of the system, it was created during the compilation of the Linux kernel and it was therefore copied to the directory `/var/modules/2.0.33/`. We use the command `modprobe` because it is an easier tool to load modules (it searches the module through the directories of modules) (See `modprobe(1)`).

If all went well, with the command `lsmod` we will see that both modules were loaded correctly.

Well, at this point the reader already has a real time program running, Do you notice anything special? If the processor is a bit slow, the reader will likely notice that Linux runs slower than usual. You may try to increase the number of iterations of the inside loop of `fun()` by varying the third parameter in `rt_task_init()`. I recommend running the program `ico` to appreciate how much less processor time is left, because the time used by real time programs is to all effects like the processor worked at a lower cycle speed, Linux will believe that all its processes need more time to perform the same tasks. If the computation time (time required to execute all the iterations of the loop) is larger than 100 microseconds then Linux "hangs" because Linux is the background task and the real time task consumes 100% of the time. Actually Linux is not hang, it just doesn't have processor time.

## Communication Between Tasks

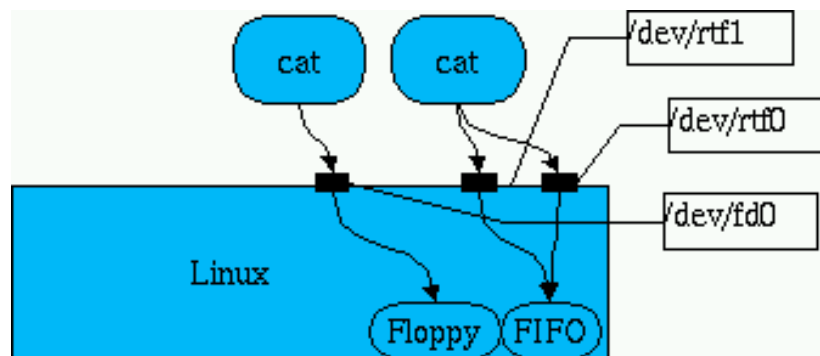
Under RT-Linux there is only one way of communication: Real-Time FIFO. The way it works is very similar to that of Unix PIPEs, a communication by stream of data without structure. A FIFO is a buffer of a fixed number of bytes on which reading and writing operations can be performed.

Using FIFOS it is possible to stablish inter-communication between real time tasks as well as among normal Linux tasks.

**From the point of view of a normal process** a FIFO is a special character file. Normally they are found as `/dev/rtf0`, `/dev/rtf1`, etc. these files do not exist in Linux so they must be created as follows:

```
# for i in 0 1 2 3; do mknod /dev/rtf$i c 63 $i; done
```

If more FIFOs are needed they can be easily created using the same procedure `rtf4`, `rtf5`, etc.. The special files act as interface to a handler on the operating system, but if the handler does not exist then the special files are worth for nothing, in fact opening any of the special files fails when the operating system lacks of the associated handler.



FIFOs are used as if they were normal files (open, read/write, close). For a normal Linux process to use them it is necessary that first a real time program creates the corresponding FIFO.

**From the point of view of a real time task**, the FIFOs are used through specific functions:

- `rt_create(unsigned int fifo, int size)`: creates a FIFO with a buffer of size `size`. From this moment on and until it is destroyed the device accessed from `/dev/rtf[fifo]` exists and can be used.
- `rt_destroy(unsigned int fifo)`: the corresponding FIFO is destroyed and its memory reallocated.
- `rt_fifo_put(fifo, char *buf, int count)`: attempts to write `count` bytes from the buffer `buf`. If there is no enough space in the FIFO's buffer it returns `-1`.
- `rt_fifo_get(fifo, char *buf, count)`: attempts to read `count` bytes from the FIFO, if not sufficient data is available it returns `-1`.

Lets see next an example of a system that makes use of these functions. This example is a small modification of one of the examples in the distribution of RT-Linux (sound):

#### **example3.c**

```
#define MODULE
#include <linux/module.h>
#include <linux/rt_sched.h>

#include <linux/rtf.h>
#include <asm/io.h>

RT_TASK task;

static int filter(int x){
    static int oldx;
    int ret;
    if (x & 0x80) {
        x = 382 - x;
    }
    ret = x > oldx;
    oldx = x;
    return ret;
}

void fun(int dummy) {
    char data;
    char temp;
    while (1) {
        if (rtf_get(0, &data, 1) > 0) {
            data = filter(data);
            temp = inb(0x61);
            temp &= 0xfd;
            temp |= (data & 1) << 1;
            outb(temp, 0x61);
        }
    }
}
```

```

    }
    rt_task_wait();
}
}

int init_module(void){
    rtf_create(0, 4000);

    /* enable counter 2 */
    outb_p(inb_p(0x61)|3, 0x61);

    /* to ensure that the output of the counter is 1 */
    outb_p(0xb0, 0x43);
    outb_p(3, 0x42);
    outb_p(00, 0x42);

    rt_task_init(&task, fun, 0, 3000, 1);
    rt_task_make_periodic(&task,
                          (RTIME)rt_get_time()+(RTIME)1000LL,
                          (RTIME)(RT_TICKS_PER_SEC / 8192LL));

    return 0;
}

void cleanup_module(void){
    rt_task_delete(&task);
    rtf_destroy(0);
}

```

As in the second example, we require the services of the module `rt_prio_sched`, but this time in order to use the FIFO's we have to load the module `rt_fifo_new` as well.

A periodic real time task of frequency 8192Hz is created. This task read bytes from the FIFO 0, and if it finds something it sends it to the speaker port of the PC. If we copy a sound file in ".au" format over `/dev/rtf0` we now will be able to listen it. It is not necessary to mention that the quality of the sound is terrible since the PC hardware only permits to use one bit to modulate the signal. The directory `testing/sound` of the distribution contains the file `linux.au` that can be used for tests.

To compile and execute it:

```

# gcc -I /usr/src/linux/include/linux -O2 -Wall
-D__KERNEL__ -D__RT__ -c example3.c
# modprobe rt_fifo_new
# modprobe rt_prio_sched
# insmod example3.o
# cat linux.au > /dev/rtf0

```

Notice how the `cat` tool can be used to write over any file, including special files.

We could also use the command `cp`.

To compare how the real time features affects the quality of the reproduction we only have to write a program that does the same operation but from a normal Linux user process:

#### **example4.c**

```
#include <unistd.h>
#include <asm/io.h>
#include <time.h>

static int filter(int x){
    static int oldx;
    int ret;
    if (x & 0x80)
        x = 382 - x;
    ret = x > oldx;
    oldx = x;
    return ret;
}
espera(int x){
    int v;
    for (v=0; v<x; v++);
}
void fun() {
    char data;
    char temp;

    while (1) {
        if (read(0, &data, 1) > 0) {
            data = filter(data);
            temp = inb(0x61);
            temp &= 0xfd;
            temp |= (data & 1) << 1;
            outb(temp,0x61);
        }
        espera(3000);
    }
}

int main(void){
    unsigned char dummy,x;
    ioperm(0x42, 0x3,1); ioperm(0x61, 0x1,1);

    dummy= inb(0x61);espera(10);
    outb(dummy|3, 0x61);

    outb(0xb0, 0x43);espera(10);

    outb(3, 0x42);espera(10);
    outb(00, 0x42);
```

```

    fun( ) ;
}

```

This program can be compiled like any normal program:

```
# gcc -O2 example4.c -o example4
```

And to execute it:

```
# cat linux.au | example4
```

To access the hardware ports of the computer from a normal Linux program we must request permission to the operating system. This is a basic and necessary protection measure to avoid program direct access to the hard disk, for example. The call `ioperm( )` tells the operating system that we wish to access a given range of input/output addresses. Only programs running with root privileges will receive such permission. Another detail worth noting is how the frequency of 8192Hz that modulates the sound is generated. Eventhough there is a system call named `nanodelay( )`, this only has a resolution of milliseconds, therefore we must make use of a temporal clock using a wait loop. The loop is adjusted so that it more or less works on a 100 MHz Pentium.

Now I suggest the reader to test the example4 together with the program ico, How do you hear it now? How does it feel like the real time version? So, Is real time worth for something?

## Conclusion

This second article concentrated on the programming details of real time tasks. The examples presented are very simple and lack of practical use, in the following article I will give a more useful application. **We will be able to control the TV from Linux or even more surprisingly to control your Linux box with the remote control!!.**

## References:

- *"Online KURT"* B. Srinivasan. <http://hegel.ittc.ukans.edu/projects/kurt>
- *"A Linux-based Real-Time Operating System"* by Michael Barabanov. Master of Science, New Mexico Institute of Mining and Technology.
- *"Linux as an Embedded Operating System"* by Jerry Epplin <http://www.espmag.com/97/fe39710.htm>
- *"Implementing Loadable Kernel Modules for Linux"* by Matt Welsh <http://www.ddj.com/ddj/1995/1995.05/welsh.html>
- *"Linux Kernel Internals"* by M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, D. Verworner. Ed: Addison-Wesley.
- *"Programación Linux 2.0, API de sistema y funcionamiento del núcleo"* by Rémy Card, Eric Dumas y Franck Mével. (Originally written in French and translated to Spanish). Ed: Eyrolles, Ediciones Gestión 2000.
- *"On Satisfying Timing Constrains in Hard-Real-Time Systems"* by J. Xu & L. Parmas. IEEE Trans. on

Web page maintained by Miguel A Sepulveda  
© Ismael Ripoll 1998  
**LinuxFocus 1998**