Implementation of a Signal Control System in a Real Time Environment

Vanessa Frías Martínez
Computer Science Department
Columbia University
110 Amsterdam Avenue, New York, NY 10027
e-mail: vf2001@cs.columbia.edu

Abstract – **Real time applications have time requirements to be accomplished. The missing of those deadlines make the system incur in fatal errors. Many Operative Systems(OS) have been upgraded to achieve preemption in order to offer applications the capability of meeting their deadlines. Linux, with its open source philosophy and robustness offers the opportunity of kernel code manipulation, as well as straight forward testing with the modules idea. This paper presents a real-time application implementation using a real time Linux kernel. The main goal is testing how well the real time OS does in comparison to a non-preemptive OS.**

1. INTRODUCTION

A Real Time (RT) system is one capable of guaranteeing timing requirements of the processes under its control. It must be fast and predictable. Fast meaning low latency, that is, 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 so as to be able to determine the task's completion time with certainty. A typical real time task will have timing constraints, resource requirements, communication requirements and concurrency constraints, all of which has to be treated.
Linux is a POSIX 1003 compliant OS. Processes can be locked into memory to prevent being paged to hard disk. But a Linux kernel does not provide the required event priorization and preemption functions needed by a real time process.
RTAI[2], Montavista[9] and RTLinux[9] deal with the implementation of real time Linux kernels. There are two main approaches to the problem:

•The preemption improvements approach, makes modifications in the original native Linux code so as to reduce the time spent by the kernel in non-preemptive sections of code. But, this approach only affords soft real time since when Linux interruptions are disabled by processes, no effective response is guaranteed. Montavista implements a fully preemptive kernel with this approach. Their Linux kernel is preemptive unless specified the contrary.

•The interrupt abstractions approach defines a two layer system where the standard Linux is run as a low priority process along with all the RT high priority preemptive processes, that are run in kernel space. The RT kernel will handle all the interruptions directly. Standard Linux interrruptions will be treated as soft. This approach implements hard real time resolutions. The original RTLinux and the RTAI offer currently this two layer system.

2. RELATED WORK

Linux distributions do not differ in the main kernel, which is common and unique for all of them, but in the applications they include, their graphical environments and other extras.
Open source implementations are being used by companies to offer complemented-payable-kernels coupled with a varied range of tools. The open source philosophy makes Linux world more active and rich in developments.
Some real-time open source Linux kernels, like RTLinux[13] or RTAI[2](Real Time Application Interface) works on the philosophy of two-layer systems and modules. Its main feature is that RT processes are considered as loadable modules. Those OS achieve response times of 15 µsecs compared to non real time resolutions ranging from 10 to 40 ms. This is obtained through running the RT processes in the kernel space. In order to avoid possible memory intrusions, both offer the possibility of user space execution. Both real time versions emulate standard Linux interrupts enable/disable so as to avoid the priority inversion problem between non-preemptive and preemptive RT processes.
Mategazza[7] designed for RTAI a more detailed and clear two layer system description known as HAL (Hardware Abstraction Layer). It supports five core loadable modules which provide the desired on-demand, real time capability. Those implement the scheduler, memory sharing, clocks control and FIFOs implementation. This will be our base system.
Andris[1] and Kupper[3] offer descriptions of real time modules ranging from the simple control of a port dealing with real time signals to the whole implementation of the control for a PUMA robotic arm, including identification of RT threads and RT Linux processes. Though, almost all the approaches up to now, have been developed for specific hardware and software problems.
Morgan K.[10] does a comparative study of the output of an audio process in a preemptive and non-preemptive OS, pinpointing the failures in the last one. But, is a local comparisson with bechmarks specifically designed for the problem.

3. RTAI ARCHITECTURE

RTAI implements a real time kernel where hardRT applications can meet their deadlines. Its design implies the modification of the current Linux kernel by adding some lines to different files of its code.
Linux is the RT task of less priority. Non RT services are to be executed by standard Linux. The communication between RT and Linux processes is made by FIFOs or shared memory. As for interruptions, the masking code has

been rewritten. Hard interruptions are redirected to the RT layer. All the Linux interruptions are considered soft and rerouted to the HAL layer. Standard Linux has no more control over hard interruptions.
RTAI is designed under the philosophy of modules. RTAI tasks are designed as dynamic extensible loadable modules. RTAI is the central module. We can install other modules on it, depending on the RT services we are interested in using. The rtai_sched module when we need to work with schedulers, rtai_fifos for FIFOs, rtai_shm for shared memory,, rtai_watchdog , and rtai_pthread for POSIX threads. Figure1 shows the basic architecture described here.

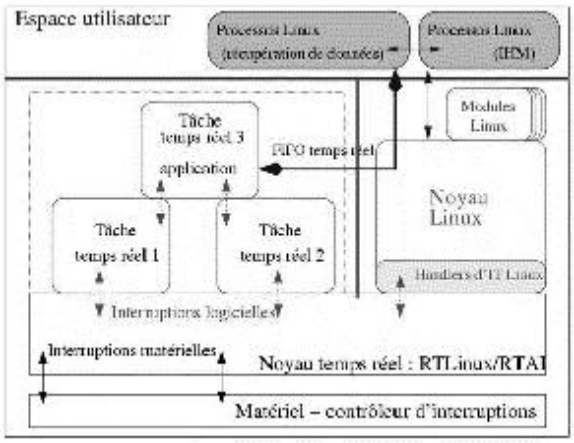

Fig.1. Basic Architecture of RTAI.

The scheduler module allows the user to define one shot or periodic tasks.They are controlled by means of the chip 8254.
The FIFO module allows us to create FIFOs at the user or kernel space. We can access to them by name. They are accessible at the device file system (/dev/rtfx,x=0...5).
Similarly, the shared memory module offers the possibility of working in user or kernel space. This is more difficult ot user, since two applications have to define a hand shaking protocol.
One of the last successes of the RTAI developers has been the LXRT services. This module, allows the use of all the services made available by RTAI and its schedulers in the user space, both soft and hard real time. All the manipulations can be done without root permissions. LXRT provides a family of real time scheduler services. Those services are the same function calls that the ones in the kernel space, with the difference that when called from user space they are executed by some "angels" taking care of each call. When defining an LXRT task, we need to fix a name, that will correspond to the LXRT system call. LXRT calls are implemented as soft interrupts(like the Linux ones) but with higher priorities. When an LXRT system call is called by a user space process, the LXRT handler will save everything onto the stack, change ds and es to *KERNEL*_DS and then execute its own angel.

4. THE SIGNAL CONTROL SYSTEM

4.1. Our proposal
Our real time signal control system will develop a set of loadable modules for the RTAI kernel[2]. The modules are designed to be a template that will be used in the study of resolutions of different RT Linux kernels. Those modules will take control of a real time input periodic signal from the parallel port and will implement a feedback control on it. The output will be generated to the serial port.
Our set of modules will allow the user to test the RT resolutions in different physical systems in order to do similar studies on its outputs. A set of benchmark programs is also designed to facilitate the test of the RT system in confront to the standard Linux.

4.2. RTAI as a basis
We have chosen the RTAI architecture to implement this system because of the advantages it offers. We have guaranteed real time behavior with resolutions of 15microsecs. It is also designed with fully modularity and scalability so as to be complemented and modified as desired. It's a compact system in its codification, the patch for the kernel adds only about 64K (100 lines) to the vanilla kernel. On the other side, we also take advantage of one of its lasts developments: the possibility of debugging our RT programs in the user space.

4.3. Design of the signal control system
The control system has been implemented in two different ways so as to offer a wide range of applications depending on our physical system. Our main interest is to define a set of modules and programs to process signals with microsecond resolutions. The real time resolutions offered by RTAI are in this range.
Our real time environment may allow us to be more or less intrusive. In this way, we propose two different approaches with similar time resolutions: a loadable module onto the kernel and a user-space executable. The first approach can reach times of 15 microseconds, whereas the second one, may have some 3 or 4 more microsecs of delay.

4.4 The Kernel Space Approach
The RTAI was originally designed in such a way that the real time programs had to be executed in the kernel space. Our first approach to the problem has been implemented in this way. Following the RTAI philosophy we have developed a set of full-loadable modules to be installed onto the patched RTAI kernel. These modules implement the control system previously described.
We have two different modules. The *parport_mod* installs on the system a function to read from the parallel port. The *serialport_mod* once installed will offer the user the possibility of using functions to write to the serial port. We have taken the needed parts of the code from rt_com program(using the LGPL license). This is a driver for the serial port implemented by Jens Michaelsen and Jochen Küpper. Finally, we have developped two kernel-user modules that will use the functions previously defined in order to test the RT application we want to. The

par_serial_user_mod will read from the parallel port and at each change write to the serial port. The *timer_serial_user_mod* allows the user to test our RT system without using any parallel port input. This module will periodically (using the timer chip 8254) write to the serial port.
Our system control approach will periodically output a signal to the serial port in real time resolutions of 15microsecs even when other I/O overhead linux applications are being executed.

4.5. The User Space Approach.

We have developed this approach by implementing a user-space executable RTAI file that uses the LXRT libraries. Those applications can also be executed in microseconds resolutions, implementing hardware real time control systems. This is our second approach to the system control. Executed in the user space, offers time resolutions of 19 microseconds without any dangerous intrusion into the kernel. The resolutions are in real time, which means that they are obtained even when executing I/O linux applications that charge the system.
Similarly to the kernel space design, our user program will periodically send an update signal to the serial port with the timing guarateed by the real time background we are working on. We also offer he possibility of controlling the serial output with the parallel input or with some periodic task controlled by the timer. In this way, we define two programs, user_space_par_ser and user_space_timer_ser. Both of them are executed in user space, but using the LXRT libraries means that the will have higher priority and better scheduler granularity scheduling than Linux tasks. Th Figure2 below, shows the LXRT environment executed by the angels.

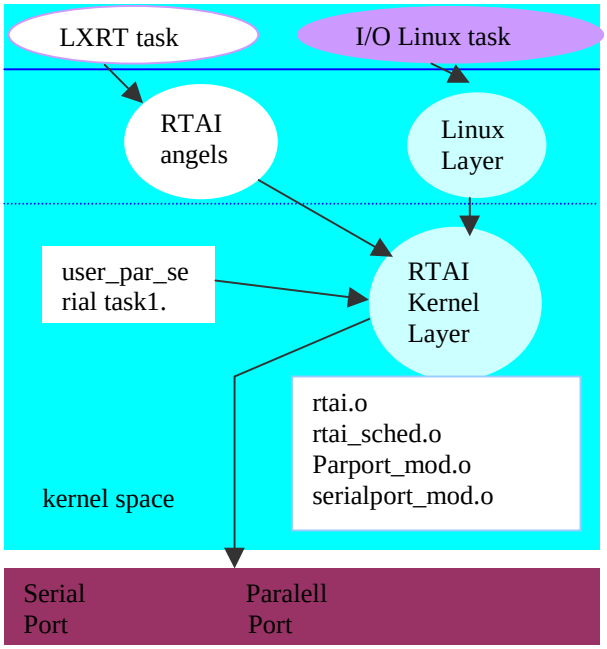

Fig.2. Modules, angels and tasks in RTAI.

5. ARCHITECTURE OF THE MODULES.

5.1. The Kernel space module.

The general algorithm for the process *par_serial_user_lxrt* will follow the following structure,

(parallel_port new data HANDLER)
then
read parallel_data
manipulate_data (parallel_data)
send data to serial port
endif

The real time modules are implemented with the basic init_module() and cleanup_module() directives. In the configuration of the parallel port we define our device with its IRQ number (#7), its io base port (0x378) and the handler we associate to it. Then, we will check and request for the io region,

```
dev.io_base   = PARPORT_BASE_ADDRESS;
dev.irq = PARPORT_IRQ_NUMBER;
dev.handler = parport_irq_handler;
request_region(dev.io_base,dev.port_size, "rtai_parport");
```

As for the serial port control, we use the code from the rt_com (RTLinux kernel module for communication across serial lines) designed by Jens Michaelsen and Jochen Küpper. We will also install that module into the kernel. The API of this module gives us some functions to configure the serial port and to write to it,

```
rt_com_set_param( 0, 0, 0 );
rt_com_setup(0,38400,RT_COM_PARITY_NONE, 1, 8 );
//write to the serial port.
n = rt_com_write( serial_port, data,size(data));
```

Our *par_serial_user* real time module, will load the main RTAI module onto the kernel with the rtai_mount() system call. By installing this module, we can then install all the others needed to use the new real time system calls provided by RTAI for schedulers, timers, general real time,ecc. Then, we associate a handler to the parallel port, so as to detect each interruption. We do this by associating a handler to the parallel port interruption,

```
rt_request_global_irq (dev.irq, dev.handler);
```

that will be fired when new data arrives, will be read and written to the serial port. The way the data is manipulated before being sent to the serial port, is specified by the function *manipulate_data()*, that can be modified by the user to do all the different testing experiments depending on its system.
As for the *timer_serial_user* module, the serial part is the same as explained above. The periodic part, was first designed with the call offered by RTAI, *rt_task_make_periodic()*. But since we want optimal execution times, the final decision was that of controlling directly the timer so as to associate to it a handler that will be executed each fixed *timer-period*. The body of this handler will communicate with the serial port.

Using the *rt_request_timer(serial_port,period,0),* we request a timer of period *period* and install the handler routine *serial_port* as a real time interrupt service routine for the timer based on the 8254 of our machine.

5.2. The User space application.

The application designed here, uses the system calls allowed by the LXRT module(executed by its angels).

The communication with the serial and parallel port has been implemented in two ways. The first approach, was that of designing an LXRT process that would directly manipulate the ports with the *inb* and *outb* directives. But we come up with a problem: to execute outb from the user space, we need to fix the permissions with ioperm() to allow the user space application access the I/O ports. We did so, but we found problems of accessibility to the port when compiling those applications. This problem was solved by compiling in an optimized way, but just with some compilers and some kernel versions.

Since we are looking for more general solutions, we decided to use higher system calls so as to compile easily the programs and adapt them to any Linux platform. LXRT modules are defined as normal kernel modules(with the install_module, clean_up module), plus an identification number that will recognize each of them in the lxrt system calls table. The way we have implemented it is,

```
int init_module(void)
{
if( set_rt_fun_ext_index(parport, MYIDX))
printk("Recompile your module with a different index\n");
...
```

The rt_fun_entry table associates an identifier to each of the functions in the module. Besides, each function needs to be defined related to its id, size for the stack, and arguments needed,

```
rtai_lxrt(MYIDX, SIZARG, RT_GET_PAR_DATA, &arg);
```

So, our user-space program, will let the user test any system with a parallel port input and a serial output.

The access to the parallel port has been done by implementing an LXRT module (parport) that is loaded into the kernel and executes the inb() directive itself. The serial port communication is implemented using the rt_com_lxrt API. This API is an adaptation of the rt_com device driver module for the LXRT space. The API calls for LXRT are similar to the kernel space ones.

So, the set of test programs designed for the user space, can be used by loading the parallel functions module that allow us to communicate with the parallel port, the serial functions module, rt_com_lxrt, adapted version of rt_com driver for lxrt, and executing our process. The process, will continuosly check for changes in the parallel port. When one is detected, it will communicate with the serial port. The main difference with the kernel space execution is that here the LXRT system calls are calls to 'angels' that will execute the kernel space real system calls. In order to use the same API, we need to fix the real time space of work with,

```
rt_make_hard_real_time();
```

For the timer_serial_user version, the parallel port is not defined. Our process will periodically communicate with the serial port. Our user space process, is defined as a periodic task that will be set with the directive *rt_task_make_periodic(task, rt_get_time() + 5*period, period).* Its periodical execution will send data to the serial port. That data may have been previously manipulated as desired by the function *manipulate_data().*

The final resolution times, taking into account that the LXRT module has to be executed, as well as the delays of the serial port manipulation will be of about 19 to 20 microsecs.

6. NON RT MODULES.

As a complement to our system control modules, we have also developed a set of non real time modules doing the same functionality, in order to allow the user compare the output of their systems in a real and non real time environment.

The *par_serial_linux_mod* is very similar to the *par_serial_mod*. When installed, this module will capture the parallel interrupt and send it to our own interruption handler. This one, will read the info from the I/O parallel port and will manipulate the data and send it to the serial port.

```
void init_module ( ) {

serial_device =  open ("/dev/ttyS0", O_RDWR );
request_irq ( PAR_IRQ, parallel_irq_handler, SA_SHIRQ,
"parallel_handler", NULL);

}
void my_irq_handler ( ) {

   par_info = inb (0x378);
  manip_info = manipulate_data ( par_info );
  n = write(serial_device, manip_info, size(manip_info));
  if (n < 0)
    fputs("write()  failed!\n", stderr);
}
```

As for the *timer_serial_module*, instead of using the inefficient *crontab file*, we have developed a module that once loaded, will put our task in the tq_timer task queue so as to be executed at each timer interrrupt.

```
void init_module ( ) {

queue_task (&Task, &tq_timer);
}
```

where *Task* has in its struct the call to its interruption handler, which will again manipulate data and send it to the serial port in a similar way than the explained above.

In this way, a user-space program has been implemented in non real time so as to be executed periodically or at each parallel port interrupt and output a signal to the serial port. Finally, we have also implemented an application to

overhead the system with I/O operations that will generate continuous interruptions from the standard linux user space. In order to make it optimal, the outb and inb have been used, as well as the ioperm() to give user permissions for the ports.

7. INSTALLATION OF THE SYSTEM.

The installation has been done on a 266MHz K6 processor. This procedure can be a big deal. We have chosen the RTAI real time application interface to patch our kernel so as to make it real time by loading different modules.
Theoretically, any distribution can be used, though, we have found problems working with Red Hat versions 7.0 and 7.2. Finally, we have installed a Mandrake 8.0. It is highly important to know that the kernel to be patched, can't be the one coming with the distribution. We need a vanilla kernel. Our linux kernel is a vanilla 2.4.8 kernel obtained free source from www.kernel.org.
Once we had the kernel and the patch, the first thing to do is patch the kernel(with patch < name_patch). The patch applied is the last version of the RTAI patches, the 2.1.9 of Sept, 1st. After applying it, we can follow the normal Linux installation procedure (make config, make install and make modules_install). When configuring the system, it is important to activate the loadable modules option, as well as deactivating the high power option.
Another important feature arises in the compilation of the kernel. When compiling, we have to take into account that our patched kernel won't compile with the gcc-2.96 compiler, because of the distribution we are using. We managed though to compile it either with kgcc and gcc-2.95 compilers.
It is interesting to browse through our kernel files and see how small are the modifications done(about 70 lines, 64k in irq.c, entry.S, smp.c, time.c io_apic.c and 386_ksyms.c). Those modifications, constitute the HAL. Finally, we need to compile the code for the loadable modules that will implement the real time features of our system. Again, the compilation has to be done with one of the defined compiler above. Once compiled, we will have a set of loadable modules defining the RTAI API for real time: rtai.o itself, rtai_mem.o, rtai_shm.o, rtai_fifo.o, rtai_sched.o and rtai_rpc.o.
Our kernel space module is designed so as to be executed in any RTAI platform where all the real time system calls can be recognized. It won't work on a i486 since is programmed to manipulate directly the 8254, not present in this architecture.
Our user space application will work in any rtai-lxrt-linux system.
The nonRT applications are to be executed as standard linux programs in the user space. Only compilation features have to be taken into account.
The compilation of the code has to be done with the same compiler used for the kernel to avoid any error. It is important not to forget the –O optimization flag for the compilation when necessary.

8. APPLICATIONS OF THE SYSTEM.

Our system, offers a set of real time modules and routines so as to test the real time resolutions of any physical system manipulating the serial port, either periodically or by some signal from the parallel port. We also offer the same routines for a non real time system so as to make comparative studies of the resolution times of the system in both real time and non real time cases.
Depending on the physical system we want to test, the user can select a more or less intrusive installation. The kernel space module version, will install some modules into the kernel to control a signal coming from the parallel port and sending some other signal to the serial port. We can guarantee time responses of maximum 15 microseconds. The user space application, will do the same task, but without interfering directly into the kernel. In this case, delays up to 20 microseconds.
A benchmark is also offered to the user: a non RT application can be executed so as to see the delays of a non real time system when reading from the parallel port and writing onto the serial port.
Finally, an I/O program in standard linux user space can be executed to overhead the system and calculate the efficiency of the RT modules and non-real time programs, in extreme system overhead situations.

9. CONCLUSIONS AND FUTURE WORK.

As a future work we propose the development of a real hardware application so as to test the modules designed and to physically see the comparisson between the RT and nonRT outputs of the system.
We propose the design of an eight leds device that has to do 3D designs in real time. The leds are distributed longitudinally in a stick that is attached to the engine always turning around itself.
The system will have an engine, whose input to the computer will be by the parallel port. The engine will communicate to the computer at each turning of 90 degrees. Each tick will go to the computer and will have to be processed, real time, so as to switch on the appropriate led in the stick to design the feature in 3D.
The proposal is that of using our real time modules to receive the periodic signal from the parallel port and manipulate the serial port as desired. This system is supposed to work properly, even if executing our overhead I/O program. This is thanks to the real time resolutions guaranteed by our RTAI system.
Finally, we would also execute it with the non real time modules so as to calculate resolutions and see how the 3D feature in the leds is obtained in a non real time system with the I/O overheading the system.
Besides, many other systems that need to manipulate parallel or serial ports in real time, or wanting to test whether their systems need or not real time resolution, can use our set of modules to do so.

10. REFERENCES

[1] Andris P. Robot Control Using Real-Time Linux Artiticulated body Movement Simulation, Proc. of the Real-Time LINUX Workshop, Orlando, USA, 2000 .

[2] Dipartimento di Ingegneria Aerospaziale, Politecnico di Milano, RTAI, www.rtai.org

[3] Kupper, J. The serial port driver of real time Linux. Institut fur Physicalysche Chemic, Frankfurt, 2000.

[4] Küpper J. Real Time Linux driver for the serial port rt-com.sourceforge.com/rt_com.pdf, (on line manual).

[5] Lineo, www.lineo.com

[6] List, S. RTAI: définition et concepts, Alcôve. France, 2000, www.courseforge.org/courses/fr/rtai1/rtai.pdf (on line manual).

[7] Mantegazza P.DIAPM-RTAI:why's, what's and how's. Proceedings of the Real Time Linux Workshop. Vienna, Austria, 1999.

[8] Mantegazza P., Dozio L., A hard Real Time support for LINUX, Dipartimento di Ingegneria Aerospaziale, Milano, Italy,1999. www.rtai.org (on line manual)

[9] Montavista, www.mvista.com

[10] Morgan, K. in Linux Devices, www.linuxdevices.org

[11] Real Time Linux website, www.realtimelinux.org

[12] Robbins K.A, Practical UNIX Programming: a guide to concurrency , communication and multithreading, Prentice Hall, 1996.

[13] RTLinux, www.rtlinux.org

[14] Ward B., The Linux Kernel HowTo (http://www.linuxdoc.org/HOWTO/Kernel-HOWTO.html)

[15] Waugh T., The Linux 2.4 parallel port subsystem, people.redhat.com/twaugh/parport/html/parportguide.html (on line manual).