RAPPORT DE PROJET DE FIN D'ETUDE

Développement d'une application en temps-réel pour le contrôle d'une carte CANPCI

Auteur: **David Lozano Garcia**

Chef de projet: **Patrice Kadionik**

ENSEIRB, Bordeaux

Project de 15 Octobre 2000 au 13 Avril 2001

INDEX

1 - Introduction

The objective is the development of a real time-based data acquisition system. The implementation of this system is based in two main elements:

- The real time element: RTLinux is to be used.
- The data acquisition element: the CANPCI board, developed here in the ENSEIRB.

With this goal in mind, RTLinux is to be used in order to develop an RT application for controlling the CANPCI board. We would program using C language under Linux environment, using RTLinux features.

Real-Time Linux (RTLinux) is a version of Linux that provides hard real time capability. A NASA computer running RTLinux flew into the eye of Hurricane Georges to collect data[14]; the Jim Henson Creature Shop in Hollywood is developing a RTLinux application to control ``animatronic'' things used in movies; RTLinux has been used for video editors, PBXs, robot controllers, machine tools, and even to stop and start hearts in medical experiments1

RTLinux provides the capability of running special realtime tasks and interrupt handlers on the same machine as standard Linux. These tasks and handlers execute when they need to execute no matter what Linux is doing. The worst case time between the moment a hardware interrupt is detected by the processor and the moment an interrupt handler starts to execute is under 15 microseconds on RTLinux running on a generic x86. A RTLinux periodic task runs within 35 microseconds of its scheduled time on the same hardware. These times are hardware limited, and as hardware improves RTLinux will also improve. Standard Linux takes up to 600 microseconds to start a handler and can easily be more than 20 milliseconds (20,000 microseconds) late for a periodic task2.

With these properties, it will be implemented a data acquisition system with hard real-time constraints, improving others implemented as standard non-real-time OS-based data acquisition system

2 - Real Time

There are many interpretations of the meaning of real time. One simple definition is: A real-time operating system is able to execute all of its tasks without violating specified timing constraints.

2.1 – Characteristics and responsibilities of an RTOS

We are going to describe here some of the basic characteristics and responsibilities of an *Real Time Operating System* (RTOS).

The job of deciding at which instances in time tasks will execute is called *scheduling*. It is one of the major responsibilities of an operating system, be it a general purpose operating system or an RTOS. But it is exactly in their scheduling that general purpose and real-time operating system differ most. They use the same basic principles, but apply them differently because they have to satisfy different performance criterions.

Distinction between “soft real time” and “hard real time” is often made. “Soft” indicates that not meeting the specified timing constraints is not a disaster, while it is a disaster for a hard real-time system. For example: playing an audio or video file is soft real time, because few people will notice when a sample comes a fraction of a second too late. Steering a space shuttle, on the other hand, requires hard real time, because the rocket moves with a velocity of several kilometers per second such that small delays in the steering signals add up to significant disturbances in the orbit which can cause erroneous atmosphere entry situations.

Practically speaking, the distinction between soft and hard real time is often (implicitly and mistakenly) related to the time scales involved in the system: soft real-time tasks must typically be scheduled with milli-seconds accuracy; hard real-time tasks with micro-seconds accuracy. But this implicit assumption has many exceptions.

Roughly speaking, one has several classes of time constraints:

Deadline: a task has to be completed before a given instant in time, but when exactly the task is performed during the time interval between now and the deadline is not important for

the quality of the final result. For example: the processor must fill the buffer of a sound card before that buffer empties; the voltage on an output port must reach a given level before another peripheral device comes and reads that value.

Zero execution time: the task must be performed in a time period that is zero in the ideal case. For example: digital control theory assumes that taking a measurement, caculating the control action, and sending it out to a peripheral device all take place instantaneously.

Quality of Service (QoS): the task must get a fixed amount of "service" per time unit. ("Service" most often means "CPU time", but could also mean "memory pages", "network bandwidth" or "disk access bandwidth".) This is important for applications such as multimedia (in order to read or write streaming audio or video data to the multimedia devices), or network servers (both in order to guarantee a minimum service as in order to avoid "denial of service" attacks). The QoS is often specified by means of a small number of parameters: "s" seconds of service in each time frame of "t" seconds.

The *latency* (or tardiness) of a task is the difference between the time the task should have started (or finished) and the time it actually did. This latency is due to several factors:
(i) the timing properties of processor, bus, memory and peripheral devices,
(ii) the scheduling properties of the RTOS, and
(iii) the load on the system (i.e., the number of tasks that want to be scheduled concurrently).
These factors are seldom constant over time; the statistical distribution of the latencies in the subsequent schedulings of tasks is called the scheduler's jitter.

An important contribution to the latency is the context switch time of the processor; i.e., the time needed to save the local data of the currently running task (e.g., the contents of the processor registers; the stack of the task; the current instruction pointer), and to replace it with the local data of the newly scheduled task.

An RTOS must not only be able to schedule tasks based on the ticks of a clock chip, but most often it also has to service peripheral hardware, such as motors, sensors, communication signals, and talk to other systems. The synchronization with this outside world is performed through *interrupts*: each peripheral device can signal the processor(s) on which the RTOS runs that it needs servicing. That signal interrupts the current task on the processor, and the RTOS must schedule the corresponding Interrupt Service Routine (ISR) as fast as possible. Therefore, it will have to pre-empt the currently running task.

A third responsibility of an RTOS (besides scheduling and interrupt servicing) is commonly known under the name of *Inter-Process Communication* (IPC). ("Process" is just another name for "task".) IPC means that when tasks need to exchange information with other tasks, the RTOS has to make sure that this communication takes place in a timely and reliable way. The simplest form of communication are signals: one task sends a message to another task, in order to synchronize their actions.

A fourth responsibility of the RTOS is *memory management*: the different tasks in the system all require part of the available memory, often to be placed on specified hardware addresses (for memory-mapped IO). The job of the RTOS then is (i) to give each task the memory it needs (this is memory allocation), and (ii) to take the "appropriate" action when a task uses memory that it has not allocated. (Common causes are: unconnected pointers and array indexing beyond the bounds of the array.) This is the so-called memory protection feature of the OS. Of course, what exactly the "appropriate action" should be depends on the application, often it boils down to the simplest solution: killing the task and notifying the user.

Finally, *task* (or "*process*") *management* is also the job of the operating system: tasks must be created and deleted on line; they can change their priority levels, their timing constraints, their memory needs; etc. Task management for an RTOS is a bit more dangerous than for a general purpose OS: if a real-time task is created, it has to get the memory it needs without delay, and that memory has to be locked in main memory in order to avoid access latencies; changing run-time priorities influences the run-time behaviour of the whole system and hence also the predictability which is so important for an RTOS. So, dynamic process management is a potential headache for an RTOS.

To summarize, the basic responsibilities of an RTOS are:

(i) scheduling,

(ii) interrupt servicing,

(iii) inter-process communication and synchronization,

(iv) memory management, and

(v) task management.

2.2 - Linux limitations

As the system is under Linux, because it is free, powerful, and reliable, we are going to show why Linux is not an RTOS, and the necessity of RTLinux to implement the real-time acquisition system under Linux.

No other general purpose OS does any better than standard Linux in mixing RT with standard services, all for the same reasons. The most obvious problem is that the most useful design rule for general purpose operating systems is: *optimize the common case*. Thus, Linux SMP locks are exceptionally fast when there is no contention (the common case) and pay a significant price when there is contention. To use another design would slow down general operation of the system in order to optimize something that should happen only rarely. Similarly, standard Linux interrupt handling is really fast for a general purpose operating system. In some measurements, standard Linux averages 2 microseconds to get to the interrupt handler on reasonably standard x86 PCs. That's impressive and it is critical for some applications. Worst case behavior is not so impressive and some interrupts are delayed by hundreds of microseconds. The problem is that it is not so easy to figure out how to decrease worst case without increasing average case.

In the Linux code can be seen many more fundamental contradictions with realtime requirements. A few examples should make the case.

- ``Coarse grained'' synchronization means that there are long intervals when one task has exclusive use of some data. This will delay the execution of a realtime task that needs the same data structures. ``Fine grained'' synchronization, on the other hand, will cause the system to spend a lot of time uselessly locking and unlocking data structures, slowing down all system tasks. Linux uses coarse grained scheduling around some of its core data structures because it would not be good to slow down the whole system to reduce worst case.

- Linux will sometimes give even the most unimportant and nicest task a time-slice, even when a more important task is runnable. It would not be smart to never run a background process that cleans up log files, even if a higher priority computation is willing to use up all available processor time. But in a realtime system, the highest priority task should not wait for

a lower priority task. In a realtime system, it cannot be assumed that low priority tasks will ever make progress, but in a general purpose operating system we do assume that low priority tasks will progress.

- Linux will reorder requests from tasks to make more efficient use of the hardware. For example, disk block reads from the lowest priority processes may take precedence over read requests from the highest priority process so to minimize disk head movement or to improve chances of error recovery.

- Linux will ``batch" operations to make more efficient use of the hardware. For example, instead of freeing one page at a time when memory gets tight, Linux will run through the list of pages clearing out as many as possible, delaying execution of all processes. It would be counter-productive for Linux to run the swapper each time a page was needed, but the worst case certainly gets a lot worse.

- Linux processes are heavyweight processes that are associated with significant overhead from process switching. Although Linux is relatively fast in switching processes, it can take several hundred microseconds on a fast machine.

- Linux follows the standard UNIX technique of making kernel processes non-preemptive. That is, when a process is making a system call (and running in kernel mode) it cannot be forced to give up the processor to another task, no matter how high the priority of the other task. To write operating systems, this is wonderful because it makes a lot of very complicated synchronization problems disappear. To run real-time programs, however, it is not so wonderful that important processes cannot get scheduled while the kernel works on behalf of even the least important process. In kernel mode, it cannot be rescheduled.

- Linux will make high priority tasks wait for low priority tasks to release resources. This is a great problem in real-time programs where the high priority task must be finished before all the others tasks.

- Linux disables interrupts in critical sections of kernel code. This means a real-time interrupt may be delayed until the current process, no matter how low priority, finishes its critical section.

Other reasons why Linux is a poor RTOS are the unpredictable delays caused by:

Accessing a network. Especially with the TCP/IP protocol, that re-sends packets in case of transmission errors.

Low-resolution timing. The smallest time slice used in most general purpose operating system is around 1 millisecond; Linux uses a basis scheduling time slice of 10 milliseconds on most processors (1 milliseconds on Alpha). Moreover, programming the timer chip often generates unpredictable delays too.

Low-resolution timer data structure. The clock of an RTOS must run at a higher frequency that that of a general purpose operating system. Hence, the data structure in which the time is kept should often be adapted to this higher rate, in order to avoid overflow.

Non-real-time device drivers. Device drivers are often sloppy about their time budget: they use busy waiting instead of timer interrupts, or lock resources longer than strictly necessary.

Memory allocation and management. After a task has asked for more memory (e.g., through a `malloc` function call), the time that the memory allocation task needs to fulfill the request is unpredictable, especially when the allocated memory has become strongly fragmented and no contiguous block of memory can be allocated. Moreover, a general purpose operating system swaps code and data out of the physical memory when the total memory requirements of all tasks is larger than the available physical memory.

`proc` *file system*. This is the user interface to what is going on inside the Linux kernel: all this information is offered to user tasks in the form of “files” in this (virtual) file system. However, accessing information in this file system implies significant overhead in some cases, because the files are virtual: they are “created” only when needed.

The exact magnitude of all the above-mentioned time delays changes very strongly between different hardware. Hence, it is not just the operating system software that makes the difference. For some applications, the context switch time is most important (e.g., for sampling audio signals at 44kHz), while other applications require high computational

performance, at lower scheduling frequencies (e.g., robot motion control at 1kHz). But again, some tasks, such as speech processing, require both.

So it is clear that Linux is not a valid OS for implementing the real time component of the system. And redesign of the scheduling algorithm will not help because there are unpredictable delays caused by the kernel preemption problem, the virtual memory paging system, and the demands of interrupt driven devices. But proprietary real-time operating systems are too expensive and too rigid to serve this purpose. On the other hand, a from-scratch operating system will lack the graphical displays, network interfaces, and developments tools needed for any but the smallest project. That is why we will use RTLinux, as it will be seen in the next chapter.

3 - RTLinux

RTLinux began life as a research project at New Mexico Tech in Socorro New Mexico. RTLinux main developers now are Yodaiken, Barabanov and Cort Dougan, although many other people have contributed code or testing.

RTLinux can be freely obtained at www.fsmlabs.com or www.rtlinux.com.

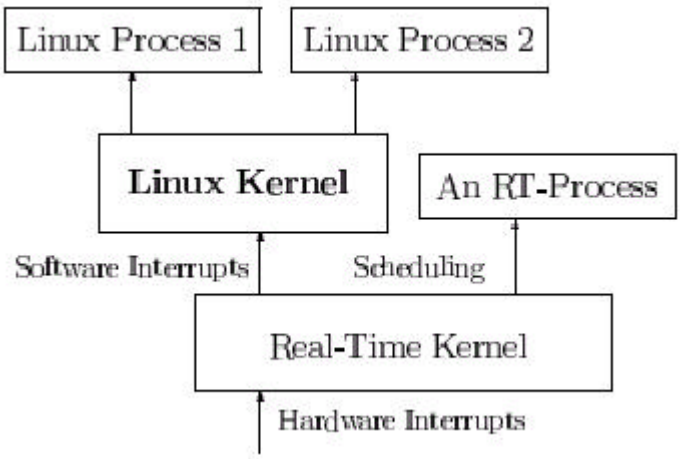
3.1 - Description

A brief description of RTLinux could be:

RTLinux is a small, deterministic, real-time operating system that is somewhat like a single POSIX process sitting on a bare machine. Hard realtime applications are threads and signal handlers in this process. Linux runs as the lowest priority thread of the RTLinux kernel and it is made always pre-emptible. The real-time kernel is itself non-preemptable, but since its routines are very small and fast, this does not cause big delays. One of the key design principles of RTLinux is that the more that is done in Linux (and the less that needs to be done on the RT side) the better. The RTLinux programming model is that anything that has strict timing requirements should be written as a thread or signal handler (interrupt handler) and whatever does not need hard realtime should go into Linux. This allows us to keep the RT side small, deterministic and as fast as the hardware will permit, while still drawing on Linux for sophisticated services and applications. So, for example, device initialization can be left to Linux. There cannot be any realtime constraints at boot time, so there is no need for the RT system to be involved; and blocking dynamic resource allocation is left to Linux; Any thread of execution that is willing to be blocked when there are no available resources cannot have hard realtime constraints.

Having this in mind, now we will see more deeply what is an how RTLinux works. What makes RTLinux useful is that it extends the standard UNIX programming environment to realtime problems. RTLinux realtime interrupt handlers and tasks can be connected to ordinary Linux processes, either via a device interface where Linux processes read/write data, or via shared memory. A standard Linux process can collect data from a realtime handler or task, process and log it and display the results.

RT-Linux is an operating system in which a small real-time kernel coexists with the POSIX-like Linux kernel. The intention is to make use of the sophisticated services and highly optimized average case behaviour of a standard time-shared computer system while still permitting real-time functions to operate in a predictable and low-latency environment. At one time, real-time operating systems were primitive, simple executives that did little more than offer a library of routines. But real-time applications now routinely require access to TCP/IP, graphical display and windowing systems, file and data base systems, and other services that are neither primitive nor simple. One solution is to add these non-real-time services to the basic real-time kernel. A second solution is to modify a standard kernel to make it completely pre-emptable. Both of these solutions have been taken in differents systems. RT-Linux is based on a third path in which a simple real-time executive runs a non-real-time kernel as its lowest priority task, using a virtual machine layer to make the standard kernel fully pre-emptable. When there is real-time work to be done, the RT operating system runs one of its tasks. When there is no real-time work to be done, the real-time kernel schedules Linux to run. So Linux is the lowest priority task of the RT-kernel. The following figure shows it:



The design philosophy behind RT-Linux was to minimize the changes made to Linux itself, providing only the essentials necessary for implementing real-time applications. Minimizing the disruption to Linux makes it easy to port RT-Linux to new versions of Linux as they appear. As a result, RT-Linux relies on Linux itself to provide almost all services needed, with RT-Linux providing only low-level task creation, installation of interrupt service

routines, and queues for communication among the low-level tasks, ISRs, and Linux processes.

One result of this design is that an RT-Linux application can be thought of as having two domains-the real-time and the non-real-time. Functions placed in the real-time domain can meet their real-time requirements, but they must be simple because the resources available to them are quite limited. On the other hand, non-real-time functionality has the full range of Linux resources available for use, but cannot have any real-time requirements. Facilities for communication between the two domains are provided. But before using RT-Linux, a system designer has to be sure that all of the needed functionality could fit into one of the two domains. Using RT-Linux does not magically make pre-existing Linux functionality real-time. Let us suppose, for example, that a designer has a Linux driver for a serial port, and wants to toggle a parallel output line (using a real-time task) within some fixed time after receiving a byte sequence on the serial port. The Linux serial driver can't be used because it resides in the non-real-time domain, and it can't be predicted when the serial driver would awaken the real-time task driving the parallel output line to perform its work. Thus, both the serial and the parallel ports would have to be done in the real-time domain, which would require redesigning the serial driver.

RTLinux is designed so that the RT kernel never has to wait for the Linux side to release any resources. The RT kernel does not request memory, share spin-locks, or synchronize onany data structures, except in tightly controlled situations. For example, the communication links used to move data between RT tasks and Linux processes are non-blocking on the RT side: there is never a case where the RT task waits to queue or dequeue data.

We can say that RTLinux decouples the mechanisms of the realtime kernel from the mechanisms of the general purpose kernel so that each can be optimized independently and so that the RT kernel can be kept small and simple.

3.2 – Interrupt handling

It has been said before that RTLinux works by treating the Linux OS kernel as a task executing under a small realtime operating system. In fact, Linux is the idle task for the realtime operating system, executing only when there are no realtime tasks to run. The Linux task can never block interrupts or prevent itself from being preempted. The technical key to all this is a software emulation of interrupt control hardware, to minimize changes in the Linux kernel and to achieve that in RT-Linux, all interrupts were initially handled by the Real-Time kernel and were passed to the Linux task only when there were no real-time tasks to run.

In RTLinux; the interrupts are divided into two groups: those under the control of Linux, and those controlled by RT-Linux. RT-Linux interrupts are restricted in what they can do; they cannot make Linux calls. So there is no reason that they can't interrupt the Linux kernel. After all, they can't interfere with anything in the kernel if they don't change anything in it. On the other hand, Linux interrupts can't be allowed to interrupt the kernel. So RT-Linux implements a virtual interrupt scheme in which Linux itself is never allowed to disable interrupts. When Linux tells the hardware to disable interrupts, the realtime system intercepts the request, records it, and returns to Linux. No matter what state Linux is in, it cannot add latency to the realtime system interrupt response time.

Linux uses "`cli`" and "`sti`" macros to implement disabling and enabling interrupts. In standard Linux, these macros simply execute the corresponding x86 instructions. RT-Linux modifies these macros so that instead of disabling interrupts when a "`cli`" is executed, it simply reroutes the interrupt to some RT-Linux code. If the interrupt is an RT-Linux interrupt, it is allowed to continue, invoking the realtime handler which corresponds with the interrupt. If it is a Linux interrupt, a flag is set to mark that the interrupt is pending. When a "`sti`" is executed, any pending Linux interrupts are executed, by invoking the corresponding Linux handlers. In this way, Linux still can't interrupt itself, but RT-Linux can interrupt it.

A more technical explanation of the interrupt handling in RTLinux is as follows. Modifications to the Linux kernel are primarily in three places:

- The `cli` routine to *disable* interrupts is modified to simply clear a global variable controlling soft interrupt enable.
- The `sti` routine to *enable* interrupts is modified to generate emulated interrupts for any pending soft interrupts.
- The low-level "wrapper" routines which save and restore state around calls to handlers have been changed to use soft return from interrupt code instead of using the machine instruction.

When an interrupt occurs, control switches to a real-time handler. The handler does whatever needs to be done in the real-time executive and then may pass the interrupt on to Linux. If the soft interrupt enable flag is set, then the stack is adjusted to fit the needs of the Linux handler and control is passed, via a soft interrupt table, to the appropriate Linux "wrapper". The "wrapper" saves additional state and calls the Linux handler (a program usually written in C). When the handler returns control to the "wrapper" a soft return from interrupt is executed. Soft return from interrupt restores state and then checks to see if any other soft interrupts are pending If not, a hard return from interrupt is executed. If there are interrupts pending, then the highest priority one is processed.

Linux is reasonably easy to modify because, for the most part, the kernel code controls interrupt hardware through the routines `cli()` and `sti()`. In standard x86 Linux, these routines are actually assembly language macros that generate the x86 `cli` (clear interrupt bit) and `sti` (set interrupt bit) instructions for changing the processor control word. Because interrupts can be disabled and enabled individually in the interrupt controller, and because some Linux drivers directly access the interrupt controllers and the hardware timer, it also had to be modified some driver code. The code for the three macros is:

```
S_CLI:   movl $0, SFIF

S_IRET:  push %ds
         pushl %eax
         pushl %edx
         movl $KERNEL_DS, %edx
         mov %dx, %ds
         cli
```

```
        movl SFREQ, %edx
        andl SFMASK, %edx
        bsrl %edx, %eax
        jz not_found
        movl $0, SFIF
        sti
        jmp SFIDT (,%eax,4)
not_found:
        movl $1, SFIF
        sti
        popl %edx
        popl %eax
        pop %ds
        iret

S_STI:  pushfl
        pushl $KERNEL_CS
        pushl $done_STI
        S_IRET
done_STI:
```

Interrupt handlers in the RT-executive perform whatever function is necessary for the RT system and then may pass interrupts on to Linux. Since the real-time system is not involved in most I/O, most of the RT device interrupt handlers simply notify Linux. On the other hand, the timer interrupt increments timer variables, determines whether a RT task needs to run, and passes interrupts to Linux only at appropriate intervals.

If software interrupts are disabled (`SFIF` = = 0), control simply returns through `iret`. Otherwise, control is passed to `S_IRET`. This macro invokes the software handler corresponding to the interrupt that has the highest priority among pending and not masked ones.

The `S_IRET` code begins by saving minimal state and making sure that the kernel data address space is accessible. In the critical section surrounded by the actual `cli` and `sti` the software interrupt mask is applied to the variable containing pending interrupts, and then it looks for the highest-priority pending interrupt. If there are no software interrupts to be

processed, software interrupts are re-enabled, the registers are restored, and it returns from the interrupt. If an interrupt to process is found, control passes to its Linux "wrapper".

Each Linux "wrapper" has been modified to fix the stack so that it looks as if control has been passed directly from the hardware interrupt. This step is essential because Linux actually looks in the stack to see if the system was in user or kernel mode when the interrupt occurred. If Linux believes that the interrupt occurred in kernel mode, it will not call its own scheduler. The body of the wrapper has not been modified, but instead of terminating with an `iret` operation, the modified wrapper invokes `S_IRET`. Thus, wrappers essentially invoke each other until there are no pending interrupts left.

On re-enabling software interrupts, all pending ones, of course, should be processed. The code simulates a hardware interrupt. The flags and the return address are pushed onto the stack and `S_IRET` is used, as it can be seen in the code. Individual disabling/enabling of interrupts is handled similarly.

So we have seen that, no matter what Linux does, whether Linux is running in kernel mode or running a use process, whether Linux is disabling interrupts or enabling interrupts, whether Linux is in the middle of a spin-lock or not, the realtime system is always able to respond to the interrupt.

3.3 – Real-Time Tasks

Real-time tasks are user-defined programs that execute according to a specified schedule under the control of the kernel.

The initial design was to give each real-time task his own address spaces to provide memory protection. This was done by utilizing Intel 80x86 processors' built-in paging mechanism. On each context switch the page directory base register was changed to point to the page directory of the new task. For security reasons, tasks were executed with the lowest priority level. Light-weight system calls were used for communication between the real-time kernel and processes. In this way, it works, but the system performance is not optimal. One reason for performance problems is that caches in 486 CPUs are virtual. Whenever the page

directory base register is changed, the translation lookaside buffer (TLB) has to be invalidated. Since real-time context switches are frequent, TLB invalidations inflict a severe performance decrease.

Another source of overhead is in system calls. Protection level changes on i486 CPUs are expensive. A trap to a more privileged level, for example, takes as long as 71 cycles to execute, while most other instructions take less than 10 cycles

One way to improve performance is to run all RT-tasks in one address space. By using the kernel space address, we also eliminate the overhead of protection level changes. Linux has a useful feature in this regard: loadable kernel module. Kernel modules are object files that can be dynamically loaded into the kernel address space and linked with the kernel code. Each module defines two routines: `init_module()` and `cleanup_module()`. The former is called when the module is loaded in the kernel, the latter when the module is removed. This provides an easy means to manipulate available drivers and filesystems in Linux.

Loadable kernel modules are used in the current version of Real-Time Linux to dynamically create real-time tasks. This approach is more fragile: a bug in a real-time task can wipe out the whole system. The use of the C Language aggravates this problem. Equivalence of arrays and pointers, type casts make it all too easy to write programs with memory referencing bugs. On the other hand, since real-time tasks often control expensive peripheral devices, it is reasonable to use the same level of caution as when programming an OS kernel.

Running tasks in the kernel address space has several advantages. Besides eliminating frequent TLB invalidation and protection level changes mentioned above, the approach allows us refer to functions and objects by names rather than descriptors. For example, real-time tasks are represented as C structs. Each task can be given an arbitrary C identifier that can be used in other tasks. Dynamic linking performed during module loading resolves symbols to addresses, so the access is very efficient.

Task switching is also easier if all tasks run in one address space. Real-Time Linux performs task-switching in software because hardware switches are slow on i486 CPUs. A context switch consists of pushing all integer registers on the stack and changing the stack pointer to point to the new task. Tasks with floating point context are also supported.

Real-time tasks run at the kernel privilege level in order to provide direct access to the computer hardware. All task resources are statically defined. In particular there is no support for dynamic memory allocation. They have fixed allocations of memory for code and data, because otherwise we would have to allow for unpredictable delays when a task requested new memory or paged in a code page. The basic approach here is that any sophisticated services that require dynamic memory allocation should be moved into Linux processes. Real-time tasks cannot use Linux system calls or directly call routines or access ordinary data structures in the Linux kernel, because this would introduce possibilities of inconsistencies.

3.4 - Scheduling

The main task of a real-time scheduler is to satisfy timing requirements of tasks. There are many ways to express timing constraints and many scheduling policies. No single policy is appropriate for all applications.

In most real-time systems, the scheduler is a large, complex piece of code that can not be extended in any way. The user can only modify the behaviour of the scheduler by adjusting parameters, which may not be enough. The generic scheduler code is often slow.

In contrast, Real-Time Linux allows users to write their own schedulers. They can be implemented as loadable kernel modules. This makes it possible to easily experiment with different scheduling policies and algorithms and find the ones that best suit the specific application.

There are some schedulers implemented so far. One of them is a priority-based preemptive scheduler implemented as a routine. The scheduling policy is as follows. Each task is assigned a unique priority. If there are several tasks that are ready to execute, the task with the highest priority is executed. Whenever a task becomes ready it will immediately preempt the currently executing task if the current task has a lower priority. Each task is supposed to relinquish the CPU voluntarily. Tasks give up the processor voluntarily, or are preempted by that higher-priority task when its time to execute comes.

The scheduler directly supports periodic tasks. The period and the offset (the starting time) is specified for each of them. An interrupt-driven (sporadic) task can be implemented by defining an interrupt handler that wakes up the needed task.

For periodic tasks with deadlines equal to periods a natural way to assign priorities is given by the *rate monotonic scheduling algorithm*. According to this algorithm, tasks with shorter periods get higher priorities. A set of n independent periodic tasks scheduled by the rate monotonic algorithm is guaranteed to meet all deadlines if

$$\frac{C_1}{T_1}+\frac{C_2}{T_2}+\cdots+\frac{C_n}{T_n}\le n(2^{1/n}-1)$$

where C_i is the worst-case execution time of task i, and T_i is the period of task i. Sporadic tasks can often be treated as periodic ones for priority assignment.

The scheduler treats Linux as the lowest priority real-time task. Thus, Linux only runs when the real-time system has nothing to do. To this end, on switching from Linux to a real-time task, soft interrupt state is remembered, and soft interrupts are disabled. When switching back, soft interrupt state is restored.

Other scheduler was implemented by Ismael Ripoll (http://bernia.disca.upv.es/~iripoll) and it uses the Earliest Deadline First (EDF) algorithm. In this algorithm tasks do not have static priorities. Rather, the task with closest deadline is always chosen to execute.

Related to the scheduling is the mutual exclusion. Any priority-based realtime system has a problem with mutual exclusion. The problem is that when a lower-priority task owns a resource that a higher priority task wants, mutual exclusion algorithms make the higher priority task wait (priorities are "inverted"). The correct solution to this problem is to make clean use of mutual exclusion mechanisms, for example by making sure that operations on shared resources are simple and fast. Other approach is to try to use semaphores to guard use of shared resources. This is not a good idea because it can lead to problems like this:

1) The low priority task acquires a semaphore.
2) The high priority task blocks waiting for the semaphore.
3) A medium priority task runs, keeping the low priority task from getting a chance to release the semaphore.

Some people claim that semaphores using something called "priority inheritance" fix this problem. In priority inheritance when the high-priority task blocks, it raises the priority of the task holding the semaphore, allowing it to complete. This is a dangerous method that is not supported in RTLinux (anyone can implement it or use some one else's implementation in RTLinux, however). RTLinux designers' philosophy is that designers of realtime systems should understand exactly what shared resources there are and should design access methods to these resources that are bounded in time. It should be considered what happens under priority inheritance if the low-priority process acquires the semaphore and waits for a second semaphore, or for an I/O event. Priority inheritance tries to cover up a design problem with a complicated, slow and failure-prone hack.

3.5 - Timing

Precise timing is necessary for the correct operation of the scheduler. Execution schedules often require task switching at specific moments of time. Timing inaccuracies cause deviations from the planned schedule, resulting in so-called *task release jitter*. In most applications task release jitter has an undesirable effect. It is important to minimize it.

One reason for low timer resolution typically found in operating systems is the use of periodic clock interrupts, because in most systems tasks are resumed in the periodic clock interrupt handler. System designers have to trade off the amount of time spent in handling clock interrupts with timer resolution. High clock interrupt rate ensures low jitter, but incurs much overhead. A comparatively low clock interrupt rate does not impose much overhead, but at the same time causes tasks to be resumed either prematurely or too late. Real-time systems sometimes require timer precision that is impossible to get with any reasonable performance using periodic clocks.

Linux is no exception to this rule. On IBM PC compatibles it programs the hardware timer to interrupt at a rate of about 100 Hz. Thus, tasks can be released with only 10 milliseconds precision. In Real Time Linux, this tradeoff is avoided by using a programmable interval timer to interrupt the CPU only when needed, a high-granularity one-shot timer. Specifically, the Intel 8354 timer chip present in some form in all IBM PC compatible

computers is put into the interrupt-on-terminal-count mode. Using this mode, an interrupt can be scheduled with approximately 1 microsecond precision. In this scheme the overhead of interrupt processing is minimal while the timer resolution is high. To keep track of the global time, all intervals between interrupts are summed up together.

The timer interface allows the scheduler to obtain the current time and to register functions to be called at particular moments. Periodic interrupts are simulated for Linux. With soft interrupts it is particularly easy: to imitate an interrupt request, a bit in the pending interrupts mask is set. On the next soft return from interrupt, or soft `sti`, the handler will be invoked.

The interval timer use in Real-Time Linux has its share of problems. Reprogramming the 8354 timer on PCs takes a long time because the timer is not on the processor chip. Fortunately, most modern CPUs, as Pentiums, have timers on-chip in addition to the 8354. It turns out, however, that many applications don't need the generality of a one-shot timer and can avoid the expense of reprogramming. The current RTLinux scheduler offers both periodic and ``one shot'' modes. On SMP systems the problem gets simpler because there is an on-processor high frequency timer chip that is available to the RTLinux system.

3.6 – Interprocess Communication

Since the Linux kernel can be preempted by a real-time task at any moment, no Linux routine can safely be called from real-time tasks. However, some communication mechanism must be present. Simple first-in-first-out (FIFO) buffers are used in RTLinux for moving information between Linux processes or the Linux kernel and real-time processes. They are point-to-point queues of serial data analogous to Unix character devices. These buffers are called real-time FIFOs to distinguish them from the UNIX IPC facility by the same name. RT-FIFO buffers are allocated in the kernel address space. They are referred to by integer numbers. In a data-collecting application, for example, a real time process would poll a device, and put the data into a FIFO. Linux process can then be used for reading the data from the FIFO and storing it in the file or displaying it on the screen

FIFOs have the following characteristics:

- FIFOs queue data, so no protocol is required to prevent data overwrites.
- Boundaries between successive data writes are not maintained. Applications must detect boundaries between data, particularly if data is of varying size.
- FIFOs support blocking for synchronization. Processes need not poll for the arrival of data.
- FIFOs are a point-to-point communication channel. FIFOs do not support the one writer and several readers model.
- The maximum number of FIFOs is declared in **rt_fifo_new.c** as:

```
#define RTF_NO 64
```

and these appear as devices **/dev/rtf0..63** in the filesystem. This limit can be changed and the **rt_fifo_new.o** module recompiled. There is no limit to the number of FIFOs an application can use, or to the size of data that can be written to a FIFO, other than practical memory limits.

The real-time task interface to RT-FIFOs includes creation, destruction, reading and writing functions. Reads and writes are atomic and do not block. Non-blocking avoids the priority inversion problem. RT-FIFOs are, like real-time tasks, never paged out. This eliminates the problem of unpredictable delays due to paging.

Linux user processes, on the other hand, see RT-FIFOs as ordinary character devices. The character device interface gives the users full power of UNIX API for communication with real-time tasks.

An alternative to FIFOs is shared memory, in which a portion of physical memory is set aside for sharing between Linux and RT processes. Shared memory has the following characteristics:

- Shared memory does not queue data written to it. Applications requiring handshaking must define a protocol to assure data is not overwritten.
- Because data is not queued, individual items in data structures megabytes in size can be quickly updated.

- Shared memory has no point-to-point restriction. Shared memory can be written or read by any number of Linux or RT processes.
- The number of independent shared memory channels is only limited by the size of physical memory.
- Blocking for synchronization is not directly supported. To determine if data is new, the data must contain a count that can be compared against previous reads.
- Mutual exclusion of Linux and RT-Linux processes is not guaranteed. Interrupted reads and writes can be detected, however.

The decision to use FIFOs versus shared memory should be based on the natural communication model of the application. For control applications involving processes that execute cyclically based on the expiration of an interval timer, where data queueing is the exception rather than the rule, shared memory is a good choice for communication.

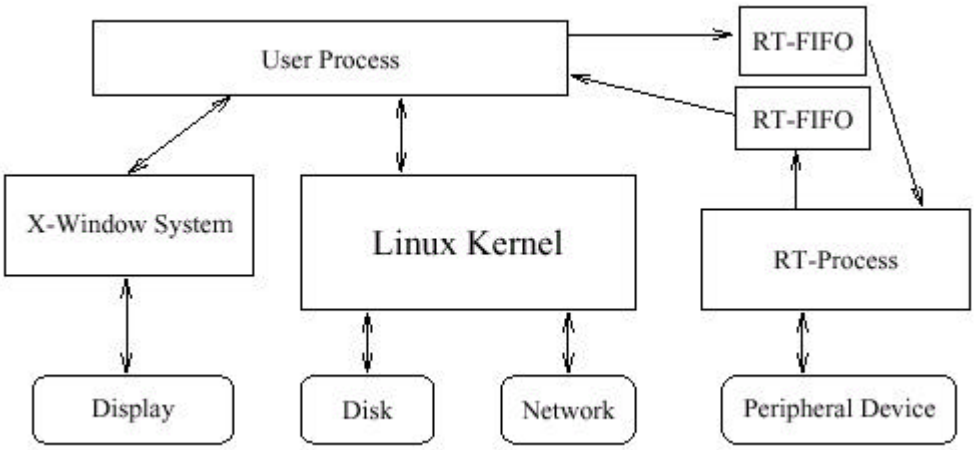
3.7 – Using RTLinux

RTLinux is very much module oriented. To use RTLinux, you load a modules that implement whatever RT capabilities you need. Two of the core modules are the scheduler and the module that implements RT-FIFOS. If the services provided by these modules don't meet the requirements of the application, they can be replaced by other modules. For example, there are two alternative scheduling modules a ``earliest deadline first'' scheduler as commented before, and a rate-monotonic scheduler (see the rtlinux.org web page). The basic scheduler simply runs the highest priority ready RT task until that task suspends itself or until a higher priority task becomes ready.

RTLinux relies on the Linux loadable kernel module mechanism to install components of the RT system and to keep the RT system modular and extensible. Loading a RT module is not a realtime operation and it can also be safely left to Linux. The job of the RT kernel is to provide direct access to the raw hardware for realtime tasks so that they can have minimal latency and maximal processing when they need it.

As it has been said before, each application should be split into real-time and non-real-time parts. The real-time part is as simple as possible. It only includes the code that is directly

time-critical. Low-level communication with hardware often belongs in the real-time part since most hardware imposes timing constraints on the program. The non-real-time part, executed in the user space, on the other hand, implements most of the data processing, including distributing and archiving of data and user interfaces. The two parts communicate using data buffers (RT-FIFOs or shared memory). The data flow in a typical real-time application according to this model should be:



RT-Linux is simple, providing only a bare minimum of functionality necessary for implementing a real-time system. But this simplicity is to the system designer's benefit. The bulk of the application is to be implemented in Linux processes because Linux itself is solid, stable, and popular with a lot of desktop users-so you know you can get help if you have trouble. Real-time tasks should have only the functionality necessary to perform real-time I/O and pass data to and from the Linux processes. The simplicity of RT-Linux has two advantages: first, its very simplicity makes it less likely that it will be buggy; and second, if a

bug id found, it's likely to be easy to find and fix. These factors are important. Because real-time systems are a minuscule portion of Linux applications, the amount of help that can be found for developing code using RT-Linux is certain to be low. So a feature-rich RT-Linux is not necessarily something to be desired. The functionality now implemented is also sufficient for the vast majority of real-time systems if properly used.

3.8 – The API

The most-current stable version of RTLinux, V3.0, offers a POSIX style application programming interface (API), runs on x86, PowerPC and Alpha processors and supports SMP (x86 only). Version 3.0 (final) is available at ftp.rtlinux.com.

The standard API of this last version of RTLinux and its compatibility with POSIX standard is in the **annex 1**. For more information and the explanation of how each function works, visit the RTLinux official website at www.rtlinux.org, or consult the man pages when RTLinux is installed in the computer.

4 – The Application

As it has been said in the introduction, we have to implement a real time-based data acquisition system, with RTLinux and the CANPCI board. To accomplish this, we are going to program an application that controls the CANPCI board, using RTLinux to make the system real time-based.

4.1 – CANPCI board

The CANPCI board was developed in the ENSEIRB. It is not the purpose of this document to explain how this board works or what are its components. There are documentation about all that in the ENSEIRB. Here it will be explained only the things that should be necessarily known to program the application or understanding it.

The CANPCI board is a data acquisition board that can memorize up to 1 M of 24 bits samples. The acquisition maximum frequency at the input of the board is of 55 MHz. Onthe other hand, 8 output bits can be controlled.

The 65 pin connector has:

- input: 24 data bits, acquisition clock.
- output: 8 bits can be individually controlled.

This board is PCI-based and therefore it uses the PCI Bus and its standard. See the **References** chapter to search for documentation about PCI-Devices. The most important things that have to be known of PCI devices by us are:

(i) Each PCI device has two identification numbers, the vendor-id and the device-id, that must be known to identify the device.

(ii) Each PCI device has a configuration block to identify and configure the basic properties for the device. Before a PCI-board can be used, it has to be determined the board specific parameters from the configuration block and set all desired options to operate the board correctly. To do this, it has to be performed the following:

1- Check if the PCI-Bios Support is enabled in the kernel and the BIOS is present with `pcibios_present()`
2- Get the configuration block for the device using `pcibios_find_device()`.
3- Get the desired parameters from the configuration block using `pcibios_read_config_byte()`, or `pcibios_read_config_word()`, or `pcibios_read_config_dword()`.
4- Set the desired parameters in the configuration block using `pcibios_write_config_byte()`, or `pcibios_write_config_word()`, or `pcibios_write_config_dword()`.

There is many setable and readable parameters in the configuration block, that can be accesed through the Linux `pcibios_XXX()` routines. Linux defines some useful Mnemonics to gain simple access to the configuration block parameter values that are defined in `include/linux/pci.h`. These Macros are described in the **annex 2**.

In our application we will use the `PCI_COMMAND` macro to enable I/O area. To do this, `PCI_COMMAND_IO` macro is used for accessing the `PCI_COMMAND` field that enables I/O area. How this is done will be seen in the next section.

(iii) PCI devices have up to 6 memory locations the device can map its memory area(s) to. To access them, macro `PCI_BASE_ADDRESS_[0-5]` is used. The lower 4 bits of the returned base address are used to specify the type and the access mode of the memory area, may it be

an I/O address or a memory mapped area. On an I/O address bit 0 is always 1 and bit 2 is always zero. On a memory mapped address type the lower 4 bits have the following meanings:

Bit 0	Bits 1-2	Bit 3
always zero	address type: 00 = arbitrary 32 bit 01 = below 1M 10 = arbitrary 64 bit	prefetchable

To keep things simple Linux defines two macros: `PCI_BASE_ADDRESS_MEM_MASK` and `PCI_BASE_ADDRESS_IO_MASK` that can be used to strip the type bits from the returned address. This is what we have done as it will be seen in the next section. In this application, I/O area is enabled and all base addresses are I/O addresses.

In the CANPCI board we use 5 of these base addresses to have 4 control and status registers. These registers are very important, because they allow to start the acquisition, read the samples, etc. They allow us to *control* the board. These registers are in the **annex 3**. They are referenced as BADRX (Base ADdRess X).

The Interface Registers are not of our interest. They are at BADR0 and we don't use them.

The Control and Status Register is at BADR1. This 32 bits register has only 10 useful bits at the moment, so other uses could be allowed in the future. ON-OFF bit shows the beginning and the end of the acquisition. It is set ON by the user and automatically OFF at the end of the acquisition. FRONT bit allows to invert the clock that comes from CAN. This way the acquisition is synchronized with the up or down front of the original clock. The last 8 bits (CE0-8) are output bits that we will use to switch on or off 8 LEDs.

The Loading Register is at BADR2 and sets the acquisition size. Because of some CANPCI design reasons, in this register is stored the difference between the maximum acquisition size (1M x 24 bits) and the desired acquisition size as it is shown here:

Loading Register = 1M – desired size

Par example, if we wanted an acquisition of 100 kilo-samples (24 bits per sample), what would have to be stored in the registers would be 1M – 100K = 900000 (decimal).
The number stored in this register doesn't need to be re-stored in each acquisition, because it is memorized in the board.

The RAM Register is at BADR3 and BADR4, it can be accessed at any of these addresses because the register is duplicated. This register is used for reading the samples acquired. To read the samples, it must be read this register a number of times equal to the size of the acquisition. We read from this register all the times, the board design addresses the consecutive accesses to give us the consecutive samples to simplify the reading. So the only thing we must do is to read always from this register at the same memory position.

If the device uses an I/O type based address, normal I/O can be performed using the usual `inb()/outb()`, `inw()/outx()` or the 32 bit `inl()/outl()` routines. If the device uses a memory mapped area it is recommended to use the `readb()/writeb()`, `readw()/writew()` or the longword `readl()/writel()` routines to read or write to a single location.

So, after having said all that before, what is to be done for a data acquisition using these registers is:

1) Store the appropriate value in the Loading Register.
2) Set the ON-OFF bit of Control and Status Register to 1 to start the acquisition.
3) Wait until the ON-OFF bit is reset to zero automatically once the acquisition has finished.
4) Read the samples from the RAM Register and store it elsewhere.
5) Data processing

4.2 – The RT application

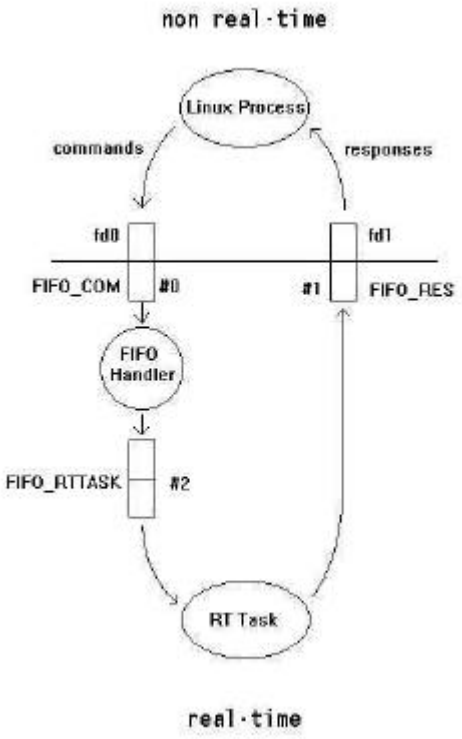
The objective of the application is to control the CANPCI board for doing real-time data acquisitions, so RTLinux is to be used. After installing RTLinux (documentation for successful installation can be found in the **References** chapter) and after running some examples, the design of the control program began.

4.2.1 - Description

The basic idea of how the application works, having in mind what has been said about RTLinux in the previous chapters, is that there is a Linux process which gives us the options for controlling the board and stores the data acquired by it, and there is also a real-time part using RTLinux functions, which actually has to control the board depending of what the user wants, and give the results to the Linux process. These two parts (real-time and non real-time part) will communicate with each other through RT-FIFOs and the help of a FIFO-handler, as we will explain.

In the real-time part there are two important elements: the real-time task and the FIFO handler. This FIFO handler, when it receives the commands from the Linux process, has to make the real-time task work. Until then, the task is "sleeping", it doesn't run, and other processes will be running. But when the task "awakes" and start running, it preempts whatever process was currently running, in our case, the Linux process that is in the non real-time part. This is what we wanted: until we command the board to do something (take samples, load the acquisition size, or anything), we want other processes to be executed in the computer. But when something has to be done by the board, we want it to be done as fast as possible, so we want the other processes to stop being executed and the board to do whatever it has to. This *is* real-time.

The following diagram shows the relationship between the two parts:



4.2.2 - Implementation

The application has been implemented in this way: we divide it in two parts, as it has been said, the real-time part and the non real-time part. The real-time part is implemented as a loadable kernel module written in C with RTLinux features (a file called **canpci_mod.c**), and the non real-time part as a Linux process, which is a program written also in C, called **canpci_ap.c**. There are also two files more, **common.h** and **common.c**, which are used for declaring some common functions and constants. There is of course the appropriate **makefile** to compile the ".c" files into **canpci_mod.o** object file in the case of the module and **canpci_ap** executable file in case of the Linux process.

The Linux process presents us a menu with the commands to send to the CANPCI board or exit the program (terminate the process). The options are:

1) *Choose front*. This is to choose the front for the acquisition clock at the board input. This front can be up or down front.
2) *Start acquisition*. Start a data acquisition of the size chosen with the *Counter load* option and with the acquisition front chosen with the *Choose front* option.

3) *Output drive*. Control the board's 8 output bits, for setting them to 1 or 0.
4) *Counter load*. Ask us for the acquisition size wanted. We have to give the number of 24 bits samples we want to acquire. This number must be between 4 and 1M.
5) *Data read*. Read all the samples acquired previously and stores them in a text file called **samples.dat**.
6) *Exit program*. Finish the program execution (terminate the process).

Apart from the sixth option, if we choose one of the other five options, the process will send to the real-time task a *command* with some *arguments.* This is done through the RT-FIFOs shown in the above diagram. The task will process it, writing or reading the appropriate things to or from the board registers, and will send to the Linux process a *response* with an *argument* that the Linux process will have to process also to know if the command has been carried out by the board. These commands along with their argument and the responses along with their argument are encapsulated into *messages* which are sent to the appropriate FIFOs to the RT task or to the Linux process.

The Linux process builds and puts its messages to the RT-FIFO #0, also called FIFO_COM. When this happens, the FIFO handler awakes (preempting the Linux process) and take the message from the RT-FIFO #0, puts it in the RT-FIFO #2, also called FIFO_RTTASK, and awakes the RT Task. The RT-Task preempts the handler and gets the message from RT-FIFO #2, desencapsulates it and after doing what the command says, builds a message with the response and the argument; puts it in the RT-FIFO #1, also called FIFO_RES, and suspends itself until the handler awakes it. After that, the handler (it had been preempted by the task) finishes its execution doing nothing and the Linux process (also preempted by the handler) gets its turn. It takes the response message from the RT-FIFO #1, desencapsulates it and shows the result in the screen. All this is kept doing until option 6 is chosen to finish the program.

The structure of the messages is: 4 bytes, the first one registers the type of command or response is, and in the other 3 bytes is the argument of the command or response. Commands and responses are numbers, and the response to a command is the same number the command is. For example, if command 3 is sent to the task, it has to respond with response 3.

The command messages sent by the Linux process for each option chosen are the following:

Number	Name	Byte 1	Byte 2	Byte3	Byte 4
1	COM_CHOOSE_FRONT	1	0		0 or 1
2	COM_START_ACQUISITION	2	0		
3	COM_OUTPUT_DRIVE	3	0		Bits
4	COM_COUNTER_LOAD	4	Loading Counter		
5	COM_DATA_READ	5	0		
6	COM_WAIT_ACQUISITION	6	0		

And the response messages sent by the RT Task for each response are the following:

Number	Name	Byte 1	Byte 2	Byte3	Byte 4
1	RES_CHOOSE_FRONT	1	0		
2	RES_START_ACQUISITION	2	0		
3	RES_OUTPUT_DRIVE	3	0		
4	RES_COUNTER_LOAD	4	0		
5	RES_DATA_READ	5	Sample		
6	RES_WAIT_ACQUISITION	6	0		0 or 1

There are 6 commands and therefore 6 responses. Each one is used in each of the menu options, apart from the *Exit program* option. The COM_WAIT_ACQUISITION command and the RES_WAIT_ACQUISITION response are used in the *Start acquisition* option, because when an acquisition is being made, the Linux process must be blocked (waiting) until it finishes. This is done by sending COM_WAIT_ACQUISITION commands to the task. That command, as it will be explained, looks a bit in the board to see if the acquisition has finished. By sending this command to the task until acquisition finishes, the process is blocked here. Being blocked means that other RT tasks could preempt the process, and this is what we want, because tasks always have time constraints an must be executed as fast as possible. If we left our task blocked waiting the end of the acquisition, no other RT task could be executed. The solution is what have been said, instead of the task looking the bit to know when the acquisition finishes, the Linux process sends messages to know it. Later we will see how this is implemented in the program.

4.2.3 – The programs

In this section, it is explained how the two programs work (the real-time module and the Linux process), and how are the other files.

It has been said that the application is made of four files plus a special file to compile the module and the process. These files are

1 – common.h

2 – common.c

3 – canpci_ap.c

4 – canpci_mod.c

and the makefile special file.

The complete commented listing of these files can be found in the **annex 4**. Now it will be explained each file and its function in the application.

1 - **common.h**

This is a header file which contains the declaration of some constants and functions used in the module and the Linux process.

```
#define COM_CHOOSE_FRONT            1
#define COM_START_ACQUISITION       2
#define COM_OUTPUT_DRIVE            3
#define COM_COUNTER_LOAD            4
#define COM_DATA_READ               5
#define COM_WAIT_ACQUISITION        6

#define RES_CHOOSE_FRONT            1
#define RES_START_ACQUISITION       2
#define RES_OUTPUT_DRIVE            3
#define RES_COUNTER_LOAD            4
#define RES_DATA_READ               5
#define RES_WAIT_ACQUISITION        6
```

This constants are used as commands and responses. This way is easier to refer to a command or a response as a name, not as a number which could lead to confusion.

```
unsigned long build_msg(int command, unsigned long argument);
int get_num(unsigned long msg);
unsigned long get_arg(unsigned long msg);
```

This file also declare these three functions which are used by the module and the Linux process.

2 – **common.c**

In this file the previous three functions are implemented.

```
#include "common.h"
```

The file common.h is included, because there the functions implemented here are defined

```
unsigned long build_msg(int command, unsigned long argument)
{
  unsigned long message;

  message = command << 24;
  argument &= 0x00FFFFFF;
  message |= argument;
```

```
  return message;
}
```

This first function builds a message from a command number (1 byte) and an argument (3 bytes) and returns it. The command number is put in first place, and after it, the argument.

```
int get_num(unsigned long msg)
{
  int num;

  msg >>= 24;
  msg &= 0x0000FFFF;
  num = msg;

  return num;
}
```

This function returns the first byte of the message (in fact it returns an integer (16 bits, but the first byte is 0, so it can be treated as having 1 useful byte, the last one), which is the command or response number.

```
unsigned long get_arg(unsigned long msg)
{
  unsigned long arg;

  msg &= 0x00FFFFFF;
  arg = msg;

  return arg;
}
```

The last function returns the argument of the message. As before, it returns an unsigned long (32 bits) but the first byte is always 0, so it can be treated as having 3 useful bytes, the last 3 ones.

3 – **canpci_ap.c**

This is the Linux process that presents us the menu and send the messages to the RT Task. It is the main program that we execute to control the board.

```
#include <stdio.h>
#include <fcntl.h>
```

```
#include <unistd.h>
#include <sys/ioctl.h>
#include <rtl_fifo.h>
#include <rtl_time.h>
#include <stdlib.h>
#include "common.c"
```

Here some necessary header files are included, along with `common.c`, for the common things.

```
int getline(char line[], int max)
{
  int nch = 0;
  int c;
  int i;

  max = max -1;
  while((c = getchar()) != '\n')
    {
      if(nch < max)
     {
       line[nch] = c;
       nch = nch + 1;
     }
    }
  for (i = 0; i < nch; i++)
    if ((line[i] < 48) || (line[i] > 57)) return -1;
  line[nch] = '\0';
  return nch;
}
```

This `getline` function is used for reading strings of numbers from the user. It stores in `line` this string, and after that, `atoi` would be used for knowing the number. This will be necessary in the menu, and to introduce numbers, such as the front and the counter load. It returns –1 if the string contains non-number characters.

```
int wait_acq(int fd0, int fd1)
{
  int com = COM_WAIT_ACQUISITION;
  unsigned long arg_com = 0;
  unsigned long msg_com;
  unsigned long msg_res;
  unsigned long arg_res = 0;
  int res;
  int n;

  msg_com = build_msg(com,arg_com);
  if (write(fd0, &msg_com, sizeof(msg_com)) < 0)
    {
      fprintf(stderr, "Can't send a command to RT-task\n");
      exit(1);
    }
  printf("Process sending the \"%d\" command to task with the
  \"%ld\" argument.\n", com, arg_com);
```

```
  // WAIT FOR RESPONSE

  if ((n = read(fd1, &msg_res, sizeof(msg_res))) == sizeof(msg_res))
    {
      res = get_num(msg_res);
      arg_res = get_arg(msg_res);
      printf("Process received the \"%d\" result from task with the
      \"%ld\" argument.\n", res, arg_res);
    }
  return arg_res;
}
```

This function is used when the acquisition starts, to see when this acquisition finishes. This way, the Linux process is blocked waiting, as it was explained before. What this function does is send messages with the COM_WAIT_ACQUISITION to the task, by putting it in the FIFO #0. Then it waits for the response from the task in the FIFO #1. The function returns the argument of this response, which will register whether the acquisition has finished (argument = 0) or not (argument = 1).

```
int main()
{
  int fd0, fd1;                    // file descriptors for the FIFOs
  int n;                           // used for reading
  int com = 0;                     // command to send
  int res = 0;                     // result received
  int choice;                      // option chosen in the menu
  char line[256];                  // buffer to read numbers with
                                      'getline' function
  int front;                       // type of front chosen
  unsigned char reg;               // for the output drive
  unsigned char output_drive = 0;  // for the output drive
  int i;                          // index in the output_drive option
  unsigned long samples = 0;       // acquisition size
  unsigned long arg_com = 0;       // command's argument
  unsigned long arg_res = 0;       // response's argument
  unsigned long msg_com;           // message that must be passed to
                                      FIFO_COM with the command
  unsigned long msg_res;           // message that must be received
                                      from FIFO_RES with the response
  int get_line;                    // to see whether the return
                                      value of getline is -1 or not
  int nosend;                      // used in COM_OUTPUT_DRIVE
  FILE *fp;                        // file pointer to the file where
                                       the samples will be written in
```

These are self-explained variables used in the program.

```
  if ((fd0 = open("/dev/rtf0", O_WRONLY)) < 0)
  {
    fprintf(stderr, "Error opening /dev/rtf0\n");
    exit(1);
```

```
}

if ((fd1 = open("/dev/rtf1", O_RDONLY)) < 0)
{
  fprintf(stderr, "Error opening /dev/rtf1\n");
  exit(1);
}
```

Here two FIFOs are opened in the Linux side as character devices (this is how the Linux processes treat the RT-FIFOs). The rtf0 (FIFO #0) corresponds to fd0, and is opened as a write-only device (to send messages--commands), and rtf1 (FIFO #1) corresponds to fd1 and is opened as a read-only device (to receive messages—responses).

```
// THE MENU

while (1)
  {
    printf("\nCommands to send to the CANPCI board : \n\n");
    printf("1- Choose front (UP/DOWN)\n");
    printf("2- Start acquisition\n");
    printf("3- Output drive (8 bits)\n");
    printf("4- Counter load (Up to 1M)\n");
    printf("5- Data read\n\n");
    printf("6- Exit program (not to send)\n\n");
    printf("Choose one : ");

    if ((get_line = getline(line, 256)) == -1)
   {
     choice = get_line;
   }
    else
   {
     choice = atoi(line);
   }
    printf("\nIt is : %d\n",choice);
```

The main menu is presented and a valid option has to be chosen. Depending on what this option is, the program will send a command or finish the execution.

```
    // OPTIONS

    switch (choice) {

   // CHOOSE FRONT

    case 1:
     com = COM_CHOOSE_FRONT;
     while (1)
       {
         printf("UP - 0 / DOWN - 1 : ");
         if ((get_line = getline(line, 256)) == -1)
```

```
        {
          front = get_line;
        }
         else
        {
          front = atoi(line);
        }
         if ((front == 0) || (front == 1 )) break;
         printf("Not a good value. Try again.\n");
      }
    arg_com = front;
    arg_com &= 0x00000001;
    printf("arg is : %ld\n",arg_com);
    printf("Option chosen is : %d\n",com);
    getchar();

    msg_com = build_msg(com,arg_com);
    if (write(fd0, &msg_com, sizeof(msg_com)) < 0)
      {
        fprintf(stderr, "Can't send a command to RT-task\n");
        exit(1);
      }
    printf("Process sending the \"%d\" command to task with the
    \"%ld\" argument.\n", com, arg_com);

    // WAIT FOR RESPONSE

    if ((n = read(fd1, &msg_res, sizeof(msg_res))) ==
    sizeof(msg_res))
      {
        res = get_num(msg_res);
        arg_res = get_arg(msg_res);
        printf("Process received the \"%d\" result from task with
         the \"%ld\" argument.\n", res, arg_res);
        printf("FRONT bit has been set to %ld\n", arg_com);
        getchar();
      }
    break;
```

In case we want to choose the clock front for the acquisition, we must choose between up front (value 0) and down front (value 1).Then, a message is built with the COM_CHOOSE_FRONT command and 0 or 1 as argument, and it is put in the FIFO_COM to send it to the task. Once this is done, a response is expected from FIFO_RES. When it arrives the only thing the program does is show the results.

```
// START ACQUISITION

 case 2:
  com = COM_START_ACQUISITION;
  arg_com = 0;

  msg_com = build_msg(com,arg_com);
```

```
if (write(fd0, &msg_com, sizeof(msg_com)) < 0)
  {
    fprintf(stderr, "Can't send a command to RT-task\n");
    exit(1);
  }
printf("Process sending the \"%d\" command to task with the
\"%ld\" argument.\n", com, arg_com);

// WAIT FOR RESPONSE

if ((n = read(fd1, &msg_res, sizeof(msg_res))) ==
sizeof(msg_res))
  {
    res = get_num(msg_res);
    arg_res = get_arg(msg_res);
    printf("Process received the \"%d\" result from task with
     the \"%ld\" argument.\n", res, arg_res);
  }
printf("Waiting...\n");

while (wait_acq(fd0, fd1));

printf("Acquisition finished.\n");
getchar();
break;
```

This is to start an acquisition. First, a message is built with the COM_START_ACQUISITION command and 0 as argument. This will make the acquisition to start. The message is put in FIFO_COM and the response is expected. Once this response arrives, the process waits until the acquisition finishes. This is done with: `while (wait_acq(fd0, fd1))`. The program loops here until `wait_acq` returns 0, in which case the acquisition is finished.

```
// OUTPUT DRIVE

 case 3:
  com = COM_OUTPUT_DRIVE;
  while (1)
    {
      nosend = 0;
      output_drive = 0;

      printf("Bits for the output drive (8) : ");
      i = 7;
      while((reg = getchar()) != '\n')
     {
       if (reg == '0') reg = 0;
       else
         {
           if (reg == '1') reg = 1;
           else
          {
```

```
                printf("\nBit %d is not valid. It must be 0 or
                1.\n", 8 - i);
                nosend = 1;
              }
            }
          reg = (reg << i);
          printf("reg = %x\n",reg);
          output_drive = (output_drive | reg);
          printf("output_drive = %x\n",output_drive);
          arg_com = (output_drive & 0x000000FF);
          printf("arg = %lx\n", arg_com);
          i -= 1;
        }
         if (i != -1) {
            printf("\nNot valid. They must be 8 bits.\n");
            nosend = 1;
            getchar();
         }
         printf("Bits are : %x\n",output_drive);
         if (!nosend) break;
      }
    printf("Option chosen is : %d\n",com);
    getchar();

    msg_com = build_msg(com,arg_com);
    if (write(fd0, &msg_com, sizeof(msg_com)) < 0)
      {
        fprintf(stderr, "Can't send a command to RT-task\n");
        exit(1);
      }
    printf("Process sending the \"%d\" command to task with the
    \"%ld\" argument.\n", com, arg_com);

    // WAIT FOR RESPONSE

    if ((n = read(fd1, &msg_res, sizeof(msg_res))) ==
    sizeof(msg_res))
      {
        res = get_num(msg_res);
        arg_res = get_arg(msg_res);
        printf("Process received the \"%d\" result from task with
        the \"%ld\" argument.\n", res, arg_res);
        printf("Output bits has been set to %x\n", output_drive);
        getchar();
      }
       break;
```

With this option we set the board's output bits to 0 or 1. First of all we must introduce 8 bits. This means we must introduce 8 values of 0 or 1. This is done by the use of the variable `nosend`, that is set to 1 when the values or the number of bits introduced are not correct. To get out of the `while(1)` loop, `nosend` must be 0, this can be seen in: `if (!nosend) break`. These correct 8 bits are the argument of the message along with the first two bytes

equal to 0. The command is, of course, COM_OUTPUT_DRIVE. This message is put in FIFO_COM and a response is expected, as always. When the response arrives, the program shows the results.

```
// COUNTER LOAD

 case 4:
  com = COM_COUNTER_LOAD;
  while (1)
    {
      printf("Acquisition size (Up to 1M) : ");
      if ((get_line = getline(line, 256)) == -1)
     {
       samples = get_line;
     }
      else
     {
       samples = atoi(line);
     }
      if ((samples > 0) && (samples <= 0x100000)) break;
      printf("Not a good value. Try again.\n");
    }
  arg_com = 0x100000 - samples;
  printf("arg is : %ld\n",arg_com);
  printf("arg is : %lx\n",arg_com);
  printf("Option chosen is : %d\n",com);
  getchar();

  msg_com = build_msg(com,arg_com);
  if (write(fd0, &msg_com, sizeof(msg_com)) < 0)
    {
      fprintf(stderr, "Can't send a command to RT-task\n");
      exit(1);
    }
  printf("Process sending the \"%d\" command to task with the
  \"%ld\" argument.\n", com, arg_com);

  // WAIT FOR RESPONSE

  if ((n = read(fd1, &msg_res, sizeof(msg_res))) ==
  sizeof(msg_res))
    {
      res = get_num(msg_res);
      arg_res = get_arg(msg_res);
      printf("Process received the \"%d\" result from task with
       the \"%ld\" argument.\n", res, arg_res);
      printf("The counter has been loaded to take %ld (decimal)
      %lx (hex) samples\n", samples, samples);
      getchar();
    }
  break;
```

With this option we have to give to the process the data acquisition size desired, this is, the number of samples we want. It must be less than 1M samples. When the value introduced is correct, and due to the board design, the value that must be loaded in the Loading Register is *1M – desired size*, as it was seen. This is done in: `arg_com = 0x100000 - samples`, where we store in the message's argument the correct value to be loaded in the Loading Register. The message is built with this argument and the COM_COUNTER_LOAD command. Then the same as other options is done: put the message in FIFO_COM and a response is expected.

```
// DATA READ

 case 5:
  com = COM_DATA_READ;
  arg_com = 0;
  printf("Option chosen is : %d\n",com);
  getchar();

  fp = fopen("samples.dat", "w+");
  msg_com = build_msg(com,arg_com);
  for(i = 0; i < samples; i++)
    {
      if (write(fd0, &msg_com, sizeof(msg_com)) < 0)
     {
       fprintf(stderr, "Can't send a command to RT-task\n");
       exit(1);
     }
      printf("Process sending the \"%d\" command to task with
      the \"%ld\" argument.\n", com, arg_com);

      // WAIT FOR RESPONSE

      if ((n = read(fd1, &msg_res, sizeof(msg_res))) ==
      sizeof(msg_res))
     {
       res = get_num(msg_res);
       arg_res = get_arg(msg_res);
       printf("Process received the \"%d\" result from task
       with the \"%ld\" argument.\n", res, arg_res);
       printf("The sample is : %lx\n", arg_res);
     }
      fprintf(fp, "%lx\n", arg_res);
    }
  break;
```

With this option the acquired samples are going to be read from the CANPCI board and stored in a file called "samples.dat". The file `samples.dat` is opened for writing and erasing the previous data contained in it. A message is built with the COM_DATA_READ command and 0 as argument. But, as this command makes the task to read only one sample, this

message is put `samples` times in FIFO_COM with: `for(i = 0; i < samples; i++)`, because in `samples` is stored the number of samples previously acquired, and these are the samples we want to read from the board. After each message is put, a response message is expected. In this message, the sample is the argument, so this argument is written in the file and another iteration is made (this means another command message is put in FIFO_COM).

```
// EXIT

 case 6:
  printf("Exiting now...\n");
  return 0;
  break;
```

This option is to exit the program and finish it.

```
// INVALID OPTION

 default:
  printf("Not a valid option. Choose again.\n");
  com = 0;
  arg_com = 0;
 }
}
```

When an invalid option is chosen.

```
  return 0;

}
```

4 – **canpci_mod.c**

This is the module that implements the real-time part of the application. It is programmed as a kernel loadable module, so it has to have two routines: the `init_module` routine that will be invoked when the module is loaded and the `cleanup_module`, that will be invoked when the module is unloaded. Apart from these, it has two other important routines: `thread_code`, this is the real-time task that controls the board; and `my_handler`, the handler that takes the messages from the Linux process in FIFO_COM and puts it in FIFO_RTTASK for the task to get them, awakening the task after putting the messages in FIFO_RTTASK.

The RTLinux functions used here are explained in the **annex 5**. It also are explained in the man pages of Linux `man` call.

```
#include <linux/errno.h>
#include <rtl.h>
#include <time.h>
#include <rtl_sched.h>
#include <rtl_fifo.h>
#include <linux/pci.h>
#include <asm/io.h>
#include "common.c"
```

Here some necessary header files are included, along with `common.c`, for the common things.

```
#define FIFO_COM     0
#define FIFO_RES     1
#define FIFO_RTTASK  2

#define VENDOR_ID_CANPCI 0x10e8
#define DEVICE_ID_CANPCI 0x8110
```

Definition of the FIFOs numbers and the necessary vendor and device-ids.

```
unsigned long canpci_ioaddr[5];  // contains the canpci IO addresses
```

In this variable the base addresses of the registers will be stored in the board initialization.

```
struct pci_dev *dev = NULL;     // points to the configuration block

pthread_t mytask;

void *thread_code(void *t)
{
  int com;                      // command received
  int res;                      // result to send
  unsigned long msg_com;        // message received with the command
  unsigned long msg_res;        // message sent with the response
  unsigned long arg;            // argument received
  unsigned long reg;            // buffer for the registers

  pthread_suspend_np(pthread_self());
```

The first time the task runs (when the module is loaded and in `init_module` the task is created), it has to wait for a command to arrive in order to start.

```
  while (1) {
    int err;
    while ((err = rtf_get (FIFO_RTTASK, &msg_com, sizeof(msg_com)))
    == sizeof(msg_com))
    {
```

```
com = get_num(msg_com);
arg = get_arg(msg_com);
rtl_printf("Task executing the \"%d\" command with the \"%x\"
argument.\n", com, arg);
```

Here the task enters and endless loop until the module is unloaded. In this loop the first to do is get the command message from the Linux process in FIFO_RTTASK, where it has been put by the handler from the Linux process. It gets the command number and the argument. Depending on the command number, there are some things to do.

```
// COMMANDS RECEIVED

switch (com) {

   case COM_CHOOSE_FRONT:
    reg = inl(canpci_ioaddr[1]);
    if(arg)
      reg |= 0x40000000;
    else
      reg &= 0xBFFFFFFF;
    outl(reg, canpci_ioaddr[1]);
    rtl_printf("canpci_ioaddr[1] = %lx\n", reg);

    res = RES_CHOOSE_FRONT;
    rtl_printf("Task sending the \"%d\" result.\n", res);
    msg_res = build_msg(res, 0);
    rtf_put(FIFO_RES, &msg_res, sizeof(msg_res));
    break;
```

In this case, the task has to set the front. To do this, the bit 30 of the Control and Status Register must be set to 1 (down front) or set to 0 (up front). The information about the front chosen is in the argument of the command message. This register is at BADR1, so its address is in `canpci_ioaddr[1]`. To set the bit, the task gets the register using `inl()` and applies it the appropriate mask. This is what will be done to set all the bits in the registers in all of the other cases. After setting this bit using `outl()` to write the register, the response message is built and put in FIFO_RES for the Linux process to get it.

```
   case COM_START_ACQUISITION:
    reg = inl(canpci_ioaddr[1]);
    reg |= 0x20000000;
    outl(reg, canpci_ioaddr[1]);
    rtl_printf("canpci_ioaddr[1] = %lx\n", reg);

    res = RES_START_ACQUISITION;
    rtl_printf("Task sending the \"%d\" result.\n", res);
    msg_res = build_msg(res, 0);
```

```
  rtf_put(FIFO_RES, &msg_res, sizeof(msg_res));
  break;
```

Here the task has to make the CANPCI board to start an acquisition. To do this, the bit 29 of the Control and Status Register must be set to 1, as we saw. So this is what the task does, in the same way as before. And after that, builds the response and puts it in FIFO_RES.

```
case COM_WAIT_ACQUISITION:
  reg = inl(canpci_ioaddr[1]);
  reg &= 0x20000000;
  reg >>= 29;

  res = RES_WAIT_ACQUISITION;
  rtl_printf("Task sending the \"%d\" result.\n", res);
  msg_res = build_msg(res, reg);
  rtf_put(FIFO_RES, &msg_res, sizeof(msg_res));
  break;
```

This is to see whether the acquisition is finished or not. To do it, it must be checked the value of the bit 29 of the Control and Status Register. Then, it builds a response message with this value of the bit (in `reg`) as argument. If the bit is 0, the acquisition is finished. If it is 1, the acquisition is not yet finished. It must be reminded that to start an acquisition, we set this bit to 1.

```
case COM_OUTPUT_DRIVE:
  reg = inl(canpci_ioaddr[1]);
  reg &= 0xFFFFFF00;
  reg |= arg;
  outl(reg, canpci_ioaddr[1]);
  rtl_printf("canpci_ioaddr[1] = %lx\n", reg);

  res = RES_OUTPUT_DRIVE;
  rtl_printf("Task sending the \"%d\" result.\n", res);
  msg_res = build_msg(res, 0);
  rtf_put(FIFO_RES, &msg_res, sizeof(msg_res));
  break;
```

Here the 8 output bits (bits 7 to 0 of the Control and Status Register) are set to 0 or 1 by writing in the appropriate positions at BADR1 (`canpci_ioaddr[1]`). After that, as always, the response message is put in FIFO_RES.

```
case COM_COUNTER_LOAD:
  reg = inl(canpci_ioaddr[2]);
  reg &= 0xFFF00000;
  reg |=arg;
  outl(reg, canpci_ioaddr[2]);
```

```
          rtl_printf("canpci_ioaddr[2] = %lx\n", reg);

          res = RES_COUNTER_LOAD;
          rtl_printf("Task sending the \"%d\" result.\n", res);
          msg_res = build_msg(res, 0);
          rtf_put(FIFO_RES, &msg_res, sizeof(msg_res));
          break;
```

In this case it is loaded in the Loading Counter the appropriate value, which arrives as the argument of the command message in FIFO_COM. This is done by writing in the appropriate positions (the last 20 bits) at BADR2 (`canpci_ioaddr[2]`). After that, as always, the response message is put in FIFO_RES.

```
        case COM_DATA_READ:
          reg = inl(canpci_ioaddr[3]);
          reg &= 0x00FFFFFF;

          res = RES_DATA_READ;
          rtl_printf("Task sending the \"%d\" result.\n", res);
          msg_res = build_msg(res, reg);
          rtf_put(FIFO_RES, &msg_res, sizeof(msg_res));
          break;
```

This is to read a sample from the board. To do this, it must be read the last 24 bits of the RAM Register where the samples are. The sample is stored temporarily in `reg`, and the response message is built with `reg` as its argument. This message is put in FIFO_RES for the Linux process to get the sample.

```
        default:
          rtl_printf("RTL task: bad command\n");
          return 0;
      }
    }
```

If a non-valid command number arrives in FIFO_COM, the task would finish itself after writing an error message.

```
    rtl_printf("RTL task waiting\n");
    pthread_suspend_np(pthread_self());
    rtl_printf("RTL task awakes\n");
  }
  return 0;
}
```

When the command message is processed and the response message is sent, the job has been finished for the task, so it suspends itself in `pthread_suspend_np(pthread_self())`

until the handler wakes it up (when a command message arrives in FIFO_COM sent by the Linux process)

```
int my_handler(unsigned int fifo)
{
  int err;
  unsigned long msg;

  while ((err = rtf_get(FIFO_COM, &msg, sizeof(msg))) ==
  sizeof(msg))
    {
      rtf_put (FIFO_RTTASK, &msg, sizeof(msg));
      rtl_printf("FIFO handler: sending the \"%d\" command to
       task.\n", get_num(msg));
      rtl_printf("FIFO handler wakes up with %d\n", get_num(msg));
      pthread_wakeup_np (mytask);
    }

  rtl_printf("FIFO handler is dead\n");
  return 0;
}
```

This is a handler for real-time FIFO data, which is associates with FIFO_COM. This means that when a message is written to FIFO_COM by the Linux process this code is executed as it is the handler of this RT-FIFO. The handler gets the command message from FIFO_COM, puts it in FIFO_RTTASK and awakes the suspended task. When the task suspends itself again, the handler tell us that it finishes and finishes its execution until it is executed again when the Linux process puts another command message in FIFO_COM for the task

```
/* #define DEBUG */
```

`DEBUG` can be defined or not, depending on whether debug is to be done or not.

```
int init_module(void)
{
  int c[4];                       // for debug
  int ret;                        // return value of pthread_create()
  u16 pci_command, new_command;  // to initialize the board


  rtf_destroy(FIFO_COM);
  rtf_destroy(FIFO_RES);
  rtf_destroy(FIFO_RTTASK);

  c[0] = rtf_create(FIFO_COM, 100);
  c[1] = rtf_create(FIFO_RES, 100);
  c[2] = rtf_create(FIFO_RTTASK, 100);
```

The three RT-FIFOs to be used in the module are first removed (`rtf_destroy`) then created (`rtf_create`). This is done in this way to be sure that neither they are not being used by other programs nor they have no data stored in.

```
#ifdef DEBUG
  printk("c[0] = %d\n", c[0]);
  printk("c[1] = %d\n", c[1]);
  printk("c[2] = %d\n", c[2]);
#endif
```

If debugging is being made, this shows the results of the RT-FIFOs creation.

```
  // INITIALIZE THE BOARD

  // look for pci card

  if (!pcibios_present())
    {
      printk ("Not PCI bios present.\n");
      return -1;
    }
```

This is to check if the PCI-Bios Support is enabled in the kernel and the BIOS is present. If it isn't, the module finishes with an error code of –1.

```
  dev=pci_find_device (VENDOR_ID_CANPCI, DEVICE_ID_CANPCI, dev);
```

Get the configuration block for the device, and `dev` points to it.

```
  if (dev == NULL)
    {
      printk("No CANPCI card found.\n");
      return -1;
    }
```

If `dev` points to `NULL`, the card has not been found, and the module finishes with an error code of –1.

```
  // This is to enable I/O area

  pci_read_config_word(dev, PCI_COMMAND, &pci_command);
  new_command = pci_command | PCI_COMMAND_IO;
  if (pci_command != new_command)
    {
      printk(KERN_INFO "  The PCI BIOS has not enabled this"
           " CANPCI!  Updating PCI command %4.4x->%4.4x.\n",
           pci_command, new_command);
      pci_write_config_word(dev, PCI_COMMAND, new_command);
    }
```

Here I/O area is enabled. First, we get the `PCI_COMMAND` word in the configuration block (`pci_read_config_word(dev, PCI_COMMAND, &pci_command)`) and store it in `pci_command`. To enable I/O area, `PCI_COMMAND_IO` mask is applied to `pci_command` (`new_command = pci_command | PCI_COMMAND_IO`). After that, it is checked if I/O area is already enabled by the PCI_Bios. If it is, there is nothing to be done, because it is already done. If not, the new `PCI_COMMAND` word is written in the configuration block and I/O area is therefore enabled.

```
  canpci_ioaddr[0]= dev->base_address[0] & PCI_BASE_ADDRESS_IO_MASK;
  canpci_ioaddr[1]= dev->base_address[1] & PCI_BASE_ADDRESS_IO_MASK;
  canpci_ioaddr[2]= dev->base_address[2] & PCI_BASE_ADDRESS_IO_MASK;
  canpci_ioaddr[3]= dev->base_address[3] & PCI_BASE_ADDRESS_IO_MASK;
  canpci_ioaddr[4]= dev->base_address[4] & PCI_BASE_ADDRESS_IO_MASK;
```

Here the base addresses of the CANPCI registers are stored. This is done by applying the `PCI_BASE_ADDRESS_IO_MASK` mask to strip the type bits from the base address pointed by `dev->base_address[X]`. By applying this mask, all base addresses are I/O addresses and are in `canpci_ioaddr[X]`.

```
  // THREAD

  ret = pthread_create (&mytask, NULL, thread_code, NULL);
  rtf_create_handler(FIFO_COM, &my_handler);
  return 0;
}
```

The thread is created and starts running, and the FIFO handler is declared as being associated with FIFO_COM.

```
void cleanup_module(void)
{
#ifdef DEBUG
  printk("0 = %d\n", rtf_destroy(FIFO_COM));
  printk("1 = %d\n", rtf_destroy(FIFO_RES));
  printk("2 = %d\n", rtf_destroy(FIFO_RTTASK));
```

If debugging is being made, this shows the results of the RT-FIFOs removal.

```
#else
  rtf_destroy(FIFO_COM);
  rtf_destroy(FIFO_RES);
  rtf_destroy(FIFO_RTTASK);
```

If debugging is not being made, here the RT-FIFOs are removed to free them for other uses.

```
#endif
  pthread_cancel (mytask);
  pthread_join (mytask, NULL);
}
```

This is to terminate the thread when the module is unloaded. What the `cleanup_module` has to accomplish is that anything done by the `init_module` has to be undone.

4.2.4 – Typical Data Acquisition example

Firstly, we have to compile the source files **canpci_ap.c** and **canpci_moc.c** to get the object files: **canpci_ap**, an executable file (the *Linux process*) and **canpci_mod.o**, which is the *module* to load in the kernel. To do this simpler it has been written a **Makefile**, whose code is the following:

```
all: canpci_mod.o canpci_ap

include ../rtl.mk

clean: rm -f canpci_ap *.o

canpci_ap: canpci_ap.c
     $(CC) ${INCLUDE} ${USER_CFLAGS} -O2 -Wall canpci_ap.c -o
canpci_ap

include $(RTL_DIR)/Rules.make
```

By doing this, typing `make` in Linux will compile the files. It has to be taken into account that the necessary file **rtl.mk** has to be in the previous directory. This code can be modified to reflect the actual location of **rtl.mk**.

The next step is to load all the RTLinux modules necessaries to run whatever application. There is a batch file to do this, **insrtl**. After that, with `insmod canpci_mod.o` in Linux our application module is loaded. Then **canpci_ap** is executed typing `canpci_ap`.

In our example, we used a board that generates a digital ramp from a frequency generator. This digital ramp is increased one by one. This board also has 8 LEDs connected to the output bit of the CANPCI board. If a bit is set to 1, the corresponding LED is switched on, and if set to 0, the LED is switched off.

With all this set up, we run the application as it has been said. After checking that the option 3 (output bits) works correctly as the appropriate bits are switched on, we choose the clock front as DOWN with option 1. Then, with option 4 we set the acquisition size to 10 samples and choose option 2 to start the acquisition. After a while, the samples are taken.
With option 5 (data read) now all the samples are read and stored in **samples.dat**. Then we look into the file and see the samples: 979797, 989898, 999999, 9A9A9A, 9B9B9B, 9C9C9C, 9D9D9D, 9E9E9E, 9F9F9F, A0A0A0.

This is what it has to get, because the other board generates a digital ramp, but with our acquisition frequency each step is registered three times in the sample acquired until it is increased by one.

After the acquisition, with option 6 we exit and terminate the Linux process. Then we unload the module with `rmmod canpci_mod`, and unload the rest of the modules with another batch file, **rmrtl**.

5 - Conclusion

I must say that this work has been very interesting for me, because at first I didn't know much about real-time OS and its application in real life problems and industry. I also had little knowledge about module programming under Linux. Now, after having done and read about the matter, I am satisfied of having committed myself to such work.

I also have realized how these real-time OS can be used in embedded system to achieve objectives which couldn't be achieved before. And nowadays the embedded systems field is very important, because almost every electronic machine has integrated circuits on board and many times they have to be controlled in such a way they have to meet some time constraints. The best way to do this is with a little real-time OS kernel on board the machine that controls it to meet these constraints. One example is the modern UMTS mobile phones field. In this technology, the mobile phone has to do many things apart from only allowing speech, and a mini-realtime kernel could be used.

One thing that has surprised me very much is the RTLinux solution of the problem of implementing the real-time features. Before I started this job, I thought that OS kernels were untouchable and nothing could be done to them. But after understanding the operation of RTLinux, I have this idea no longer.

The only bad thing about RTLinux is that because of being so new, it lacks of documentation as other real-time OS have. But I think this is only in printed documentation, because in Internet can be found several helpful documents, specially in www.rtlinux.org.

So, as a conclusion, I can say that this project has allowed me to better understanding and learn more about real-time features and problems, which are very important nowadays, and its implementation in a modern environment such as RTLinux. And I think that the results achieved are good, because the application works correctly and allows a fast data acquisition even when the system is heavy loaded, that it was we have to achieve.

6 – References

World Wide Web:

- www.fsmlabs.com
- www.rtlinux.org
- bernia.disca.upv.es/~iripoll

Books:

- Jack G. Ganssle: *"The Art of Programming Embedded Systems"*, Ed. Academic press 1991.
- J.J. Labrosse: *"µC/OS-II: the real time kernel"*, Ed. R&D Books.

Documents:

- M. Barabanov: *"A Linux-based Real-Time Operating System"*
- V. Yodaiken: *"The RTLinux Manifesto"*
- V. Yodaiken and M. Barabanov: *"A Real-Time Linux"*
- J. Esplin: *"Linux as an Embedded Operating System"*
- M. Barabanov and V. Yodaiken: *"Real-Time Linux"*
- FSMLabs: *"RTLinux FAQ"*
- V. Yodaiken: *"Cheap operating systems research and teaching with Linux"*
- F.M. Proctor: *"Using Shared Memory in Real-Time Linux"*
- H. Bruyninckx: *"Real Time and Embedded HOWTO"*, K.U. Leuven
- Claus Schroeter: *"Programming PCI-Devices under Linux"*
- P.Kadionik: *"Drivers de la carte CANPCI sous MacOS et sous MSDOS"*, ENSEIRB
- R. Baruch and C. Schroeter: *"Writing Character Device Driver for Linux"*

ANNEX 1 – RTLinux API

sigaction.2
clock_gethrtime.3
free_RTirq.3
gethrtime.3
pthread_attr_getcpu_np.3
pthread_attr_setcpu_np.3
pthread_delete_np.3
pthread_make_periodic_np.3
pthread_setfp_np.3
pthread_suspend_np.3
pthread_wait_np.3
pthread_wakeup_np.3
request_RTirq.3
rt_com.3
rt_com_read.3
rt_com_setup.3
rt_com_table.3
rt_com_write.3
rt_get_time.3
rt_task_delete.3
rt_task_init.3
rt_task_make_periodic.3
rt_task_suspend.3
rt_task_wait.3
rt_task_wakeup.3
rt_use_fp.3
rtf_create.3
rtf_create_handler.3
rtf_destroy.3
rtf_get.3
rtf_flush.3
rtf_make_user_pair.3
rtf_link_user_ioctl.3
rtf_put.3
rtl_allow_interrupts.3
rtl_free_irq.3
rtl_free_soft_irq.3
rtl_get_soft_irq.3
rtl_getcpuid.3
rtl_getschedclock.3
rtl_get_soft_irq.3
rtl_global_pend_irq.3
rtl_hard_disable_irq.3
rtl_hard_enable_irq.3
rtl_no_interrupts.3

rtl_printf.3
rtl_request_irq.3
rtl_restore_interrupts.3
rtl_setclockmode.3
rtl_stop_interrupts.3
rtf.4
rtl_v1.3

POSIX Functions

RTLinux also supports a subset of the POSIX interface. The following POSIX interface functions are available in this release:

clock_gettime
clock_settime
time
usleep
nanosleep
clock_nanosleep
sched_get_priority_max
sched_get_priority_min
pthread_self
pthread_attr_init
pthread_attr_getstacksize
pthread_attr_setstacksize
pthread_attr_getschedparam
pthread_attr_setschedparam
pthread_attr_getdetachstate
pthread_attr_setdetachstate
sched_yield
pthread_getschedparam
pthread_setschedparam
pthread_create
pthread_exit
pthread_cancel
pthread_setcancelstate
pthread_setcanceltype
pthread_join
pthread_kill (signals supported: 0, RTL_SIGNAL_SUSPEND,
RTL_SIGNAL_WAKEUP, RTL_SIGNAL_CANCEL)
pthread_mutexattr_getpshared
pthread_mutexattr_setpshared
pthread_mutexattr_init
pthread_mutexattr_destroy
pthread_mutex_init
pthread_mutex_destroy
pthread_mutexattr_gettype

pthread_mutexattr_settype
pthread_mutex_lock
pthread_mutex_trylock
pthread_mutex_unlock
sysconf
uname

The following group of functions is supported if _POSIX_THREAD_PRIO_PROTECT options is defined:

pthread_mutexattr_getprotocol
pthread_mutexattr_setprotocol
pthread_mutexattr_getprioceiling
pthread_mutexattr_setprioceiling

POSIX condition variables:

pthread_condattr_getpshared
pthread_condattr_setpshared
pthread_condattr_init
pthread_condattr_destroy
pthread_cond_init
pthread_cond_destroy
pthread_cond_wait
pthread_cond_timedwait
pthread_cond_broadcast
pthread_cond_signal

POSIX semaphores:

sem_init
sem_destroy
sem_getvalue
sem_wait
sem_trywait
sem_post
sem_timedwait

Note:

Certain RTLinux API functions have restrictions on their use. Some functions may only be called during RTLinux module initialization and cleanup (e.g., rtf_create, rtf_destroy). Other functions may not be used to operate on threads running on processors other than the current

one (pthread_make_periodic_np). The following functions should not be used in interrupt handlers:

pthread_delete_np
pthread_wait_np
usleep
nanosleep
clock_nanosleep
pthread_self
pthread_exit
pthread_join
pthread_mutex_lock
pthread_mutex_trylock
pthread_cond_wait
sem_wait
sem_trywait
sem_timedwait

See individual manual pages for additional information. Except if listed in this section or in the individual manual pages, there are no restrictions on the API function usage.

ANNEX 2 – Linux PCI-Configuration Macros

Here are the Macros for accessing to the configuration block to set and read parameters

Macro	Width	Description
PCI_VENDOR_ID	16bit	Unique Vendor ID (see pci.h) This ID is defined by the PCI consortium
PCI_DEVICE_ID	16bit	Unique Device ID (see pci.h) this is defined by the vendor unique for each device
PCI_COMMAND	16bit	This field is used for device specific configuration commands (see table 2)
PCI_STATUS	16bit	This is used for board specific status results (see table 3)
PCI_CLASS_REVISION	32bit	The high 24bits are used to determine the device's class type the low 8bit for revision code
PCI_CLASS_DEVICE		The device's class code (see pci.h)
PCI_BIST	8bit	If PCI_BIST_CAPABLE is set the device can perform a Build-in self test that can be started by writing PCI_BIST_START to this field. The result mask is PCI_BIST_CODE_MASK
PCI_HEADER_TYPE	8bit	Specifies the Layout type for the following 48 bytes in PCI configuration (currently only 0x0)
PCI_LATENCY_TIMER	8bit	This is the maximal time a PCI cycle may consume (time=latency+8cycles)
PCI_CACHE_LINE_SIZE	8bit	Specifies the Cache-Line Size in units of 32bytes.
PCI_BASE_ADDRESS_[0-5]	32bit	This are up to 6 memory locations the device can map its memory area(s) to. The lower 4 bits are used to specify the type and the access mode of the memory area.
PCI_ROM_ADDRESS	32bit	bit 11-31 is the start address of the device's ROM area. (write bit 1 to enable ROM)
PCI_MIN_GNT	8bit	minimal latency time (vendor specific)
PCI_MIN_LAT	8bit	maximal latency time (vendor specific)
PCI_INTERRUPT_PIN	8bit	This entry denotes the IRQ pin that should be used 1=INTA 2=INTB 0=disabled
PCI_INTERRUPT_LINE	8bit	This entry specifies the interrupt line on which the device IRQ is mapped (usually IRQ 0-15)

Table 1: The Linux PCI-Configuration macros

To refer to the individual bits of the `PCI_COMMAND` and `PCI_STATUS` words, there are other Macros. For the bits of `PCI_COMMAND`, these are:

Macro	Description
PCI_COMMAND_IO	Enable I/O area
PCI_COMMAND_MEMORY	Enable Memory area
PCI_COMMAND_MASTER	Enable Busmastering
PCI_COMMAND_SPECIAL	Enable response to special cycles
PCI_COMMAND_INVALIDATE	Use memory write and invalidate
PCI_COMMAND_VGA_PALETTE	Enable video palette access
PCI_COMMAND_PARITY	Enable parity checking
PCI_COMMAND_WAIT	Enable address/data stepping
PCI_COMMAND_SERR	Enable SERR
PCI_COMMAND_FAST_BACK	Enable back-to-back writes

Table 2: The Linux PCI-Command Bit-Settings

And for the bits of `PCI_STATUS`, these are:

Macro	Description
PCI_STATUS_66MHZ	Support 66 Mhz PCI 2.1 bus
PCI_STATUS_UDF	Support User Definable Features
PCI_STATUS_FAST_BACK	Accept fast-back to back
PCI_STATUS_PARITY	Detected parity error
PCI_STATUS_DEVSEL_[MASK|FAST|MEDIUM|SLOW]	DEVSEL timing
PCI_STATUS_SIG_TARGET_ABORT	Set on target abort
PCI_STATUS_REC_TARGET_ABORT	Master ack of
PCI_STATUS_REC_MASTER_ABORT	Set on master abort
PCI_STATUS_SIG_SYSTEM_ERROR	Set when we drive SERR
PCI_STATUS_DETECTED_PARITY	Set on parity error

Table 3: The Linux PCI-Status Bit-Settings

ANNEX 3 – CANPCI: control and status registers

31 … 0	
OMB1	0h
OMB2	04h
INTSCR	38h
MCSR	3Ch

Interface Registers

BADR0

BADR1

31	30	29 … 7	6	5	4	3	2	1	0
X	FRONT	ON/OFF	XXX	CE7	CE6	CE5	CE4	CE3	CE2	CE1	CE0
X	R/W	R/W	XXX	W	W	W	W	W	W	W	W

Control and Status Register

BADR2

31 20	19	18		1	0
XXXXXXXXXXXXXXXX	C19	C18	C17 to C2	C1	C0
XXXXXXXXXXXXXXXX	W	W		W	W

Loading Register

BADR3 or BADR4

31 24	23	22		1	0
XXXXXXXXXXXXX	D23	D22	D21 to D2	D1	D0
XXXXXXXXXXXXX	R	R		R	R

RAM Register

ANNEX 4 – Listings

Here are the complete listings of the four files of the application plus the makefile special file for compiling.

common.h

```
#define COM_CHOOSE_FRONT            1
#define COM_START_ACQUISITION       2
#define COM_OUTPUT_DRIVE            3
#define COM_COUNTER_LOAD            4
#define COM_DATA_READ               5
#define COM_WAIT_ACQUISITION        6

// to see what the commands are

#define RES_CHOOSE_FRONT            1
#define RES_START_ACQUISITION       2
#define RES_OUTPUT_DRIVE            3
#define RES_COUNTER_LOAD            4
#define RES_DATA_READ               5
#define RES_WAIT_ACQUISITION        6

// results

unsigned long build_msg(int command, unsigned long argument);
```

```
// to build the message that must be passed to the FIFOs

int get_num(unsigned long msg);

// to get the command or response from the message

unsigned long get_arg(unsigned long msg);

// to get the argument of the command or response from the message
```

common.c

```
#include "common.h"

unsigned long build_msg(int command, unsigned long argument)

// to build the message that must be passed to the FIFOs
{
  unsigned long message;

  message = command << 24;
  argument &= 0x00FFFFFF;
  message |= argument;

  return message;
}

int get_num(unsigned long msg)

// to get the command or response from the message
{
  int num;

  msg >>= 24;
  msg &= 0x0000FFFF;
  num = msg;

  return num;
}

unsigned long get_arg(unsigned long msg)
```

```
// to get the argument of the command or response from the message
{
  unsigned long arg;

  msg &= 0x00FFFFFF;
  arg = msg;

  return arg;
}
```

canpci_ap.c

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <rtl_fifo.h>
#include <rtl_time.h>
#include <stdlib.h>
#include "common.c"

int getline(char line[], int max)
     // read strings of numbers from the user.It returns -1 if the
        string contains non-number characters
{
  int nch = 0;
  int c;
  int i;

  max = max -1;                                  // leave room for '\0'
  while((c = getchar()) != '\n')
    {
      if(nch < max)
     {
       line[nch] = c;
       nch = nch + 1;
     }
    }
  for (i = 0; i < nch; i++)
    if ((line[i] < 48) || (line[i] > 57)) return -1;
         // if the string is not made only of numbers
  line[nch] = '\0';
```

```
  return nch;
}

int wait_acq(int fd0, int fd1)
     // it tells us if the acquisition has finished
{
  int com = COM_WAIT_ACQUISITION;
  unsigned long arg_com = 0;
  unsigned long msg_com;
  unsigned long msg_res;
  unsigned long arg_res = 0;
  int res;
  int n;

  msg_com = build_msg(com,arg_com);
  if (write(fd0, &msg_com, sizeof(msg_com)) < 0)
     // write the command to FIFO_COM
    {
      fprintf(stderr, "Can't send a command to RT-task\n");
      exit(1);
    }
  printf("Process sending the \"%d\" command to task with the
  \"%ld\" argument.\n", com, arg_com);

  // WAIT FOR RESPONSE

  if ((n = read(fd1, &msg_res, sizeof(msg_res))) == sizeof(msg_res))
     // read the response from FIFO_RES
    {
      res = get_num(msg_res);
      arg_res = get_arg(msg_res);
      printf("Process received the \"%d\" result from task with the
      \"%ld\" argument.\n", res, arg_res);
    }
  return arg_res;
}

int main()
{
  int fd0, fd1;                    // file descriptors for the FIFOs
  int n;                           // used for reading
  int com = 0;                     // command to send
  int res = 0;                     // result received
  int choice;                      // option chosen in the menu
  char line[256];                  // buffer to read numbers with
                                      'getline' function
  int front;                       // type of front chosen
  unsigned char reg;               // for the output drive
  unsigned char output_drive = 0;  // for the output drive
  int i;                          // index in the output_drive option
  unsigned long samples = 0;       // acquisition size
  unsigned long arg_com = 0;       // command's argument
  unsigned long arg_res = 0;       // response's argument
  unsigned long msg_com;           // message that must be passed to
                                      FIFO_COM with the command
  unsigned long msg_res;           // message that must be received
```

```
                                        from FIFO_RES with the response
int get_line;                         // to see whether the return
                                        value of getline is -1 or not
int nosend;                           // used in COM_OUTPUT_DRIVE
FILE *fp;                             // file pointer to the file where
                                           the samples will be written in

if ((fd0 = open("/dev/rtf0", O_WRONLY)) < 0)
   // fd0 is FIFO_COM to send commands
{
  fprintf(stderr, "Error opening /dev/rtf0\n");
  exit(1);
}

if ((fd1 = open("/dev/rtf1", O_RDONLY)) < 0)
   // fd1 is FIFO_RES to recieve responses
{
  fprintf(stderr, "Error opening /dev/rtf1\n");
  exit(1);
}


// THE MENU

while (1)
  {
    printf("\nCommands to send to the CANPCI board : \n\n");
    printf("1- Choose front (UP/DOWN)\n");
    printf("2- Start acquisition\n");
    printf("3- Output drive (8 bits)\n");
    printf("4- Counter load (Up to 1M)\n");
    printf("5- Data read\n\n");
    printf("6- Exit program (not to send)\n\n");
    printf("Choose one : ");

    if ((get_line = getline(line, 256)) == -1)
   {
     choice = get_line;
   }
    else
   {
     choice = atoi(line);
   }
    printf("\nIt is : %d\n",choice);

    // OPTIONS

    switch (choice) {

   // CHOOSE FRONT

    case 1:
     com = COM_CHOOSE_FRONT;
     while (1)
       {
         printf("UP - 0 / DOWN - 1 : ");
```

```
        if ((get_line = getline(line, 256)) == -1)
        {
          front = get_line;
        }
        else
        {
          front = atoi(line);
        }
        if ((front == 0) || (front == 1 )) break;
        printf("Not a good value. Try again.\n");
      }
    arg_com = front;
       // give to 'arg_com' the value, 0 for UP and 1 for DOWN
    arg_com &= 0x00000001;
    printf("arg is : %ld\n",arg_com);
    printf("Option chosen is : %d\n",com);
    getchar();

    msg_com = build_msg(com,arg_com);
    if (write(fd0, &msg_com, sizeof(msg_com)) < 0)
       // write the command to FIFO_COM
      {
        fprintf(stderr, "Can't send a command to RT-task\n");
        exit(1);
      }
    printf("Process sending the \"%d\" command to task with the
    \"%ld\" argument.\n", com, arg_com);

    // WAIT FOR RESPONSE

    if ((n = read(fd1, &msg_res, sizeof(msg_res))) ==
    sizeof(msg_res))           // read the response from FIFO_RES
      {
        res = get_num(msg_res);
        arg_res = get_arg(msg_res);
        printf("Process received the \"%d\" result from task with
        the \"%ld\" argument.\n", res, arg_res);
        printf("FRONT bit has been set to %ld\n", arg_com);
        getchar();
      }
    break;

  // START ACQUISITION

   case 2:
    com = COM_START_ACQUISITION;
    arg_com = 0;

    msg_com = build_msg(com,arg_com);
    if (write(fd0, &msg_com, sizeof(msg_com)) < 0)
       // write the command to FIFO_COM
      {
        fprintf(stderr, "Can't send a command to RT-task\n");
        exit(1);
      }
    printf("Process sending the \"%d\" command to task with the
```

```
  \"%ld\" argument.\n", com, arg_com);

  // WAIT FOR RESPONSE

  if ((n = read(fd1, &msg_res, sizeof(msg_res))) ==
  sizeof(msg_res))           // read the response from FIFO_RES
    {
      res = get_num(msg_res);
      arg_res = get_arg(msg_res);
      printf("Process received the \"%d\" result from task with
      the \"%ld\" argument.\n", res, arg_res);
    }
  printf("Waiting...\n");
  while (wait_acq(fd0, fd1));         // wait until it finishes
  printf("Acquisition finshed.\n");
  getchar();
  break;

// OUTPUT DRIVE

 case 3:
  com = COM_OUTPUT_DRIVE;
  while (1)
    {
      nosend = 0;
      output_drive = 0;

      printf("Bits for the output drive (8) : ");
      i = 7;
      while((reg = getchar()) != '\n')
      {
       if (reg == '0') reg = 0;
       else
         {
           if (reg == '1') reg = 1;
           else
          {
            printf("\nBit %d is not valid. It must be 0 or
            1.\n", 8 - i);
            nosend = 1;
               // we don't want to send if it's not correct
          }
         }
       reg = (reg << i);
       printf("reg = %x\n",reg);
       output_drive = (output_drive | reg);
       printf("output_drive = %x\n",output_drive);
       arg_com = (output_drive & 0x000000FF);
       printf("arg = %lx\n", arg_com);
       i -= 1;
      }
      if (i != -1) {
        printf("\nNot valid. They must be 8 bits.\n");
        nosend = 1;
          // we don't want to send if it's not correct
        getchar();   // wait
```

```
        }
        printf("Bits are : %x\n",output_drive);
        if (!nosend) break;
      }
    printf("Option chosen is : %d\n",com);
    getchar();

    msg_com = build_msg(com,arg_com);
    if (write(fd0, &msg_com, sizeof(msg_com)) < 0)
       // write the command to FIFO_COM
      {
        fprintf(stderr, "Can't send a command to RT-task\n");
        exit(1);
      }
    printf("Process sending the \"%d\" command to task with the
    \"%ld\" argument.\n", com, arg_com);

    // WAIT FOR RESPONSE

    if ((n = read(fd1, &msg_res, sizeof(msg_res))) ==
    sizeof(msg_res))           // read the response from FIFO_RES
      {
        res = get_num(msg_res);
        arg_res = get_arg(msg_res);
        printf("Process received the \"%d\" result from task with
        the \"%ld\" argument.\n", res, arg_res);
        printf("Output bits has been set to %x\n", output_drive);
        getchar();
      }
       break;

  // COUNTER LOAD

   case 4:
    com = COM_COUNTER_LOAD;
    while (1)
      {
        printf("Acquisition size (Up to 1M) : ");
        if ((get_line = getline(line, 256)) == -1)
       {
         samples = get_line;
       }
        else
       {
         samples = atoi(line);
       }
        if ((samples > 0) && (samples <= 0x100000)) break;
             // good value
        printf("Not a good value. Try again.\n");
      }
    arg_com = 0x100000 - samples;
    printf("arg is : %ld\n",arg_com);
    printf("arg is : %lx\n",arg_com);
    printf("Option chosen is : %d\n",com);
    getchar();
```

```
    msg_com = build_msg(com,arg_com);
    if (write(fd0, &msg_com, sizeof(msg_com)) < 0)
       // write the command to FIFO_COM
      {
        fprintf(stderr, "Can't send a command to RT-task\n");
        exit(1);
      }
    printf("Process sending the \"%d\" command to task with the
    \"%ld\" argument.\n", com, arg_com);

    // WAIT FOR RESPONSE

    if ((n = read(fd1, &msg_res, sizeof(msg_res))) ==
    sizeof(msg_res))           // read the response from FIFO_RES
      {
        res = get_num(msg_res);
        arg_res = get_arg(msg_res);
        printf("Process received the \"%d\" result from task with
        the \"%ld\" argument.\n", res, arg_res);
        printf("The counter has been loaded to take %ld (decimal)
        %lx (hex) samples\n", samples, samples);
        getchar();
      }
    break;

  // DATA READ

   case 5:
    com = COM_DATA_READ;
    arg_com = 0;
    printf("Option chosen is : %d\n",com);
    getchar();

    fp = fopen("samples.dat", "w+");
    msg_com = build_msg(com,arg_com);
    for(i = 0; i < samples; i++)
      {
        if (write(fd0, &msg_com, sizeof(msg_com)) < 0)
            // write the command to FIFO_COM
       {
         fprintf(stderr, "Can't send a command to RT-task\n");
         exit(1);
       }
        printf("Process sending the \"%d\" command to task with
        the \"%ld\" argument.\n", com, arg_com);

        // WAIT FOR RESPONSE

        if ((n = read(fd1, &msg_res, sizeof(msg_res))) ==
        sizeof(msg_res))      // read the response from FIFO_RES
       {
         res = get_num(msg_res);
         arg_res = get_arg(msg_res);
         printf("Process received the \"%d\" result from task
         with the \"%ld\" argument.\n", res, arg_res);
         printf("The sample is : %lx\n", arg_res);
```

```
          }
           fprintf(fp, "%lx\n", arg_res);     // text data file mode
         }
       break;

     // EXIT

      case 6:
       printf("Exiting now...\n");
       return 0;
       break;

     // INVALID OPTION

      default:
       printf("Not a valid option. Choose again.\n");
       com = 0;
       arg_com = 0;
      }
    }
  return 0;
}
```

canpci_mod.c

```
#include <linux/errno.h>
#include <rtl.h>
#include <time.h>
#include <rtl_sched.h>
#include <rtl_fifo.h>
#include <linux/pci.h>
#include <asm/io.h>
#include "common.c"

#define FIFO_COM     0    // the FIFO used to send commands from the
                             Linux process to the FIFO handler
#define FIFO_RES     1    // the FIFO used to send responses from
                             the RT task to the Linux process
#define FIFO_RTTASK  2    // the FIFO used to send commands from the
                            FIFO handler to the RT task

#define VENDOR_ID_CANPCI 0x10e8      // used to initialize the board
#define DEVICE_ID_CANPCI 0x8110      // used to initialize the board

unsigned long canpci_ioaddr[5];  // contains the canpci IO addresses
struct pci_dev *dev = NULL;     // points to the configuration block

pthread_t mytask;

void *thread_code(void *t)
{
  int com;                      // command received
  int res;                      // result to send
  unsigned long msg_com;        // message received with the command
  unsigned long msg_res;        // message sent with the response
```

```
unsigned long arg;            // argument received
unsigned long reg;            // buffer for the registers

pthread_suspend_np(pthread_self());
   // it has to wait for a command to come in order to start

while (1) {
  int err;
  while ((err = rtf_get (FIFO_RTTASK, &msg_com, sizeof(msg_com)))
  == sizeof(msg_com))
  {
   com = get_num(msg_com);
   arg = get_arg(msg_com);
   rtl_printf("Task executing the \"%d\" command with the \"%x\"
   argument.\n", com, arg);

   // COMMANDS RECEIVED

   switch (com) {

      case COM_CHOOSE_FRONT:
       reg = inl(canpci_ioaddr[1]);
       if(arg)
         reg |= 0x40000000;     // if arg = 1 (DOWN) then the
                                   bit 30 (front bit) is set to 1
       else
         reg &= 0xBFFFFFFF;    // if arg = 0 (UP) then the bit 30
                                  (front bit) is set to 0
       outl(reg, canpci_ioaddr[1]);
       rtl_printf("canpci_ioaddr[1] = %lx\n", reg);

       res = RES_CHOOSE_FRONT;
       rtl_printf("Task sending the \"%d\" result.\n", res);
       msg_res = build_msg(res, 0);
       rtf_put(FIFO_RES, &msg_res, sizeof(msg_res));
          // send response
       break;

      case COM_START_ACQUISITION:
       reg = inl(canpci_ioaddr[1]);
       reg |= 0x20000000;
       outl(reg, canpci_ioaddr[1]);
       rtl_printf("canpci_ioaddr[1] = %lx\n", reg);

       res = RES_START_ACQUISITION;
       rtl_printf("Task sending the \"%d\" result.\n", res);
       msg_res = build_msg(res, 0);
       rtf_put(FIFO_RES, &msg_res, sizeof(msg_res));
          // send response
       break;

      case COM_WAIT_ACQUISITION:
       reg = inl(canpci_ioaddr[1]);
       reg &= 0x20000000;
       reg >>= 29;
```

```
        res = RES_WAIT_ACQUISITION;
        rtl_printf("Task sending the \"%d\" result.\n", res);
        msg_res = build_msg(res, reg);
           // 0 = acquisition finished, 1 = under acquisition
        rtf_put(FIFO_RES, &msg_res, sizeof(msg_res));
           // send response
        break;

      case COM_OUTPUT_DRIVE:
        reg = inl(canpci_ioaddr[1]);
        reg &= 0xFFFFFF00;
        reg |= arg;
        outl(reg, canpci_ioaddr[1]);
        rtl_printf("canpci_ioaddr[1] = %lx\n", reg);

        res = RES_OUTPUT_DRIVE;
        rtl_printf("Task sending the \"%d\" result.\n", res);
        msg_res = build_msg(res, 0);
        rtf_put(FIFO_RES, &msg_res, sizeof(msg_res));
           // send response
        break;

      case COM_COUNTER_LOAD:
        reg = inl(canpci_ioaddr[2]);
        reg &= 0xFFF00000;
        reg |=arg;
        outl(reg, canpci_ioaddr[2]);
        rtl_printf("canpci_ioaddr[2] = %lx\n", reg);

        res = RES_COUNTER_LOAD;
        rtl_printf("Task sending the \"%d\" result.\n", res);
        msg_res = build_msg(res, 0);
        rtf_put(FIFO_RES, &msg_res, sizeof(msg_res));
           // send response
        break;

      case COM_DATA_READ:
        reg = inl(canpci_ioaddr[3]);
        reg &= 0x00FFFFFF;

        res = RES_DATA_READ;
        rtl_printf("Task sending the \"%d\" result.\n", res);
        msg_res = build_msg(res, reg);
        rtf_put(FIFO_RES, &msg_res, sizeof(msg_res));
           // send response
        break;

      default:
        rtl_printf("RTL task: bad command\n");
        return 0;
    }
  }

  rtl_printf("RTL task waiting\n");
  pthread_suspend_np(pthread_self());
  rtl_printf("RTL task awakes\n");
```

```
  }
  return 0;
}

int my_handler(unsigned int fifo)
{
  int err;
  unsigned long msg;

  while ((err = rtf_get(FIFO_COM,&msg, sizeof(msg))) == sizeof(msg))
    {
      rtf_put (FIFO_RTTASK, &msg, sizeof(msg));
      rtl_printf("FIFO handler: sending the \"%d\" command to
       task.\n", get_num(msg));
      rtl_printf("FIFO handler wakes up with %d\n", get_num(msg));
      pthread_wakeup_np (mytask);
    }

  rtl_printf("FIFO handler is dead\n");
  return 0;
}

/* #define DEBUG */                    // if debug is to be done

int init_module(void)
{
  int c[4];                       // for debug
  int ret;                        // return value of pthread_create()
  u16 pci_command, new_command;  // initialize the board

  rtf_destroy(FIFO_COM);
  rtf_destroy(FIFO_RES);
  rtf_destroy(FIFO_RTTASK);

  c[0] = rtf_create(FIFO_COM, 100);
  c[1] = rtf_create(FIFO_RES, 100);
  c[2] = rtf_create(FIFO_RTTASK, 100);

#ifdef DEBUG
  printk("c[0] = %d\n", c[0]);
  printk("c[1] = %d\n", c[1]);
  printk("c[2] = %d\n", c[2]);
#endif

  // INITIALIZE THE BOARD

  // look for pci card

  if (!pcibios_present())
    {
      printk ("Not PCI bios present.\n");
      return -1;
    }

  dev=pci_find_device (VENDOR_ID_CANPCI, DEVICE_ID_CANPCI, dev);
```

```
  if (dev == NULL)
    {
      printk("No CANPCI card found.\n");
      return -1;
    }

  // This is to enable I/O area

  pci_read_config_word(dev, PCI_COMMAND, &pci_command);
  new_command = pci_command | PCI_COMMAND_IO;
  if (pci_command != new_command)
    {
      printk(KERN_INFO "  The PCI BIOS has not enabled this"
           " CANPCI!  Updating PCI command %4.4x->%4.4x.\n",
           pci_command, new_command);
      pci_write_config_word(dev, PCI_COMMAND, new_command);
    }

  canpci_ioaddr[0]= dev->base_address[0] & PCI_BASE_ADDRESS_IO_MASK;
  canpci_ioaddr[1]= dev->base_address[1] & PCI_BASE_ADDRESS_IO_MASK;
  canpci_ioaddr[2]= dev->base_address[2] & PCI_BASE_ADDRESS_IO_MASK;
  canpci_ioaddr[3]= dev->base_address[3] & PCI_BASE_ADDRESS_IO_MASK;
  canpci_ioaddr[4]= dev->base_address[4] & PCI_BASE_ADDRESS_IO_MASK;

  // THREAD

  ret = pthread_create (&mytask, NULL, thread_code, NULL);
  rtf_create_handler(FIFO_COM, &my_handler);
  return 0;
}

void cleanup_module(void)
{
#ifdef DEBUG
  printk("0 = %d\n", rtf_destroy(FIFO_COM));
  printk("1 = %d\n", rtf_destroy(FIFO_RES));
  printk("2 = %d\n", rtf_destroy(FIFO_RTTASK));

#else
  rtf_destroy(FIFO_COM);
  rtf_destroy(FIFO_RES);
  rtf_destroy(FIFO_RTTASK);

#endif
  pthread_cancel (mytask);
  pthread_join (mytask, NULL);
}
```

ANNEX 5 – RTLinux functions used in the application

Here are the man pages of the RTLinux functions used in the application module.

pthread_cancel

pthread_cancel - cancel execution of a thread

SYNOPSIS

#include <pthread.h>

int pthread_cancel(pthread_t thread);

DESCRIPTION

The pthread_cancel() function requests that thread be canceled. The target threads cancelability state and type determines when the cancellation takes effect. When the cancellation is acted on, the cancellation cleanup handlers for thread are called. When the last cancellation cleanup handler returns, the thread-specific data destructor functions are called for thread. When the last destructor function returns, thread is terminated.

The cancellation processing in the target thread runs asynchronously with respect to the calling thread returning from pthread_cancel().

RETURN VALUE

If successful, the pthread_cancel() function returns zero. Otherwise, an error number is returned to indicate the error.

ERRORS

The pthread_cancel() function may fail if:

[ESRCH]
No thread could be found corresponding to that specified by the given thread ID.

The pthread_cancel() function will not return an error code of [EINTR].

pthread_create

pthread_create -- create a thread

SYNOPSIS

#include <rtl_sched.h>

int pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);

DESCRIPTION

This function is an RTLinux version of standard POSIX threads function. pthread_create creates a realtime thread that will have attributes given by attr, and that begins executing function start_routine(arg). If the attribute is NULL, default attributes are used. The thread attributes in RTLinux are extended to allow thread creation on specific processors (pthread_attr_setcpu_np), to enable FPU operations in the created thread (pthread_attr_setfp_np).

RETURN VALUE

pthread_create returns 0 on success and a non-zero error code on error.

ERRORS

[EAGAIN]

Either not enough memory or PTHREAD_THREADS_MAX would be exceeded.

[EINVAL]

Bad attributes.

pthread_join

pthread_join - wait for thread termination

SYNOPSIS

#include <pthread.h>

int pthread_join(pthread_t thread, void **value_ptr);

DESCRIPTION

The pthread_join() function suspends execution of the calling thread until the target thread terminates, unless the target thread has already terminated. On return from a successful pthread_join() call with a non-NULL value_ptr argument, the value passed to pthread_exit() by the terminating thread is made available in the location referenced by value_ptr. When a pthread_join() returns successfully, the target thread has been terminated. The results of multiple simultaneous calls to pthread_join() specifying the same target thread are undefined. If the thread calling pthread_join() is canceled, then the target thread will not be detached.

It is unspecified whether a thread that has exited but remains unjoined counts against _POSIX_THREAD_THREADS_MAX.

RETURN VALUE

If successful, the pthread_join() function returns zero. Otherwise, an error number is returned to indicate the error.

ERRORS

The pthread_join() function will fail if:

[EINVAL]
The implementation has detected that the value specified by thread does not refer to a joinable thread.
[ESRCH]
No thread could be found corresponding to that specified by the given thread ID.

The pthread_join() function may fail if:

[EDEADLK]
A deadlock was detected or the value of thread specifies the calling thread.

The pthread_join() function will not return an error code of [EINTR].

pthread_self

pthread_self - get calling thread's ID

SYNOPSIS

#include <pthread.h>

pthread_t pthread_self(void);

DESCRIPTION

The pthread_self() function returns the thread ID of the calling thread.

RETURN VALUE

See DESCRIPTION above.

ERRORS

No errors are defined.

The pthread_self() function will not return an error code of [EINTR].

pthread_suspend_np

pthread_suspend_np -- suspend execution of a realtime thread.

SYNOPSIS

#include <rtl_sched.h>

int pthread_suspend_np(pthread_t thread);

DESCRIPTION

This function is a non-portable Realtime Linux extension. pthread_suspend_np suspends the execution of the thread thread until a call to pthread_wakeup_np. Note: If the target thread is running on a different processer, it is not guaranteed to be suspended immediately. Calls to pthread_suspend_np(pthread_self()) do work immediately.

RETURN VALUE

Always returns 0.

NOTES

pthread_suspend_np can be called from a signal or interrupt handler, but should be used with some care. RTLinux hard signal handlers (interrupt handlers) execute in the context of whatever thread was running when the interrupt was accepted by the hardware. If thread "A" was running when an interrupt handler started and the handler calls pthread_suspend_np, then the scheduler will save thread "A" and run another thread. If you call pthread_suspend_np from an interrupt handler running in the context of the general purpose operating system thread (Linux), then all general purpose operations and processes will stop until the thread is waked.

ERRORS

None.

pthread_wakeup_np

pthread_wakeup_np -- wake up a realtime thread.

SYNOPSIS

```
#include <rtl_sched.h>

int pthread_wakeup_np(pthread_t thread);
```

DESCRIPTION

This function is a non-portable Realtime Linux extension. pthread_wakeup_np wakes up the realtime thread thread.

RETURN VALUE

Always returns 0.

ERRORS

None.

NOTES

This function is safe to call from interrupt handlers and threads, but not from the Linux kernel. Calling the function will cause a thread switch if the awakened thread is higher priority than the current thread and they are both on the same processor. In this case, pthread_wakeup called from an interrupt handler will suspend the current thread at once. The handler will complete when there are no more runnable higher priority threads.

To illustrate, suppose we want to have very simple interrupt handlers (always a good idea), but we need complex processing done after each interrupt. We can designate a thread to do this processing and give it the highest priority. Then the interrupt handlers may push some data into a queue and call pthread_wakeup, to wake up the thread and switch to it at once. (WARNING: Do not leave any critical hardware handshaking until after the return from pthread_wakeup_np, unless it is okay for the handshaking to be postponed until the interrupted thread is scheduled again.)

rtf_create

rtf_create -- create a realtime fifo

SYNOPSIS

#include <rtl_fifo.h>

int rtf_create(unsigned int fifo, int size);

DESCRIPTION

rtf_create creates a realtime fifo (RT-FIFO) of size size and assigns it the identifier fifo. fifo is a value unique within the system, and must be less than RTF_NO.

The RT-FIFO is a mechanism, implemented as a character device, to communicate between realtime tasks and ordinary Linux processes. The rtf_* functions are used by the realtime tasks; Linux processes use standard character device access functions such as read(3), write(3), and select(3).

RETURN VALUE

On success, 0 is returned. On failure, a negative value is returned as described below.

ERRORS

[ENODEV]

fifo is greater than or equal to RTF_NO.

[EBUSY]

fifo is already in use. Choose a different ID.

[ENOMEM]

size bytes could not be allocated for the RT-FIFO.

NOTES

This function should only be used in the Linux init_module() context or in user space via PSC library (please see below). RT-FIFOs created in init_module() should be destroyed with rtf_destroy() in cleanup_module.

rtf_create is a system call made available by PSC, the user-level real-time signal library. It can be called from user space, but not from PSC handlers.

rtf_create_handler

rtf_create_handler -- install a handler for realtime fifo data

SYNOPSIS

#include <rtl_fifo.h>

int rtf_create_handler(unsigned int fifo, int (* handler)());

DESCRIPTION

rtf_create_handler installs a handler which is executed when data is written to or read from a real-time fifo (RT-FIFO). fifo is an RT-FIFO that must have previously been created with a call to rtf_create. handler is then called whenever a Linux process accesses that fifo.

rtf_create_handler is often used in conjunction with rtf_get to process data acquired asynchronously from a Linux process. The installed handler calls rtf_get when data is present. Because the handler is only executed when there is activity on the fifo, polling is not necessary.

The RT-FIFO is a mechanism, implemented as a character device, to communicate between realtime software components (either in threads or handlers), and between realtime software and ordinary Linux processes. The rtf_* functions are used in RT mode; Linux processes use standard character device access functions such as read(2), write(2) and select(2).

RETURN VALUE

On success, 0 is returned. On failure, a negative value is returned as described below.

ERRORS

[EINVAL]

fifo is greater than or equal to RTF_NO, or is not a valid RT-FIFO identifier; or handler is NULL.

rtf_destroy

rtf_destroy -- remove a realtime fifo created with rtf_create(3)

SYNOPSIS

#include <asm/rtl_fifo.h>

int rtf_destroy(unsigned int fifo);

DESCRIPTION

rtf_destroy removes a realtime fifo (RT-FIFO) previously created with rtf_create(3). fifo is then available for re-use by another call to rtf_create(3). All handlers for the FIFO are uninstalled.

The RT-FIFO is a mechanism, implemented as a character device, to communicate between realtime tasks and ordinary Linux processes. The rtf_* functions are used by the realtime

tasks; Linux processes use standard character device accesed by functions such as read(3), write(3) and select(3).

RETURN VALUE

On success, count is returned. On failure, a negative value is returned as described below.

ERRORS

[ENODEV]

fifo is greater than or equal to RTF_NO.

[EINVAL]

fifo is not a valid RT-FIFO identifier.

NOTES

This function should only be used in the Linux cleanup_module() context or in user space via PSC library (please see below).

rtf_create is a system call made available by PSC, the user-level real-time signal library. It can be called from user space, but not from PSC handlers.

rtf_get

rtf_get -- read data from a realtime fifo

SYNOPSIS

```
#include <rtl_fifo.h>

int rtf_get(unsigned int fifo, char * buf, int count);
```

DESCRIPTION

rtf_get reads a block of data from a realtime fifo (RT-FIFO) previously created with a call to rtf_create(3). fifo is the ID with which the RT-FIFO was created. buf is the block of data to be filled with the received bytes, while count is the size of the block in bytes. This mechanism is available only to realtime tasks; Linux processes use a read(2) from the corresponding fifo device to dequeue data from a fifo. Similarly, Linux processes use write(2) or similar functions to write the data to be read via rtf_put by a realtime task.

rtf_get is often used in conjunction with rtf_create_handler to process data received asynchronously from a Linux process. A handler is installed via rtf_create_handler; this handler calls rtf_get to receive any data present in the RT-FIFO as it becomes available. In this way, polling is not necessary; the handler is called only when data is present in the fifo.

The RT-FIFO is a mechanism, implemented as a character device, to communicate between realtime tasks and ordinary Linux processes. The rtf_* functions are used by the realtime tasks; Linux processes use standard character device access functions such as read(2), write(2) and select(2).

RETURN VALUE

On success, the size of the received data block is returned. Note that this value may be less then count if count bytes are not available in the fifo. On failure, a negative value is returned as described below.

ERRORS

[ENODEV]

fifo is greater than or equal to RTF_NO.

[EINVAL]

fifo is not a valid RT-FIFO identifier.

NOTES

rtf_get is available in PSC, the user-level real-time signal library. It can be called from handlers installed via rtlinux_sigaction(3).

rtf_put

rtf_put -- write data to a realtime fifo

SYNOPSIS

```
#include <rtl_fifo.h>

int rtf_put(unsigned int fifo, char * buf, int count);
```

DESCRIPTION

rtf_put writes a block of data to a realtime fifo (RT-FIFO) previously created with a call to rtf_create(3). fifo is the ID with which the RT-FIFO was created. buf is the block of data to be filled with the received bytes, while count is the size of the block in bytes. This mechanism is available only to realtime tasks; Linux processes use a write(2) to the corresponding fifo device to enqueue data to a fifo. Similarly, Linux processes use read(2) or similar functions to read the data previously written via rtf_put by a realtime task.

The RT-FIFO is a mechanism, implemented as a character device, to communicate between realtime tasks and ordinary Linux processes. The rtf_* functions are used by the realtime tasks; Linux processes use standard character device access functions such as read(2), write(2) and select(2).

RETURN VALUE

On success, count is returned. On failure, a negative value is returned as described below.

ERRORS

[ENODEV]

fifo is greater than or equal to RTF_NO.

[EINVAL]

fifo is not a valid RT-FIFO identifier.

[ENOSPC]

insufficient space is available in the RT-FIFO for count bytes.

NOTES

rtf_put is available in PSC, the user-level real-time signal library. It can be called from handlers installed via rtlinux_sigaction(3).

rtl_printf

rtl_printf -- print formatted output

SYNOPSIS

```
#include <rtl_printf.h>

int rtl_printf(const char *format, ...);
```

DESCRIPTION

This function is a non-portable RTLinux extension.

The rtl_printf function converts and formats its arguments similarly to printf(3) and places output to the kernel message buffer. Unlike the Linux kernel printk output facility, rtl_printf is safe to use from RTLinux threads and interrupt handlers.

RETURN VALUE

This function returns the number of bytes output on success, and a negative value on error.

ERRORS

No errors are defined.

NOTES

Generally, if at the Linux console, the kernel messages that rtl_printf produces are visible on the screen. You can also use the dmesg(1) command to view the kernel messages buffer. This is particularly useful when working in X. In addition, the syslogd(8) daemon is often configured to place the kernel messages to /var/log/messages.