15th Triennial World Congress, Barcelona, Spain

EXPLOITING A REAL-TIME LINUX PLATFORM IN CONTROLLING ROBOTIC MANIPULATORS

C. Bellini * **F. Panepinto** * **S. Panzieri** * **G. Ulivi** *

* *Dipartimento di Informatica e Automazione*
Università di Roma Tre
Via della Vasca Navale 79, 00146 Roma, Italy
{*bellini, panepi, panzieri, ulivi*}*@dia.uniroma3.it*

Abstract: Operating systems normally used for *office* applications are not suitable for real-time operations. As in all Unix systems, Linux scheduler is *preemptive* only at user lever, that means that an high priority user process, when ready to be executed, can suspend a lower priority user process but not a kernel one. *Not-preemptability* of kernel processes is the greater obstacle for the execution of real time tasks. Real Time Linux uses the Linux Operating Systes as a normal proces executed by a small real time operating system. The main goal of this work is to exploit a RealTime Linux platform in controlling a robot arm.

Keywords: Linux, Real-time, Control, Manipulators.

1. INTRODUCTION

To implement a control system on a Personal Computer there are, now, two main ways: the former is to resort to the "old but goldy" DOS "operating system" and to write anything from the beginning, including interrupt handlers and, perhaps, card drivers. This approach requires a good deal of experience and time but can result in a reliable, economic, and small footprint systems. What is most difficult, is to design systems capable of communicating with other ones, e.g. using network services.

On the other side, one can resort to a commercial real-time system that generally has all the provisions required to build a complex, interconnected control system. This is somehow the dual of the previous approach: one obtains a huge, expensive (often per unit royalties are to be paid) system, whose reliability is, at least, difficult to assess. One of the drawbacks of this way, often perceived as quite important, is that the code is masked and therefore the designer has to rely on it and on the documentation without being able to "open the box" and to look inside at the inner mechanisms.

In the last years a third possibility was opening up, in connection with the diffusion of open-source operating system, in particular in connection with the diffusion of Linux. There are several proposal to force this system to become or, at least, to behave as a Real Time one. Some of them are open source as the original operating system.

The purpose of this work is two-fold. Mainly it is devoted to the assessment of the feasibility of the implementation of the control system for a real 5-dof manipulator (Macchelli *et al.*, 2001) using RTLinux, a patch for kernels newer than 2.0.36. The other goal is to explore the possibility of using the parallel printer port to interface the PC with the external hardware of the manipulator. This idea is mainly related to research and didactic: any calculator provides such a port for free.

The paper is concluded by experimental results on the RT performance and on the control ones. Moreover a short discussion will show that the

parallel port is convenient only for very small projects.

2. OVERALL ARCHITECTURE

The control algorithm for the mechanical structure has been fully implemented on a personal computer. It has been therefore obtained a discrete-time control scheme where the sampling time can be set via software. Hence it is necessary that the operating system guarantees a careful process scheduling and that it is able to manage the timing with high precision.

Operating systems normally used for *office* applications are not suitable in this case. Our interest has been concentrated on GNU/Linux operating system that, compared with Microsoft systems, has the great advantage of being completely OpenSource and being based on the Unix system. As all Unix systems, Linux scheduler is *preemptive*, that means that an high priority process, when ready to be executed, can suspend a lower priority one. Most recent versions of Linux kernel, introduce the possibility to place side by side a static priority, definable from the user, and a dynamic priority, periodically calculated from the scheduler[1]. All normal processes have 0 as static priority therefore a process with a priority greater than zero will be favorite for the processor utilization. Kernel processes remain however excluded from the normal priority mechanism, they can always interrupt the other processes and temporary take the exclusive use of the processor, inhibiting the possibility to have a context change.

A scheduling algorithm like the one described gives good results in the management of normal activities but is not suitable for real time applications. To be able to guarantee sufficiently precise sampling times and to assure that the control related computations take a short time, it is necessary that a process is able to obtain the exclusive utilization of processor within a well-know time from the request.

2.1 *RealTime Linux*

To overcome the limitations due to the use of the Unix scheduler, more and more techniques are evolving to make Unix a system suitable to execute hard real time applications[2]. To improve the support to real time applications, Linux, as other Unix-based systems, conforms, in part, to POSIX.1b-1993 standard. This standard introduces a scheduler with user definable static priorities and the possibility to execute more than one thread in a single process. Usually, only one program counter is used to execute a block of instructions in a process; according to the POSIX standard it is possible to run more than one block or instructions side by side in the same process. Hence it is possible to design a cooperating threads architecture to optimize process resource handling.

unfortunately there are still some unsolved problems, as:

(1) *not-preemptability* of kernel processes
(2) low clock resolution
(3) high wait-time for IRQ response

Various techniques, based on that standard, have been developed to solve these problems, permitting to execute hard real time tasks in unix-like systems. One of these solutions, that has the characteristic of being completely free and OpenSource, is called *RealTime Linux*. The greater obstacle for the execution of real time tasks is the first listed point; kernel processes use, to disable the interrupts, specific processor instructions (e.g. `cli` and `sti` for Intel family processors). In RealTime Linux a software layer has been inserted between the request to disable interrupts and the effective call of `cli` and `sti`; this layer allows to prevent the interrupt of selected tasks from other processes (Barabanov *et al.*, 1997).

Regarding points 2 and 3, it has been possible to obtain a resolution for the IRQ response of approximately $15\mu s$ in the worse case, taking advantage of the built-in timer on Intel 8354 chip, present on all IBM compatible PC.

Through these tools, RTLinux provides some APIs that permit to build real time applications with performances suitable for our application.

2.1.1. *Structure of a RTLinux control application*

A typical control application in RTLinux environment is composed from a low level layer and a high level one.

The former consists of a kernel module where the real time threads run. These instructions are part of an infinite cycle; the time spent in order to execute an iteration of this cycle represents the sampling time applied to the structure. Among the several functions of the RTLinux APIs there are some that allow to regulate with great precision the iteration time, permitting to the designer

[1] In less recent versions, the scheduling algorithm, to optimize processor allocation, calculates only the priority of active processes with a regular period; in this case we speak of *dynamic priority*

[2] Two different Real Time applications can be defined: those that need more accurate sampling times are called *Hard Real Time* applications, those that, instead, do not need particularly stringent performances are called *Soft Real Time* applications.

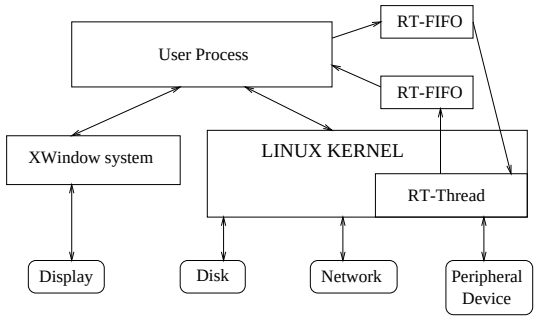

Fig. 1. Software architecture for a RTLinux application

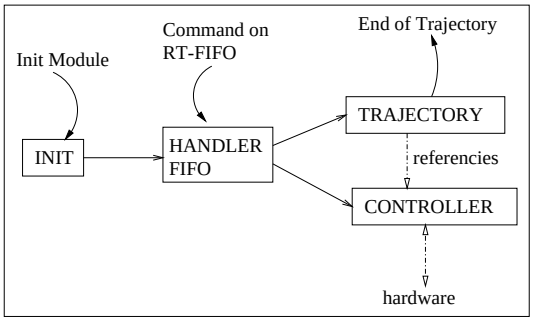

Fig. 2. Low level architecture

to choose the sampling time he prefers. It is important to remark that, during the wait-time between two sampling interval, the processor is free and it can, therefore, be used for other applications.

The high level part, instead, consists of user-interface applications that send to the low level part the START and STOP commands and the references. These applications can provide to the user with a set of services with different complexity. As all the operating system processes (e.g. user applications) become active only when real time threads are in a wait state, the designer has to consider which percentage of time is used for these operations and which is available for other processes. Taking care of this percentage, it is possible to determine the complexity of high level applications.

The figure 1 shows an outline of the software architecture for a typical RTLinux application.

To allow communications between user processes and real Time threads there are appropriate structures named RT-FIFO. These are seen from the kernel level as queues where it is possible to read or write blocks of characters by the typical operations `pop` and `push`. At the user level, instead, these are seen as characters devices (`/dev/rft*` with major number equal to 150 and minor number assigned when instantiated) where it is possible to read or write blocks of text by the standard library functions `write` and `read`. Since a RT-FIFO structure is monodirectional, in order to obtain a bidirectional data flow, it is necessary to instantiate two separate structures.

2.1.2. *Scheduler* Using OpenSource software, we have available every part of the system sources. In RTLinux it is possible to implement a scheduling algorithm, designed for a specific application, simply loading an appropriate kernel extension module (Aydin *et al.*, 1999).

In the released RTLinux version a very simple priority preemptive scheduler is provided: a priority is *statically* assigned to every process, when more then one task is ready, the one with the greatest priority is executed; if a task with a greater priority becomes ready it immediately interrupts the task in execution; moreover each task releases the CPU when the critical real time block is terminated.

This scheduler supports periodic applications, and is possible to execute isolated task defining an interrupt handler. Linux is, for this scheduler, the Real Time process with lower priority; working in this way the system is ready for other applications only when no Real Time thread is in execution.

2.2 *Threads architecture*

As reported in 2.1.1, our application is composed from a low level and a high level layers, where the position controller and the user interface are respectively implemented; the second one is used to display on the screen the state of controlled system.

Figure 2 shows the architecture of the kernel module where the position controller runs. RT-FIFO manager operates creation and destruction of Real Time threads receiving commands from the higher level. In this example a one joint position controller is implemented. There are two Real Time threads running simultaneously with two different tasks. The first one reads position data and runs the control algorithm, the other one provides references to controller, planning trajectory with trapezoidal joints speed profile. Obviously the controller priority is higher than trajectory planner one. On the contrary, the controller sampling time is shorter than planner one; due to this the trajectory reference is a sequence of steps.

A global architecture for the manipulator can be obtained from the one previously considered for one joint. The sequence of operation described below is executed in each control cycle:

(1) Encoder read
(2) Output of voltage values computed in the previous step
(3) Computation of control action

Position references are updated in real time from separated threads, one for each joint, running with an higher sampling time. Up to now, we

```
void * codice_cntr(void *arg)
{
  unsigned short int v;
  unsigned short int Vref;

  pthread_suspend_np(pthread_self());
  while(1) {
    pthread_wait_np();
    outb(inb(CONTROL) & 251,CONTROL);
    outb(1,ADDRESS);                  /* Addres
    v = inb(DATA);                    /* Data R
    check_status(BASE);
    Vref = P_PI_indipendente(v);      /* contro
    outb(inb(CONTROL) | 4,CONTROL);
    outb(1,ADDRESS);                  /* Addres
    outb(Vref,DATA);                  /* Data W
  }
}
```

Fig. 3. C code for the critical part of the control thread

execute only a simple trajectory planner version that could receive data from a more complex algorithm running at an higher level. In the complete version, the control module is split in six Real Time threads running at the same time: five threads produce trajectories and one implement the position controller; the last one runs with greater priority and higher sampling frequency than all the others. We observed that, working in this way, sometimes all threads are recalled at the same time, and the sampling time obtained for position controller thread is afflicted by a small delay because the scheduler has to handle this situation. Starting any thread at a different times improves sampling accuracy.

2.2.1. *Motor control and Trajectory generation* A great advantage, obtained by reading encoder measures and closing the feedback loop directly on the computer, is the possibility to easily modify the control algorithm.

In figure 3 the C code implementing the critical part of the control thread is reported. The control routine is invoked after parallel port input commands. Hence it is possible to test new control algorithms just modifying this routine with no needs to modify the critical part of the code.

We decided to implement a very simple joint independent control algorithm with speed and position feedback loop. A proportional controller is used in the position loop, whereas in the speed loop a controller with proportional and integral actions is used. Simulating the global system we obtained a first choice for control parameters, then tuned on the real system (Büskens *et al.*, 2000).

2.2.2. *Publishing* The only values of interest for high level processes are the angular positions of the joints. The utilization of the RT-FIFO structures to buffer position values could be a good choice to keep the kernel module separated from user applications but, unfortunately, due to delay times for file access, this is not possible.

Hence, a kernel module has been developed and added to RTLinux to instantiate memory areas accessible both from kernel module and user processes. This could be dangerous for data corruption and it is very important to be careful during data access, but it is very handy and can be used for a monitor process. At the kernel level, the area of memory used to store position values must be shared, so user processes could read those data and display the angular position values reached from each joint. In this way the measurement does not introduce errors on the values, in fact position value is updated from real time module but it is read and displayed from user process that run slower.

This structure introduces remarkable advantages if compared with the trivial solution of printing on video at kernel level, using for example the function `rtl_printf`. The great difference is that, in this last way, the real Time process holds the processor during all the duration of the print, in the other one, instead, the printing process is a normal user process that can be anytime interrupted from real Time processes.

Hence, using shared memory, it is possible to realize a complex graphical man-machine interface; for example it is possible to create a graphical representation of the manipulator position in the work space or a client-server network: when a PC on the network asks to know the position of the manipulator, the control computer reads data on the shared memory and sends them on the network. It has to be remarked that the publishing process, as all user processes, does not hinder the control system that can interrupt whichever other process and can take CPU anytime it is necessary.

2.3 *Parallel port access*

A bidirectional communication between the PC and the external interface is provided through a parallel port: the PC outputs voltage references and receives encoders readings as inputs. We performed bidirectional communication using EPP mode as defined in the IEEE 1284 standard. In this standard, the introduction of an handshake, permits to perform a safe and efficient bidirectional data exchange, using control signals.

In Linux, the access to the memory area where the parallel port is mapped, is implemented by means of the macro `outb` and `inb`. These macro operate an efficient data transfer from one memory location to another one. Obviously, in order to operate data transfer, it is necessary that the process owns the rights for memory access on those areas. This fact does not represent a problem because the access to the parallel port map is made directly

from the real Time control thread, which is a kernel process and has full access to all the system resources.

3. EXTERNAL HARDWARE

3.1 *Parallel port interface*

The lines of the parallel port are mapped on three memory registers: `Base`, `Status` and `Control`. In the `Base` register, the 8-bits used for data are stored, the read only register `Status` is used for external signals received from the hardware, while the `Control` register is used to output control data.

In order to operate a valid data transfer in EPP mode, it is necessary to implement a simple handshake cycle. If the PC BIOS is properly set, such cycle is correctly managed from the motherboard hardware and is activated when a read or write operation is executed on the memory area where the parallel port is mapped. To realize external hardware EPP compatibility we have designed two finite state machines to set and reset control signals when demanded. If the handshake cycle does not receive an acknowledgement from the external hardware within a time of approximately $10\mu s$ the cycle is terminated and a timeout is set on the `Status` registry.

3.2 *External interface design*

External interface board has to perform three task:

(1) Analog voltage references generation.
(2) Absolute position measure by processing encoder signals
(3) Parallel port comunication handling

The figure 4 shows the external interface block diagram. Five independent blocks called `SENC1..5` count the encoder signal hedges. Data exchange is provided by a bidirectional bus that is directly connected with the parallel port `Base` register. Logic components of the boards are connected to the bus through a *3-state* buffer that permits to temporary isolate some components.

In the logic scheme it is possible to see two Finite State Machine (FSM); those receive as input 4 bit of the parallel port `Control` register and output control signals for *3-state* buffer, DAC control signals and parallel port acknowledgement. A FSM handles input section and the other one manages output section; when a FSM is running the other one is waiting in the initial state. State transitions are determined by control register values; hence software access to parallel port has to accomplish

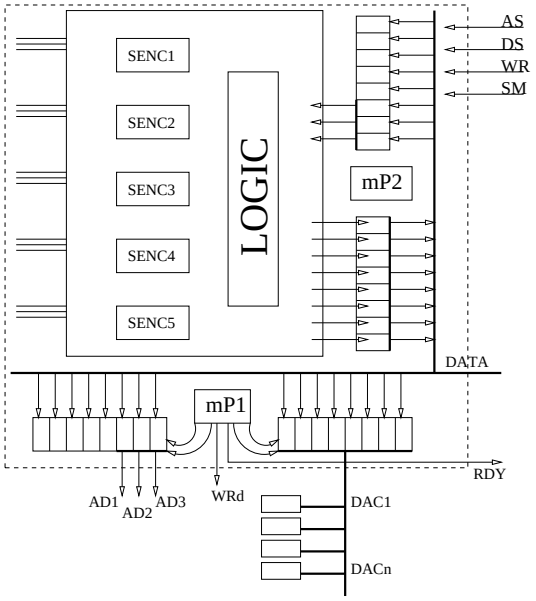

Fig. 4. External Interface Logic Scheme

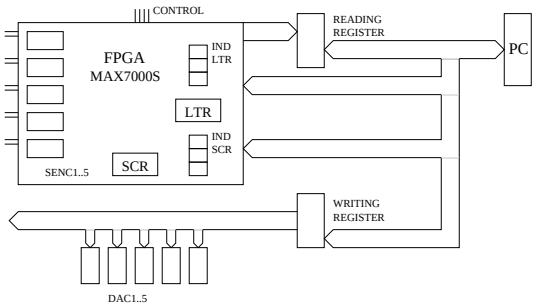

Fig. 5. External interface global architecture

a precise sequence of operation established during FSM design. Both FSMs are implemented on a programmable component, as an FPGA (Field Programmable Gate Array), hence it has been possible to modify theirs structure more times to obtain the right behaviour. For this reason, writing a C code for interface a device driver was not difficult thanks to the possibility to modify both hardware and software at the same time.

In both FSMs we can find a data buffer and an address buffer; during a write operation, the address, stored in the address buffer, indicates which DAC has to work, while, during a read operation, it indicates which encoder data we need to read.

Figure 5 shows the external interface global architecture; its logic scheme is represented in figure 4. Counters and FSMs have been designed in VHDL (Very high speed integrated circuit Harware Description Language) and implemented in the FPGA. Data buffers, used to improve signal synchronization on the circuit, are separated integrated circuit, while address buffers are implemented in the FPGA with 3-bit registers.

Absolute position is provided counting rising and falling hedges of both A and B channel of each incremental encoder. A synchronous circuit is

implemented on the FPGA to determine direction every time an hedge is detected on a channel.

3.3 *Digital to Analog conversion*

Typical sampling time for this application are about some milliseconds so we did not need to use high performance DAC. We choice the low cost and easy to find 7254, that has current outputs and is easy to connect to a microprocessor. Using two operational amplifiers it provides an output range between -5 and +5 volts.

4. CONCLUSIONS AND RESULT REPORT

4.1 *Hardware*

Thanks to several tests, we determined that a parallel port read or write access takes about $1.5\mu s$, so it is possible to make a great number of accesses to this port in a sampling time that is usually of some milliseconds in control applications. Moreover, using EPP mode, we obtained safe bidirectional communications, designing the external interface to accomplish EPP mode. The prototypal EPP interface board, was used to control one joint of a manipulator, with good results.

The cost of an experimental 5 axes card, as estimated after its design, would be comparable to that of a commercial card, with, possibly a worse reliability. We concluded that, for simple applications (one or maximum two axes) the EPP parallel port is a viable and interesting opportunity, whilst in more complex cases, the external hardware became too cumbersome and expensive.

4.2 *Software*

When using a Real Time Operating System, it is very important to measure the accuracy of timing. We decided to make these measures via software building an assembler macro to read clock cycles every time an interruption is called. Hence, it was possible to obtain high accuracy measures for sampling time and thread length, without introducing disturbances in the system.

Fig. 6 reports sampling time distribution obtained in two different conditions; in the first one, only the control threads are running on the computer, in the second one there are two programs performing complex mathematical tasks. As we can see, accurancy on sampling time is very high in both situations; as expected, covariance in the second one in higher than in the first one. Note that the introduction of a relative offset in the starting times of the different tasks is of paramount importance to obtain good results.

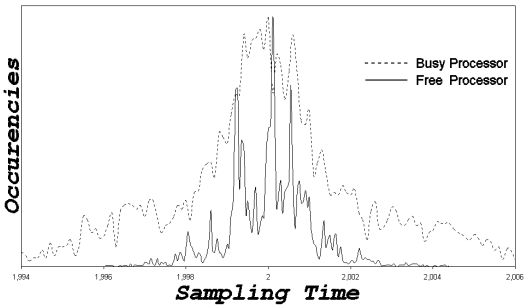

Fig. 6. Sampling time distribution with free and busy processor (properly scaled)

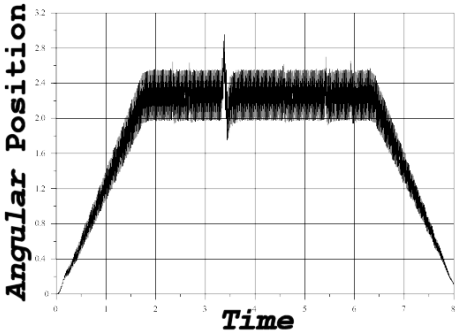

Fig. 7. Position error with trapezoidal velocity reference

4.3 *Control algorithm*

The control algorithm has been tested on one joint using the structure of figure 3. The errors have been measured using the same encoder used for control. The maximum dynamic error is about 0.2 degrees(fig.7).

The ripple on the error is mainly due to the stepped shape of the reference trajectory. Currently we are working on a low level micro-interpolator to get rid of it.

5. REFERENCES

H. Aydin, R. Melhem, D. Mossé, P. Mejìa Alvarez(1999). Optimal reward-based scheduling for periodic real-time tasks. *Real-Time Systems Symposium (RTSS'99),*

C. Büskens, H. Maurer(2000). Nonlinear programming methods for real-time control of an industrial robot. *Jurnal of optimizations theory and applications,* Vol. 107, no. 3, pp. 505-527.

A. Macchelli, C. Melchiorri, D. Pescoller(2001). An experimental set-up for robotics and constrol system research using Real-Time Linux and Comau SMART 3-S robot. *Real-Time Linux workshop 2001,*

M. Barabanov(1997). A Linux-Based Real-Time operating system. *Master thesis* , New Mexico Inst. of Mining and Technologies, Socorro, New Mexico