SOME EXPERIENCES IN FAST HARD-REAL TIME CONTROL IN USER SPACE WITH RTAI-LXRT

E. Bianchi, L. Dozio
Dip. Ing. Aerospaziale, Politecnico di Milano,
via la Masa 34, 20158 Milano, Italy
bianchi@aero.polimi.it
dozio@aero.polimi.it

Abstract

This paper demonstrates the possibility of executing wide band hard real time application in user space under Linux. Such a possibility is demonstrated through examples of its implementation at DIAPM of three digital active control systems, that require relatively high (up to 12 KHz) control frequencies: an active noise control of an acoustic duct using feedforward techniques, an application of active vibration suppression of a plate with piezoelectric materials, and an active noise reduction system used to control combustion instabilities in a Rijke tube. A brief introduction of the basic ideas on which hard real time control in user space is based, its most recent developments, and a complete description of the software architecture adopted in each application are included.

1 Introduction

Since the birth of RTAI almost two years ago, all the experimental researches on active noise, vibration and flutter control at Dipartimento di Ingegneria Aerospaziale, Politecnico di Milano (DIAPM)) have been implemented using it [1]-[2]. While maintaining its performances, RTAI has evolved consistently, especially in adding new features [3], such as shared memory, newfifos, mailboxes, etc., following its main author's "philosophy" that "if the user has a feature, probably he/she will not need it; if the user doesn't have a feature, surely he/she will need it".

Many undergraduate students and PhD candidates have been involved in these control activities for their thesis. Furthermore in the future RTAI will be used also for teaching a course of aerospace control systems, confirming its role of main real-time platform at DIAPM. But generally, people who used, uses or will use it, have no much experience in real-time systems and kernel-modules programming, so this approach to RTAI is often hard and slows down significantly the development of its control application.

The primary reason of the birth of RTAI-LXRT was an easier, less risky and faster development of real-time applications in user space, making available any of the RTAI schedulers functions to Linux processes and allowing a full symmetric implementation of real time services. In such a way students familiar with C-programming, after a brief tutorial on multi-tasking, timing and IPC communications, can quickly become good soft-real-time programmers, bypassing the shock of many hard machine crashes while working in kernel space.
LXRT eventually evolved to the point of making it possible full hard real-time in user space. In this way, users can not only develop and test their final applications in user space, but also run them without going into the kernel. It is believed that HARD-LXRT, introducing only a few microseconds more overhead, is acceptable for many applications, as the results presented further on demonstrate.

2 LXRT (and mini ...)

As already suggested, LXRT is a module that allows the symmetric use of all the services made available by RTAI and its schedulers in user space, both for soft and hard real time. That implies you can call all the RTAI functions in whatever space you are, i.e. inter-intra kernel real time tasks and Linux processes. Thus RTAI can become an alternative way also for programming standard user space applications, exploiting both Linux and RTAI services. Hence you can freely use RTAI services, 156 functions, including flexible timings, semaphores, mail-

boxes, inter-task messages and remote procedure calls, in whatever space and application you want to work.

A definite advantage given by LXRT is that one can start developing his/her application safely in user space, test it in soft real time, eventually go to hard real time through a simple call a function and, only if required, to kernel space for top performances.

Hard real time LXRT in user space allows full kernel preemption and the only penalty you pay is a slightly increased overhead, jitter and latency remaining very close to the kernel ones. All of it comes without (almost) touching the kernel, just 4 lines of code. For sure it is a far easier approach than having a herd of fancy preemption points scattered around the kernel code.

There is however a constraint that must be satisfied to implement hard real time in user space: you cannot use Linux kernel services. It is not a heavy burden as, thanks to the wealth of RTAI inter task communication services, it is trivial to mate each hard real time process to a Linux server that takes up all the kernels services on behalf of its hard real time master. Such a policy could be the right one to be chosen also when one is working with any native hard real time operating system available on the market. Note that, having done it independently from the kernel, if, and when, plain Linux will ever reach hard LXRT performances, that will imply just that LXRT developers will be relieved from the burden of maintaining it. Any existing RTAI application in user space will be untouched and RTAI will still remain a valuable tool to make your work easier.

LXRT has been wholly developed at DIAPM by Paolo Mantegazza (`mantegazza@aero.polimi.it`). However its hard real time support is jointly copyrighted with Pierre Cloutier (`pcloutier@poseidoncontrols.com`) and Steve Papacharalambous (`stevep@lineo.com`), as they played a substantial part in making it work his hard real time initial draft code. It should be remarked that Pierre Cloutier has also implemented, and is still improving, a more robust and user friendly version of LXRT, he called LXRT-INFORMED. It will eventually become a full substitute of LXRT. LXRT will nonetheless survive as the base initial development tool for implementing new features. Moreover in the near future RTAI and LXRT hard real time operations will benefit also from having full traps protection. The related work is being jointly carried out by Pierre Cloutier and Ian Soanes(`ians@lineo.com`).

To access RTAI services, Linux processes, using LXRT, create a real time task (i.e. the buddy) with `rt_task_init()`. The buddy's job is to execute the real time services on behalf of its parent process. Afterward you can start a timer (`start_rt_timer()`), mark the process as periodic with `rt_task_make_periodic()`, sleep for a while with `rt_sleep()`, wait on a semaphore (`rt_sem_wait()`), and so on. To delete the buddy, just call `rt_task_delete()`.

To distinguish a hard real time process from a LXRT firm real time process, the user simply calls `rt_make_hard_real_time()`, whereas by using `rt_make_soft_real_time()` he/she can return to standard Linux task switching. The soft Linux interrupts are kept disabled for hard real time user space processes. This way, hard real time tasks and interrupts can preempt user space processes, but they cannot be preempted neither by Linux interrupt nor by Linux processes, while they can be preempted by real time task in kernel space and hard real time processes of high priority.

The reader is invited to have a look at the RTAI documentation and manual, as well as to the wealth of LXRT examples, found in the RTAI distribution, for more detailed information and for a check of its performances.

The new development version of RTAI (24.1.xx), aimed at the approaching 2.4.xx kernel, contains also the `mini_rtai_lxrt` tasklets module, which adds an interesting new feature along the line of a symmetric usage of all its services inter-intra kernel and user space, both for soft and hard real time applications. In such a way one has an even wider spectrum of development and implementation lanes, allowing maximum flexibility with uncompromized performances.

The new services provided by `mini_rtai_lxrt` can be useful when you have many tasks, both in kernel and user space, that must be executed in soft/hard real time, but do not need any RTAI scheduler service that could lead to a task block. Such tasks are called tasklets and can be of two kinds: normal tasklets and timed tasklets (timers). Tasklets should be used whenever the standard hard real time tasks available with RTAI and LXRT schedulers can be a waist of resources and the execution of simple, possibly timed, functions can be more than enough. Instances of such applications are timed polling and simple Programmable Logic Controllers (PLC) like sequences of services. Obviously there are many other instances that can make it sufficient the use of tasklets, either normal or timers. In general such an approach can be a very useful complement to fully featured tasks in controlling complex machines and systems, both for basic and support services.

Timed tasklets executes their function either in oneshot or periodic mode, on the base of their time deadline and according to their, user assigned, priority. Instead plain tasklets are just functions whose

execution is simply triggered by calling a given service function at due time, either from a kernel task or interrupt handler requiring, or in charge of, their execution when they are needed. Since only non blocking RTAI schedulers services can be used in any tasklet functions, user and kernel space `mini_rtai_lxrt` applications can cooperate and synchronize by using shared memory. Note that the very name `mini_rtai_lxrt` remind to a kind of light soft/hard real time server that can partially substitute RTAI and LXRT in simple non blocking applications.
To initialize in kernel space a timed tasklet to be used in user space, you have to call `rt_init_timer()`, while `rt_insert_timer()` inserts it in the list of timers to be processed by a timers manager task. For a normal tasklet, the corresponding functions are `rt_init_tasklet()` and `rt_insert_tasklet`, while for its execution just call `rt_tasklet_exec()`.
Timed tasklets are fired under the control of a server kernel space task, while the agent in charge of executing normal tasklets finds and executes them by means of an agreed name.

3 Software architecture

As already said, LXRT makes available all the services and programming mechanisms of RTAI, and its schedulers, in user space. Thank to them you are allowed maximum freedom in implementing whatever hard real time policy is most suited to solve your problems. Without entering in many details we now explain a few typical simple schemes that can be used to implement a single controller with a user interface and a Linux server.
The schemes presented here are all based on a `main` program that expands into two threads of execution and then acts as a user interface. The first thread, called `fun`, will run in hard real time and executes the controller task. The second thread, called `linux_server`, acts as server toward Linux and its services. Both `main` and the `linux_server` are often soft POSIX real time tasks implementing a `SCHED_FIFO` policy. Such an implementation is by no means compulsory but is usually adopted to achieve a better response from Linux. Both `main` and `linux_server` usually cooperate in building up a suitable user interface. By way of example `main` can take the burden of user input, by controlling the mouse, keyboard and network communications, while the `linux_server` can provide data logging to any file system and monitoring scopes.
The terms thread above is taken in a loose sense and can mean either POSIX threads, created by calling `pthread_create` as in the presented examples, or standard LINUX processes created by forking and/or direct commands, or any combination thereof. Clearly the use of true threads allows an easy sharing of a common data space, so it is much more similar to the use of kernel space module. Such a solution could ease the transition to kernel space, if that will ever be required for maximum performances. However thank to the inter-task communication/synchronization services and shared memory available in RTAI the use of full processes makes such a final pass as easy as that based on threads. Any possible following porting to kernel space will just translate in the use of communicating modules. Once more it is important to remark that such a scheme can be expanded endlessly to allow you the more appropriate and easy way to implement your control system.

Naturally all threads/processes needing to access RTAI services must mate with a buddy RTAI task server. After having acquired a buddy they can run either in soft real time, relying on a `SCHED_FIFO` Linux scheduling policy (by calling `sched_setscheduler()`), or in hard real time, by calling `rt_make_hard_real_time()` to access the services of the fully preemptive LXRT scheduler. Hard real time in user space make it compulsory the use of `mlockall()` to avoid paging and related memory faults. Soft real time tasks do not strictly require memory locking, however it is strongly recommended to use it anyhow.

Once more the reader is strongly advised to have a look at the example available in LXRT and LXRT-INFORMED for a more detailed presentation of the wealth of usage of RTAI services in user space, for both soft and hard real time. So for sake of simplicity in the following part we will present just some solutions, more appropriate to the implementation of the controllers used in this paper. Before going on we note that the controller (`fun`) must be timed precisely to achieve a precise sampling rate. Such a timing can be provided either by the schedulers timing or by an external source. In our specific cases `main` acted as a simple user interface standard Linux Process, using mouse and keyboard, `linux_server` acted as a soft POSIX real time monitor process, logging data to disk for post processing and implementing a simple tcl/tk based scope, `fun` was the actual hard real time controller, interfacing to sensor and actuators by using AD/DA ISA boards, programmed directly in user space.

3.1 Task and tasklets directly driven by a scheduler timer

Having chosen to rely on RTAI schedulers timing, Figs. 1 and 2 sketch two possible solutions. In Fig. 1 the use of periodic task is exemplified, while a simple timed tasklet is adopted in Fig. 2. The sketched frame of operation should be self explaining. The two solutions are very similar, the notable difference being that the timer tasklet (`fun`) can interact with the other components of the application only through the common data space, as it is forbidden to use any RTAI service that can lead to a task switch. So such a solution can simply execute a timed function under the control of the RTAI TIMERS MANAGER TASK. The timers manager is the kernel task in charge of calling `fun`, either in kernel or user space according to its type, i.e. periodic or oneshot, and user assigned priority. The choice between the two solutions depend on what `fun` has to do. If it is sufficient a timed tasklet function, it is the simplest solution.

3.2 Task and tasklets directly driven by an external timer

If an independent external timing source is available, as it often is when a DA/DA board is being used, it is possible to avoid using the scheduler timing and rely on an external timer only. This was clearly possible in our case so that the solutions depicted in Figs. 3 and 4 were also possible. Fig. 3 still uses the RTAI schedulers. The external timer interrupt drives `timer`, its kernel space handler, who resumes the hard real time task `fun`. In our case the use of a counting semaphore is preferred but a simple suspend/resume scheme could also be suitable. It must be noted however that the use of a counting semaphore allows an easy check of timer overruns thus permitting a safe halt of the control system prior to jamming the computer. In fact RTAI `rt_sem_wait` returns the semaphore count and so timer overruns are promptly detected. The timer can be controlled both by `main` and `fun` in any preferred way. The tasklet solution is much the same as the previous timed tasklet execution. The notable difference is that a timed tasklet is executed by A TIMERS MANAGER while a plain tasklet can be executed from whatever RTAI kernel space function, i.e. tasks and interrupt handlers, the latter being our case.

3.3 Some specific comments

As summarized in paragraph 4 for the tests at hand no noticeable difference in performance was evidenced, the emphasis being mainly placed on the possibility of executing in user space high rate hard real time controllers, carrying out a relatively large amount of floating point calculations, with reasonable user interfaces. The fact that such a possibility allows also an easy development needs not to be commented any more. However it should be noted that in the present case the exemplified applications were already available in kernel space. So the implementation path went opposite to the usual implementation, i.e. soft user space, hard user space, kernel space (if needed). Nonetheless the possibility of developing first in soft real time was fully appreciated and, when every thing proved fine, `rt_make_hard_real_time` was simply uncommented thus easily getting into hard real time performances.

4 How we tested

All the experimental activities, described in the next paragraphs, were carried out on a Uni-Processor Intel Pentium III 700 MHz, using a Das1600 board for analog inputs and a PCL727 board for analog outputs. While testing our hard real time applications in User Space a sustained high processing load was always generated by simultaneously running the following commands:

```
1 - ping -f somewhere1
2 - ping -f somewhere2
3 - while "true"; do
ls -aR /; sync; done
4 - while "true"; do
cp /usr/src/linux-2.2.16.tgz tmp;
sync; rm tmp; sync; done
5 - top -d 0.05
6 - while "true"; do
cat /proc/interrupts;
cat /proc/rtai/*; done
```

Such a load caused top to constantly show a 100% CPU usage and allowed to verify that hard real time in user space was not affected by any glitches related to any foreground load.
Since all hard real time tasks were running periodically, there scheduling jitter/latency was continuously monitored by:
- toggling a bit on the parallel port at each task period, and displaying the related square wave on a persistent trace of a digital scope;
- internally computing the actual period by using the CPU Time Stamp Clock (TSC).
The thickened scope trace allowed to directly measure the related "true" jitters. All the scope and internal measurements were always within 1-2 μs at

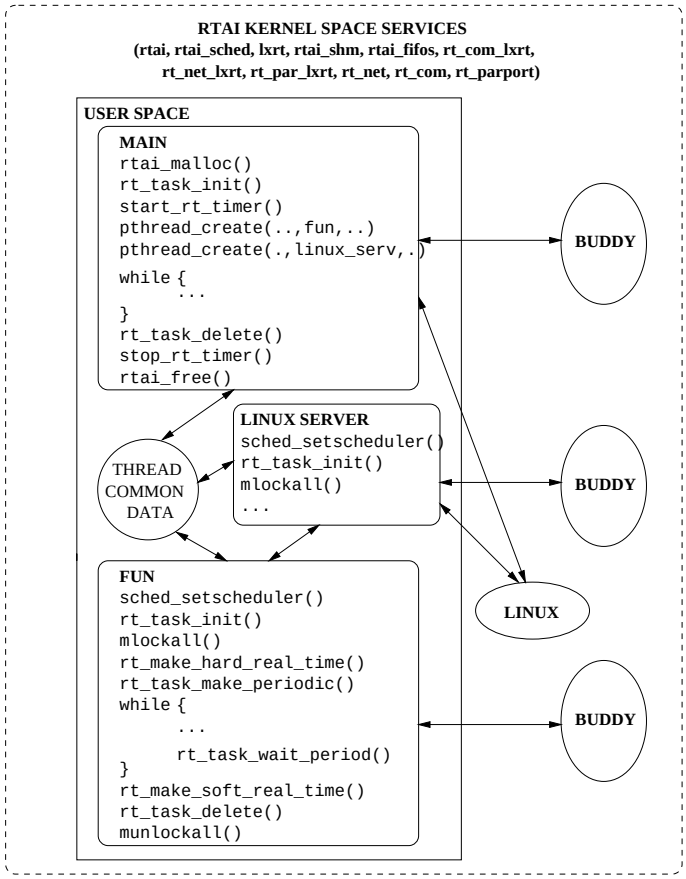

Figure 1: Typical use of a fully featured user space hard real time process with a scheduler timer

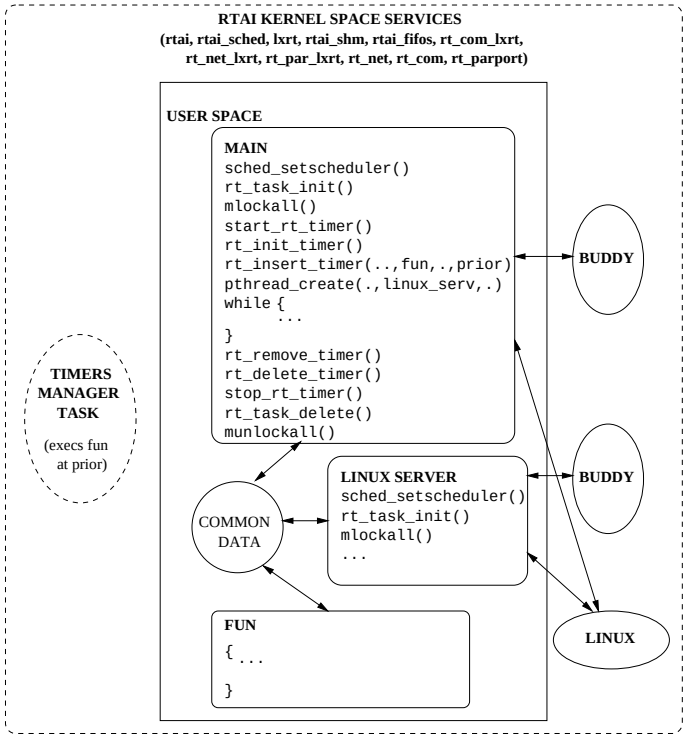

Figure 2: Typical use of a hard timed tasklet (timer) in user space

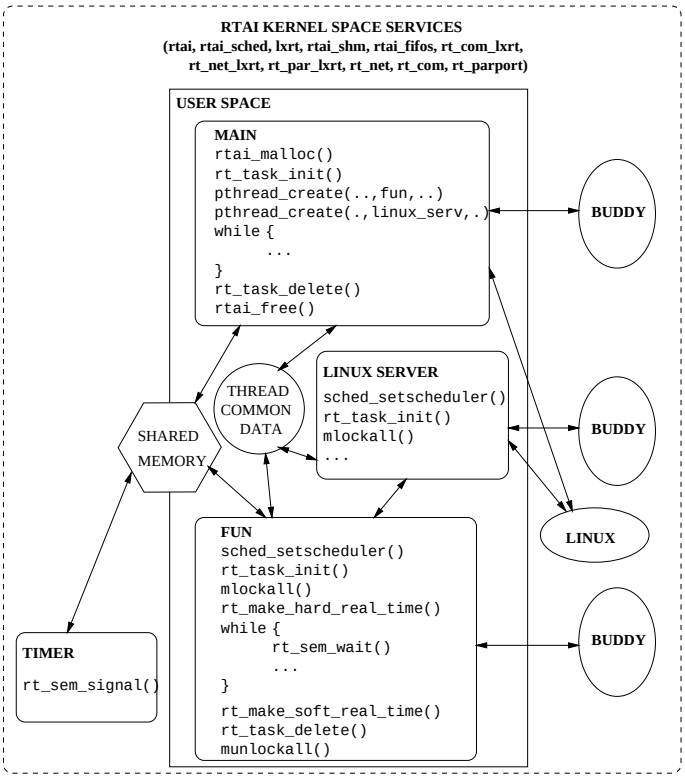

Figure 3: Typical use of a fully featured user space hard real time process with an external timer

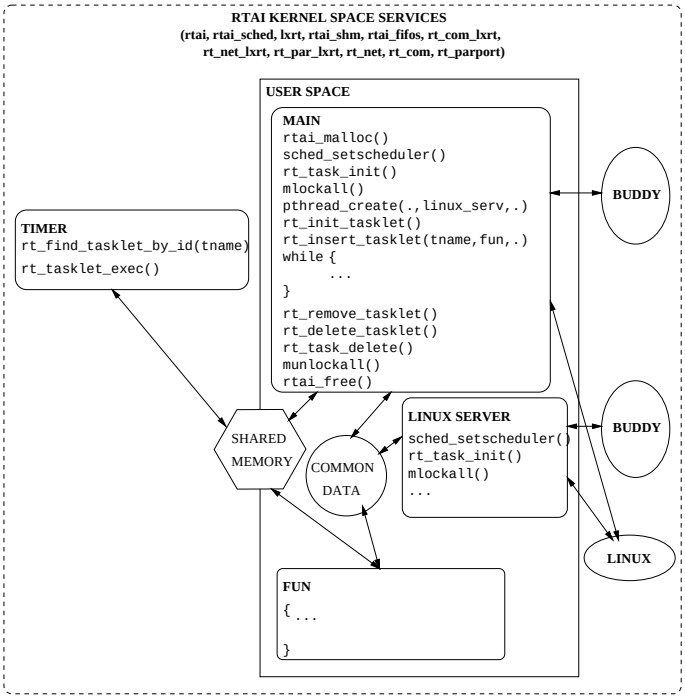

Figure 4: Typical use of a hard interrupt tasklet (timer) in user space

worst. The maximum jitter measured during all the tests was 15 μs.
All tests were run using each of the schemes described above. The specific task periods will be given, for each test case, in the following part of the paper.

In the following paragraphs a brief description of the experimental applications made at DIAPM, using the LXRT module is reported . For more details about the related activities see the references.

5 Vibration control of a panel

This section describes the active vibration suppression experiment of a flat aluminum panel (600x400mm), using a Multi Input Multi Output (MIMO) sub-optimal control system and three couples of piezoelectric sensors and actuators applied on both the two sides of the panel [4].

5.1 Suboptimal control

The design of the co-located control system is obtained from a model which correlates experimental data and numerical simulations. These are based on an integrated structural-piezoelectric-electric finite element model describing the dynamic behaviour of the structure and of the embedded piezoelectric devices. It can also take into account signal conditioning circuits, power amplifiers and any other electronic device used to implement the control system. The sub-optimal regulator has been designed by using an existing, in-house developed, software, mainly aimed at aeroservoelastic systems, e.g. the design of active control for aeroelastic systems. Such a program allows to design robust sub-optimal regulators that match various performance and stability criteria. It is based on a multi-model, multi-objective approach that minimizes quadratic cost functions, in time and frequency domains [5]. Therefore, the designer can obtain a particular time response performance of the closed-loop system and at the same time can achieve a very accurate frequency-domain loop-shaping.
The most important advantage of the sub-optimal approach is that it allows to assign an arbitrary structure to the regulator, ranging from a simple direct feedback of the measurements to a dynamic compensator. In this application, a direct feedback scheme has been adopted, and an optimization of only the time domain cost function has been performed, to obtain a square positive semi-definite feedback matrix. For further details see [4].

5.2 Results

Table I shows control results in presence of single harmonic excitation conditions at several frequencies and in presence of a multi-tone excitation signal. The reduction ratios of the signals of the sensors are reported. It is apparent that the first two modes are considerably attenuated by the controller, whereas the third mode is nearly unaffected, because it is only partially controllable by the current experiment setup of the locations of sensing/actuating piezos (Fig.5). Such results are obtained using a sampling frequency of 12 KHz.

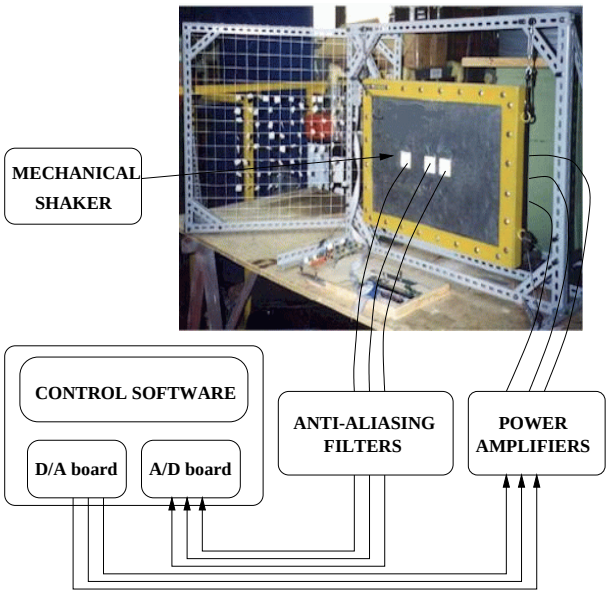

Figure 5: Experimental setup

	attenuation [dB]			
frequency	acc.	Vp1	Vp2	Vp3
89 hz	-15.15	-15.56	-14.26	-15.8
142 hz	-13.64	-3.5	-9.3	-12.31
227 hz	-9.54	-5.7	-7.36	-2.07
multifreq	-7.45	-6.66	-5.7	-5.8

Table I: Attenuation factors with suboptimal control

6 Noise reduction in a duct

The active control of sound [6] involves the introduction of a number of control sources, called secondary

in literature, driven in such a way that the field generated by them interferes destructively with the field caused by the noise sources, called primary.
In this application an Active Noise Control (ANC) system has been developed to reduce the sound field inside a plexiglas duct, 2 m long (Fig. 6). The primary source is constituted by a loudspeaker mounted at one end of the pipe and driven by a signal generator with a pure sine signal, whose frequency varies over the range 100-400 Hz. The secondary source is another loudspeaker, perpendicular to the duct and placed at the middle of the tube length.

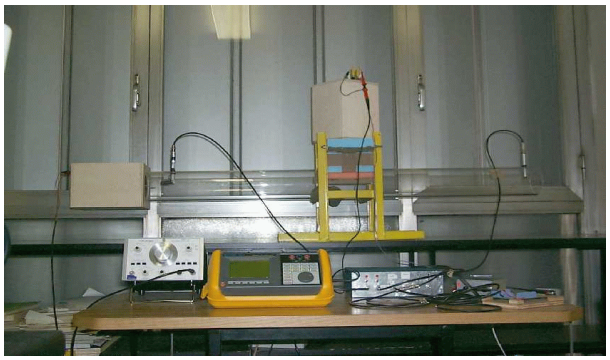

Figure 6: Experimental setup

6.1 FXLMS feedforward adaptive control

The ANC system implemented here is a so called FXLMS feedforward adaptive control. In this configuration (Fig. 7), the reference signal coming from a microphone placed close to the noise source is processed by a digital adaptive Finite Impulse Response (FIR) filter to generate the control signal that is applied to the secondary source. Another microphone, placed where its desired to "create silence", measures the residual noise and is used to adjust the coefficients of the adaptive filter, by means of a Least Means Square (LMS) algorithm.

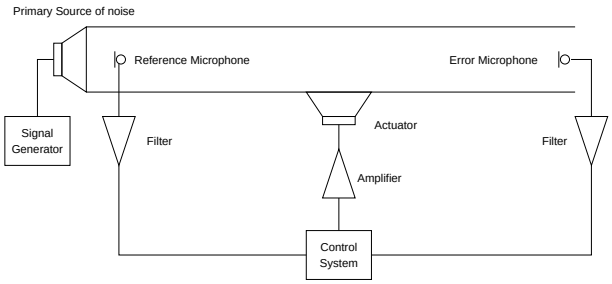

Figure 7: Control architecture

The fact that the electrical reference and the error signals are obtained from the acoustic pressure using a microphone and the cancelling sound is produced by the electrical output signal using a loudspeaker introduce a delay in the system. To compensate for this "secondary-path" transfer function (indicated with S(z)in the scheme reported in Fig. 8), a possible solution is to estimate it (i.e. the transfer function between the control loudspeaker and the error microphone) with an off-line identification technique, using a FIR filter and to place such an estimate after the reference signal. In such a way a so called filtered-X LMS (FXLMS) algorithm is realized. For more details see [6].

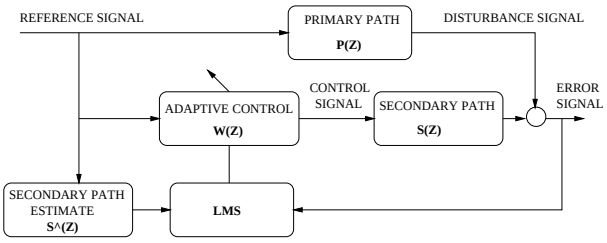

Figure 8: Feedforward scheme

6.2 Results

Table II shows the attenuation factors in decibel obtained with the FXLMS feeforward control system. The great attenuation is due either to the feeforward scheme which uses the information of the disturbance signal to enhance closed-loop performances and to the simplicity of the plant. The sampling frequency for this experiment is 9 KHz, because of the heavy computational load of the algorithm.

frequency [Hz]	*attenuation* [dB]
100	35.7
150	36.6
200	37.8
250	38.2
300	38.5
350	40.1
400	39.9

Table II: Attenuation factors with feedforward control

7 Active control of combustion instabilities

Thermo-acoustic instability is a well-know phenomenon that can occur when a flame, or any other heat source, is located inside a chamber. The control of this phenomenon is an important task in a combustion systems, because of the deleterious effects in term of performance, environmental pollution due to the bad combustion process and the risk of structural damages that such instabilities could produce.
The benchmark problem of thermo-acoustic instabilities is the Rijke Tube [7]. It is simply a long vertical tube containing a compact heat source that produces acoustic emission, provided that the heat source is located in the lower half of the tube.

7.1 Rijke Tube's experiment

In this application an active acoustic feedback control has been developed to reduce the noise emissions of a Rijke Tube. The primary noise source produces an acoustic fluctuation of the pressure inside the tube, measured by a pressure transducer. The signal coming from this sensor is processed by a digital controller and then connected to a loudspeaker perpendicular to the tube at the primary source length. This scheme (Fig. 9) provides a secondary noise source inside the tube that, by controlling the flow, interferes destructively with the acoustic fluctuation of pressure generated by the heat source (closely working with the same mechanism of the duct's experiment). In this case the regulator is a simple inverter amplifier, which in theory should have an infinite gain in order to minimize the closed-loop transfer function amplitude. The inversion of the signal is accomplished to obtain a phase of 180^o.
An important role in this architecture is also covered by the electro-acoustic transfer function (from the secondary source to the sensor) that, basically, represent a time delay (or a phase shift) between the primary and the secondary source. To achieve good reduction ratio of the noise emission, the regulator should consider this effect, i.e. by using a digital phase shifter. However in this test it has been verified that the phase margin was big enough to compensate the delay.

7.2 Results

The efficiency of the controller can be appreciated looking at Fig. 10, displaying the pressure at the point where the control sensor is located. The control begins operating after 5 seconds and for 20 seconds. The sampling frequency is 5 KHz.

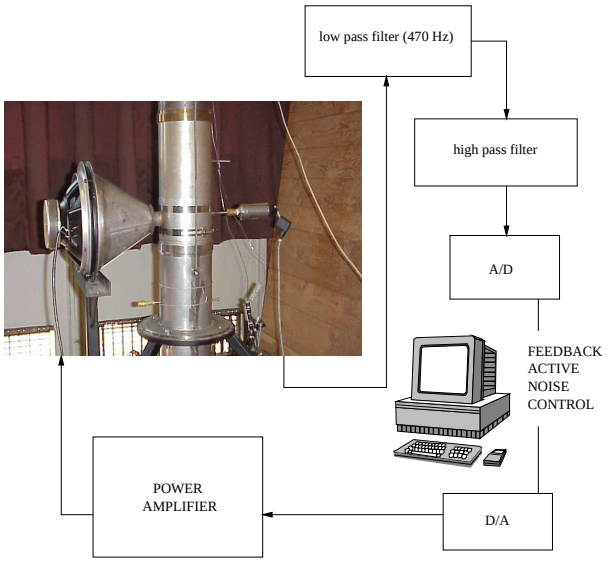

Figure 9: Experimental setup

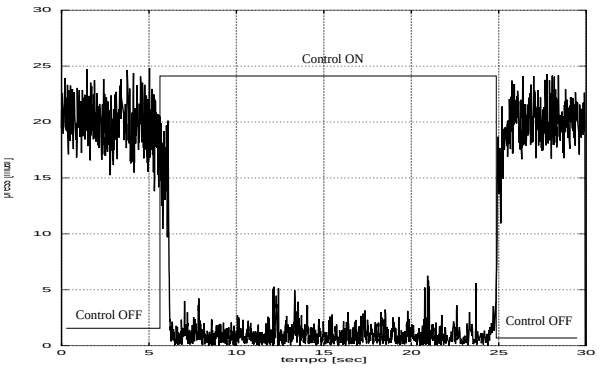

Figure 10: Experimental results

8 Concluding remarks

Born for tutorial research and teaching at DIAPM, LXRT has evolved up to being a complete tool for implementing, from the initial development to the final production phase, real and complex hard real time applications in user space, practically without any

performances loss respect to its kernel space counterpart.
The full symmetric approach allows any previous RTAI user to have the same kernel module function calls and so to be familiar with its usage, and new users to quickly learn how to build a soft/hard real time RTAI-based user space application, eventually converting it to a Linux kernel module. The performances of hard-LXRT, as reported in the three controllers presented here, demonstrates that it is suitable not only as a development but also as a production tool.

9 Acknowledgements

This research has been supported by grant RAAZS202/0/00 from the Italian Electric Power Company. Thanks to Prof. Ghiringhelli for his help and technical support, and to Davide Martini for his precious cooperation.

References

[1] E. Bianchi, L. Dozio, P. Mantegazza, G.L. Ghiringhelli, *Complex Control System, Applications of DIAPM-RTAI at DIAPM*, Real Time Linux Workshop, 1999 Vienna - Austria.

[2] E. Bianchi, L. Dozio, D. Martini, P. Mantegazza, *Applications of a hard real-time support in digital control of complex aerospace system*, AIDAA Congress, 1999 Turin - Italy.

[3] D. Beal, E. Bianchi, L. Dozio, S. Hughes, P. Mantegazza, S. Papacharalambous, *RTAI: Real Time Application Interface*, Linux Journal, April 2000.

[4] E. Bianchi, G.L. Ghiringhelli, D. Martini, P. Masarati, *Neural Active Control for Vibrations and Noise Suppression*, I.C.A.S.T., 1999 Boston - USA.

[5] G. Attanasio and P. Mantegazza, *Design of Low Order Flutter Suppression System*, CEAS Int. Forum on Aeroelasticity and Structural Dynamics, 1997, Rome - Italy.

[6] S.M. Kuo, D.R. Morgan, *Active Noise Control Systems*, Wiley Interscience, 1996.

[7] M.A. Heckl, *Active Control of the Noise from a Rijke Tube*, Journal of Sound and Vibration, 1988.