Linux Real Time Application Interface (RTAI) in low cost high performance motion control

Lorenzo Dozio[†], **Paolo Mantegazza**[‡]
Dipartimento di Ingegneria Aerospaziale,
Politecnico di Milano,
via La Masa 34, 20158, Milano, Italy

Keywords: hard real time, real time operating systems, Linux, digital control, motion control.

Abstract

Advanced high performance multi-input-multi-output motion controllers are generally implemented on dedicated digital signal processors and microcontrollers, having an interrupt latency within a few execution cycles at most. The paper gives an operational view of the hard real time definition and an appropriate perspective on why low cost general purpose computers can be an effective substitute for such high demanding hard real time applications, provided a suitable hard real time operating system is available. The Real Time Application Interface for Linux described here is a viable and effective open-free source software approach for adding hard real time capabilities to a widely available general purpose operating system. It keeps real time applications separated from non real time ones, achieving high efficiencies for both kinds of executions by affording appropriate synchronisation and communication tools to allow an efficient interaction between the two environments. An overview of the development services and tools made available within such a framework is given, along with a sample of specific motion control systems implementations and a list of some known applications.

1. Real time (using low cost general purpose computers and open-free source operating systems)

The term "real time" can have significantly different meanings, depending on the audience and application at hand. The computer science literature generally divides real-time systems in two main categories: *soft* and *hard*.
A soft real time (SRT) system is characterised by its ability to execute a task according to a desired time schedule on the average. A video display is usually taken as a typical SRT example. It is clear that, because of the human eye dynamics, the loss of an occasional frame will not cause any perceived system degradation, providing the average case performance remains acceptable. Even if interpolation techniques can often be used to compensate for missing frames, the system remains SRT; the real frame is missed and the interpolated one is derived rather than actual timely data.
Hard real time (HRT) systems instead embody guaranteed timing, cannot miss deadlines and must have bounded latencies, whose level depends on the particular application at hand. So an HRT system cannot use average case performances to compensate for worst case results. A typical example of an HRT system consists of a controlling system (computer) and a controlled system (plant). It is imperative that the state of the plant, as perceived by the controlling system, is consistent with the actual plant state, within an acceptable error margin (noise level). Moreover timing correctness requirements arise for the control actuations, which have to be performed according to the sampling rates for which the discrete time control system has been designed.
Generally speaking hard real time constraints can be met with strict determinism by dedicated Central Processing Units (CPU) only, e.g. Digital Signal Processors (DSP) and DSP like

[†] Research Fellow, dozio@aero.polimi.it, +39-02-2399-8329
[‡] Full Professor, mantegazza@aero.polimi.it, +39-02-2399-8340

microcontrollers, having a guaranteed interrupt latency in the range of a single/few execution cycles. General Purpose CPUs (GPCPU), i.e. any brand used in workstations, desktops, personal computers and their industrialised clones, are in principle rather unsuitable for hard real time applications. In fact Virtual Memory (VM) and its related Memory Management Unit (MMU), high dependence of performances on multi level caches, possible bus arbitration from "intelligent" Input Output (IO) subsystems, high depth piped execution and speculation, subject them to many non deterministic latencies and jitter, that can largely exceed even the longest instruction execution time. It is thus important to get both an operational view of the hard real time definition and an appropriate perspective of why GPCPUs are usable indeed for hard real time applications related to high demanding control systems, such as those typified by advanced high performance Multi-Input-Multi-Output (MIMO) motion controllers. To more and more meet high demanding specifications nowadays such controllers often include some form of adaptivity, force control and active vibration suppression. Taking a high precision profiling machine as an example it is likely that, in such a view, the sampling rate could approach the 10 KHz range. A well programmed modern GPCPU has double precision floating point capabilities from a few to many hundreds millions Floating Point Operations (multiply-add) per Seconds (FLOPS), with possible giga FLOPS peaks achievable by using vector computation units (Single Instruction Multiple Data, SIMD), that come for free with many recent GPCPUs. So it allows the implementation of fairly complex control schemes, fully featuring what hinted above. Add the possibility of easily achieving even higher performances using Symmetric Multi Processors (SMP) with the adoption of off the shelf consumer dual processors boards, with just a small increase of the cost of a single processing unit, to realise that such a solution can meet high requirements at unprecedented cost/performance figures.
Let us now suppose that the related process runs over a GPCPU with a Real Time Operating System (RTOS) which allows for the test code to be locked into memory, to be prevented from being paged to hard disk, and to be scheduled with a precisely timed execution at 10 KHz. It will roughly do the following:

```
Loop forever
        acquire data (sensors and commands)
        compute control action
        output data (actuators and logs)
        do some bookkeeping and supervisory work
        wait for the next sampling period
        read absolute resume time and compare it to the expected one (possibly)
end loop
```

If our application is run standalone on a GPCPU, the possible time check included in the code will likely indicate worst case latencies in the range of few microseconds. Such a situation is rarely met in real applications; often our controller needs to communicate data through a fast link, to receive commands and to supervise its operation. It is not uncommon that some logging of data to a local disk is required also, not to speak of more complex data visualisation and monitoring. So the control action has to work cooperatively in a prioritised fully preemptable multitasking execution environment. Because of the previously hinted GPCPUs architectural features, such an execution mix can lead to non deterministic latencies, here called jitter, that can, pessimistically, peak up to 30 microseconds worst case. At a first sight a jitter amounting to 30% of the sampling period will impede classifying such a system as hard real time. Nonetheless if we dare trying it and look at the finishing and tolerances of our machined items we will verify that they meet our top expectations, around the clock, seven days a week. A comparison against any equivalent dedicated DSP implementation, i.e. one insuring almost no jitter and a worst case latency within a few microseconds always, will show no difference in results. The reason is clearly in the fact that our hard real time specifications must be related to the actual system bandwidth, and only indirectly to the sampling rate. Roughly speaking it means that our 10 KHz controller will likely be controlling a 1 KHz bandwidth system for which a worst case 3% timing uncertainty will produce measurements

and command errors that are guaranteed to be filtered, down to an acceptable noise level, by the inherent system low pass attenuation.
What just said makes it clear how hard real time should be specified and why GPCPUs can be used in practice for many demanding high end motion control systems, once a suitable real time operating system is available. It should be remarked also that such a point of view makes blurred any distinction between soft and hard real time. In fact mating the classification of real time applications to the bandwidth of a system makes the transition from soft to hard mainly related to the cut off frequency of interest. So there is no clear cut between hard and soft real time but simply a continuous transition from low to high bandwidths and when we say that within soft real time performances can be satisfied "on the average" we simply acknowledge that averaging is just a kind of digital low pass filtering acceptable for the bandwidth at hand. Despite such a point of view in the followings we will continue using the terms hard/soft for reason of convenience.
At this point it must be noticed that there are many RTOSes that work appropriately on GPCPUs. Often they promote themselves on the base of performance figures related to task switching times, and their likes, but miss pointing out to their users that their performances are dominated by the architectural features of GPCPUs and not by task switching times and worst instructions count based latencies, nowadays often at a sub microsecond level. In such a view anything that can be honestly called an RTOS will afford equivalent performances on GPCPUs and the choice becomes more and more dominated by safety and reliability, availability and ease of use of development tools and programs, support, cost and so on. It is at this point that Free Open Source Software (FOSS)[1] based RTOSes can have an edge, with the added advantage of full code sources availability. It is worth noting that what said above can be easily verified by using even the lowest performance personal computers one can buy at any general store today. It can also be safely extended to lower end GPCPUs, such as those we could roughly classify as old plain Pentium class, sold no more for personal computers but still used in industrial applications. In practice, once design specifications and performances are scaled accordingly, anything said should be acceptable for any 32 bits GPCPUs marketed today.

Then, assuming we are interested in applications that fit into the just presented framework, we will now consider if it is possible to meet the required hard real time performances within any existing truly FOSS operating systems. Among these the most known are likely Linux[2] [1] and the BSD streamline [2], none of which has true hard real time capabilities. Out of them Linux is likely the most widely known and advertised and, being also the one we will use, we concentrate on it. Avoiding any detailed explanation of the legalese and licensing term that these systems bring with them, we will assume them as FOSS in a loose sense[3], leaving to the interested reader any further digging into what accompany the various FOSS licensing and the subtleties related to these terms [1-3].
Despite supporting some POSIX Application Programs Interfaces (API) for real time extensions, Linux is a standard non preemptable UNIX like operating system, with superb performances as a general purpose computational/desktop/server/multitasking-multiusers operating system, but with limited soft and no hard real time capability. More recent releases have been fitted with distributed preemption points, already used in UNIX SVR4.0 [4], that should make it possible to guarantee worst case latencies around one millisecond, with averages in the range of a hundred microseconds, thus making it possible its hard real time usage for very low bandwidth control systems, say of a

[1] FOSS is assumed here without any purist concern and rigor of terminology.
[2] Since Linux is just the kernel but a whole Linux system is based on GNU tools, a better name should be GNU-Linux; we acknowledge this and use Linux just for conciseness.
[3] Our view of FOSS is simply "use it and do what you want" but: fairly and fully acknowledge its usage, do not gain money on what is not your work, if you fix-improve something the community has made available contribute it back absolutely, try contributing something of your own. It must however be pointed out once more that there are much stricter views of the constraints implied by using FOSS, especially in relation to proprietary applications under GPL Linux, other licensing schemes being instead closer to our point of view.

few tens of Hertz. The adoption of preemption points will become native with the next major Linux releases, now under development. So, even if low end hard real time applications are going to fit into standard Linux, it will still remain totally unsuitable for high performance motion controls.
However, being a well performing general purpose operating system with source code and development tools fully available, Linux offers a very good base to which hard real time capabilities can be added. To achieve such a goal with ease and at a low cost it is necessary to devise an implementation scheme that avoids a tight interaction between hard and non hard real time applications, be them either soft real time or standard processes, while allowing adequate interactions between the two separated execution lanes. It is noticed that such a conception does not enforce any undue constraint as it is usually adopted by control system designers, even if a true native RTOS is available, because it allows a cleaner, easier to develop and more robust implementation of complex high performance control systems. Thus instead of seeing the separation of hard and non hard real time applications as a constraint, we take it as a good hard real time design practice to be adhered anyhow and embed it into our Linux hard real time extension. So the Real Time Application Interface (RTAI) for Linux described below will follow such a practice and sets a strict separation of hard real time from the rest as its base foundation, while affording efficient communication and synchronisation services to allow an effective interaction between the two environments.

2. RTAI overview

The RTAI project began at the "Dipartimento di Ingegneria Aerospaziale del Politecnico di Milano" (DIAPM) in 1996/97. It stemmed from the need of making available a tool to support a varied set of internal research activities related to advanced active controls for generic aeroservoelastic systems, including large space structures, acoustics and flexible manipulators. Its aim was to make it possible their development, implementation and testing on standard 32 bits personal computers (PC) and data acquisition cards, by using high level language programming tools, so that anybody, including graduating students, could proceed to their implementation with a relative ease in building it all.
It mostly gathered scattered real time software services and tools used under DOS on earlier less powerful 16-32 bits PCs, mainly through terminate and stay resident programs that trapped DOS slaving the PC to real time execution needs. The appearance of high performance 32 bits PCs made available much more computational power and memory, thus allowing relaxing all DOS constraints with a consequent quantum leap that made it possible envisaging the use of off the shelf PCs as a full substitute of costly dedicated DSPs boards. Nonetheless it did not came without pain as the simplicity of the DOS way of working disappeared, thus making it compulsory the use of a hard real time RTOS. Budget constraints and the satisfaction of some DIAPM researchers in using Linux as a general purpose operating system brought the idea of adding hard real time capabilities to it. Further details on the early story of RTAI can be found in the documentation available at the RTAI home site [5] and need not to be repeated here.

As it will be seen RTAI is integrated into Linux through a text file containing a set of changes to its kernel source code, known as a patch, and a series of add on programs expanding Linux to hard real time. As such it is bound to be a GPLed licensed [3] FOSS, as Linux is. So RTAI has been freely available on the net from its very beginning. In 1999, after the appearance of the 2.4.xx release of Linux, RTAI began being relatively widely known and after some time it became an FOSS development effort with a team of developers worldwide. The coordination of the RTAI projects remained at DIAPM, which hosts its home site also. Thanks to such a team RTAI has expanded the CPUs supported to include the following 32 bits cores: INTEL x86, PowerPC, MIPS, StrongARM, MMUless Motorola Coldfire, with some other new ports being either pursued or under consideration.

We will now briefly explain how RTAI expands Linux to hard real time. The presentation will cover only the essential concepts of its implementation while trying to give a clear insight of what has been done to meet our objectives. To that end RTAI patches the Linux kernel by installing a generic Real Time Hardware Abstraction Layer (RTHAL). RTHAL performs three primary functions:

- Gathers all the pointers to the time critical kernel internal data and functions into a single structure, to allow the easy trapping of all the kernel functionalities that are important for real time applications, so that they can be dynamically substituted by RTAI when hard real time is needed. Generally speaking such kernel functionalities include all functions and data structures required to manipulate: the hard interrupt flag, external interrupts and internal traps/faults, the system call, programmable interrupt controllers and hard timers. The related objects are substituted by pointers that can be changed dynamically.
- Reworks the related Linux functions, data structures and macros to make it possible to use them to initialise RTHAL pointers for normal Linux operations.
- Changes Linux to use what pointed in RTHAL for its operation.

The related patch is quite simple and changes or add about a hundred lines in the Linux kernel. Linux is a dynamically expandable kernel and new functionalities can be added to any running instance of it by linking a relocatable object code, called module in Linux jargon. Such a feature is used to extend Linux to hard real time. This is achieved by an RTAI specific module that steals from Linux all the RTHAL hardware related objects and emulates them in software while letting Linux to continue working unchanged. This includes substituting the hard interrupt flag with a variable and the trapping of all the Linux interrupts, faults/traps, system call, programmable interrupt controllers and timers, substituting them with soft dispatching functions [6] so that any Linux operation is carried out with hard interrupts fully enabled. After that only hard real time activities can freely use the hardware and so have full priority and preemption authority on Linux. It must nonetheless be noted that the hardware is unique and must be shared with Linux anyhow. It is thus possible that a Linux interrupt happens within hard real time activities and they must be registered and pended for subsequent soft dispatching and processing, when no hard real time game is being played, without loosing any of them. If such an interrupt pending was carried out on a DSP it would cause an added latency deterministically related to the corresponding count of machine cycles, with a totally unnoticeable loss of hard real time performances. Instead the architectural features of GPCPUs previously pointed out will cause a bounded random jitter that is responsible of the unavoidable scatter and dependency of latencies on the overall system activity, as it was explained in the introductory paragraph. What just presented is the core concept that allows making Linux as a truly hard real time operating system as allowed by a GPCPU .

The ideas exploited by RTAI have been known to and used by the operating system community for quite some time. They have progressed and matured into more comprehensive generalised conceptions that have made it possible structuring effective hierarchical operating systems consisting of different layer of operating systems [7-10]. A FOSS implementation of such concepts, called Adaptive Domain Environment for Operating Systems (ADEOS) [11] is now available. In its full breadth and scope the purpose of ADEOS is to provide a flexible environment for sharing hardware resources among multiple operating systems, or among multiple instances of a single operating system. The ADEOS nanokernel, i.e. the scheduler of the different operating systems instances opens a full range of new possibilities, notably in the fields of SMP clustering, patchless kernel debugging and real-time systems addition to general purpose operating systems. To this end ADEOS enables multiple prioritised domains to co-exist simultaneously on the same hardware. To share the hardware among the different operating systems, ADEOS implements a pipeline scheme which allows them to virtualise all what RTAI gathers into its RTHAL. Every domain has an entry

in the pipeline and each event that comes in the pipeline is delivered to the registered domains according to their respective priority. In order to achieve hard real time determinism, RTAI over ADEOS is the highest priority domain which always processes interrupts before the Linux domain, thus serving any hard real time activity either before or fully preempting anything that is not hard real time.
RTAI has then been ported onto ADEOS to allow the RTAI development team the possibility of getting rid of kernel patching and maintenance while exploiting a more structured and flexible way to add real time to Linux.

It is so that with the RTAI (ADEOS) core module installed Linux can extend its execution domain to hard real time. Nonetheless at such a point there is little that can be done but the bare interrupt handling. It thus arises the need of providing scheduling services to be executed in hard real time and mated with efficient communication tools to allow interacting with standard Linux scheduled tasks.
RTAI does provide the required schedulers along with a wealth of services. The RTAI schedulers are fully preemptible and can be scheduled directly from within interrupt handlers so that Linux has no chance of delaying any RTAI hard real time activity. Before giving a short overview of the RTAI schedulers and related services, it is important to anticipate the important fact that they allow to symmetrically work inter-intra both user and kernel space, by using the same APIs everywhere. So any communication and interaction between the two lands is simple in RTAI, down to full hard real time interrupt handling in user space. It is possible to implement hard real time multitasking applications in kernel space also, by either using standard kernel threads or RTAI proper tasks. The possibility of embedding a control system as part of the kernel allows achieving maximum execution efficiency. In fact RTAI proper kernel tasks add further efficiency in kernel space since they can avoid using the memory management unit of GPCPUs, with a saving in task switching time that can be significant on low end GPCPUs. Thank to such a feature a control system can begin being, wholly and safely, designed in user space and then migrate to become an integral component of the kernel, in part or as a whole, when it is required so to achieve the best performances on the GPCPU at hand. This is a relatively original feature of RTAI/Linux and RTAI has always seen a strict separation of kernel and user space as a drawback for hard real time applications. So it has always striven for their easy integration, leaving any related choice to the control system designer. This is part of the general philosophy behind RTAI: afford mechanisms and not policies; meaning that policies must be wholly in the hand of users. The RTOS must provide just adequate mechanisms to make it easy implementing any policy without any undue constraint.

RTAI makes available three schedulers: Uniprocessor (UP), optimised for uniprocessor machines; Symmetric Multi Processors (SMP) and Multi UniProcessor (MUP) for symmetric multiprocessors applications. The SMP scheduler affords the best compromise between flexibility and efficiency in kernel space applications using RTAI proper kernel tasks, as it can schedule any ready task on any CPU, while allowing to selectively impose selected tasks to run on a specific CPU or CPUs cluster. The MUP scheduler instead imposes that any task is assigned to a specific CPU from its very creation and can achieve better performances because it can exploit memory caching more efficiently. RTAI specific kernel tasks can in any case be moved to different CPUs dynamically at execution time. Instead inter CPU migration of Linux tasks and kernel threads cannot be done in true hard real time. There is no restriction in the use of any scheduler, real time tasks can interact without any constraint, irrespective of what CPU they are running on. SMP and MUP schedulers can be used also with uniprocessors. Another important feature of all the schedulers is the possibility of choosing between either a base periodic timing, with a fixed assigned time resolution tick, the approach mostly used in RTOSes, and an arbitrary timing, allowing scheduling a task at the resolution of the available clock by firing a oneshot timer at the time instant imposed by the highest

priority task waiting on the timed list. The oneshot mode avoids any compromise on the least scheduling resolution, thus giving an almost continuous time resolution, while the periodic mode requires to have any task timed at an integer multiple of the basic timer period. However we must recall what pointed out in the introduction and avoid any illusion that on GPCPUs a task can be scheduled with a nanosecond precision. It is also important to note that the MUP scheduler uses independent per CPU timers and each of them can run independently from any other, so that timers mode of operation can be freely assigned, e.g. there can be periodic and oneshot timers and periodic timers need not to run at the same period.
RTAI schedulers make available the following scheduling policies:

- Fully preemptable First In First Out (FIFO), for voluntary co-operative scheduling. A task owns the CPU till it does not release it or a higher priority task preempts its execution. Under FIFO scheduling there is a support function to help meeting periodic tasks deadlines with statically assigned priorities according to the Rate Monotonic Scheduling (RMS) concept.
- Round Robin (RR), like FIFO but only up to a certain allowed per task time slot, after which the CPU is tentatively handed over to any equal priority task waiting on the ready list.
- Early Deadline First (EDF), to dynamically assign priorities in order to meet periodic tasks end of execution deadlines. It requires that the user assigns a relatively good estimate of the execution time required by each periodic task.

It must be noted that under symmetric multiprocessing it is also possible to handle external interrupts either in a symmetric way or to force them to a specific CPU, or CPU cluster.

Let us take now a quick overview of the services made available by RTAI and its schedulers, recalling, once more, they are symmetrically available inter-intra kernel/user space and pointing out that, even if not cited explicitly, all synchronisation and communication functions are fitted with both various conditional and timed out executions mode, using either relative delays or absolute times:

- ***Basic task management*:** time management and conversions, dynamic priority assignment, scheduler policy assignment, scheduling locking/unlocking, counting suspend/resume to avoid trivial deadlocks, task yielding, busy, absolute and relative timed suspensions, specific support for periodic execution.
- ***Memory Management*:** shared memory for inter tasks, inter-intra user/kernel space data sharing, dynamic memory allocations.
- ***Object registration*,** to allow an easy reference of RTAI objects across applications by using short alphanumeric mnemonic identifiers.
- ***Semaphores***: wait, send, broadcast on: counting, binary and resources with full priority inheritance to avoid priority inversion.
- ***Conditional variables*:** wait, signal, broadcast, equivalent to the related POSIX APIs, but with an RTAI specific implementation.
- ***Bits synchronisation*:** likely an inappropriate RTAI jargon for multi events/flags/signals synchronisation, i.e. semaphore like operations with different logical masking/unmasking on a set of bits at call and return time.
- ***Mailboxes*:** send, receive of messages with multi readers/writers capability, messages queued in either FIFO or priority order. It is possible to use overwriting and urgent sends, broadcast a single message to all task waiting to receive on a mailbox queue, preview any message before reading it.
- ***Direct Intertask Messages***, either in the form of asynchronous: send, receive, or with synchronous remote procedures calls (RPC): rpc, receive, return. Messages can be either a single 32 bits value or any size, with the possibility of being previewed before receiving them.

RPCs implement priority inheritance to avoid priority inversion and can integrate such a feature with resource semaphores.

- ***Tasklets*** and ***Timers*****,** for prioritised executions of either asynchronous event (tasklets) or time (timers) driven non blocking tasks. Timers allow a lean implementation of flexible timing policies without using a fully featured RTAI task.
- ***User Space Interrupts (USI) handling*****,** allows to implement interrupt/drivers management directly in user space, making it easier their development and test, eventually going to kernel space with much less troubles and only if needed.
- ***FIFOS communications*****,** for direct data exchange between interrupt handlers and RTAI and Linux tasks.
- ***Watchdog monitor*****,** to help supervising task execution and avoid locks due to inappropriate timings.
- ***Hard real time Posix support***, for kernel threads and RTAI proper tasks.
- ***Distributed services (net_rpc)*****,** to allow using any of the RTAI APIs and data on remote nodes. It integrates distributed and local applications by just adding a node/port identifier in front of any RTAI function call argument list. In such a way any application can be run on a single machine or on many networked machines without changing a single line of its source code. It can use the ***RTNet*** support addressed below.

Moreover RTAI can use the Linux Trace Toolkit [12] and the GNU debugger as helper development tools. The RTAI distribution contains also a support script that allows you to easily prepare a floppy to boot and use RTAI on small embedded systems, e.g. those based on PC-104 boards.

To be usable in practice RTAI needs clearly also some support for peripherals. So it comes with a real time serial and parallel port driver while many other drivers can be downloaded from the network. They include:

- ***RTNet*** [13], a real time protocol stack offering standard UDP socket APIs.
- ***COMEDI*** [14], a vast collection of drivers for a large variety of data acquisition plug-in boards. The drivers are implemented as a core Linux kernel module providing common functionality and individual low-level driver modules. RTAI has proper APIs to make them symmetrically usable in kernel and user space.
- ***CANBUS***, including a Linux universal driver [15] supporting 82c200/sja1000 and 82527 based I/O interfaces and a RTAI specific driver for Intel 82527 [16].
- ***SPDRV***, a real time serial port driver symmetrically usable in kernel and user space applications.

3. RTAI meets Computer Aided Control System Design (CACSD) and supervision

CACSD subsumes a broad variety of computation tools and computation environments for control system design, real time simulation, with and without hardware in the loop, making the best use of high desktop computer power, graphical capabilities and ease of interaction with low hardware cost. Integrated CACSD software environments allow an iterative control system design process to be automated with respect to multi-objective performances evaluation and multi-parameter synthesis tuning. Visual decision support provides the engineer with the clues for interactively directing an automated search process to achieve a well balanced design under many conflicting objectives and constraints. Local/remote on line data down/upload make it possible a seamless interaction with the control system, to supervise its operation and to adapt to changing operational needs.

3.1 MATLAB/Simulink/RTW – SCILAB/Scicos/CodeGen/Syndex and RTAI-Lab

In such a context the MATLAB/Simulink/Real-Time Workshop [17] suite, simply RTW in the followings, offers a high-quality proprietary solution, while the Scilab/Scicos/Syndex suite [18] is a widely known and somewhat equivalent FOSS system. RTW is an automatic C language code generator for Simulink. Under Simulink it is possible to create, simulate and analyze complex dynamic systems by simply connecting functional blocks, mostly available from various preconfigured libraries, within a friendly graphical user interface. Furthermore, being Simulink part of MATLAB problem solving environment, it shares the same straightforward integration of computation, monitoring and visualization. One of the main advantages of RTW consists in its fully configurable code generator that specifies how to transform a Simulink blocks model into a C code, allowing to produce a target software for virtually any operating system and platform onto which MATLAB can be run. This feature is the backdoor for the RTAI interface. Scilab/Scicos/CodeGen have similar capabilities while Syndex automatic code generation goes down to RTOS conception so it seems somewhat more difficult to use. Nonetheless Syndex interfaces for UNIXes with real time extensions are surfacing and are being taken in consideration at DIAPM.
To set up a new control system within such an environment the first step is the creation of Simulink/Scicos block diagrams that represent the system implementing the chosen control strategy. After some simple tests, it is possible to add specific Digital Acquisition (DAQ) blocks supporting the boards to be used and start the code generation phase. Such a support come both with RTAI specific implementations and with a COMEDI based general support, thus allowing to easily access a very large number of off the shelf ready to use DAQ hardware. The process is straightforward and the user has to choose only the right template makefile for the target language compiler to have the control programming automatically generated with a mouse click.

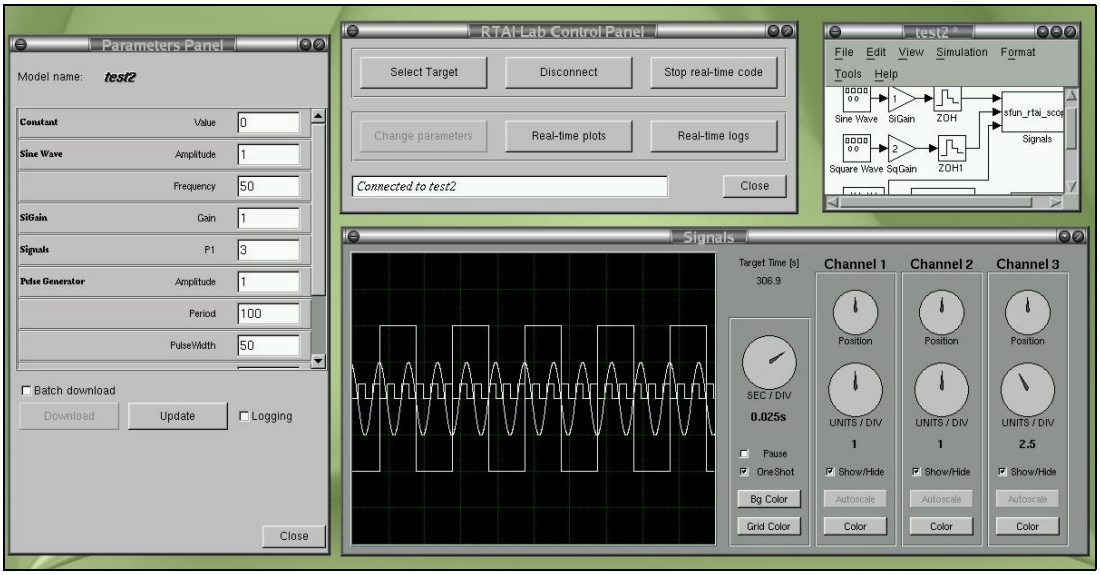

Figure 1: RTAI-Lab user interface

The controller code can then be executed by using the native RTAI networking layer under the supervision and monitoring application called RTAI-Lab. Such a tool allows to integrate running any suite of real time controllers/real_time_simulators, automatically generated by Matlab/Simulink/RTW and/or Scilab/Scicos/CodeGen in a local distributed way, monitor their execution locally/remotely, changing their parameters on the fly for performance supervision, monitoring, tuning and optimisation. The basic concept of RTAI-Lab is to allow two separate systems, the host and the target, to communicate. In a remote implementation, the host is the machine where RTAI-Lab is executing in soft real time, the target is the machine where the

generated hard real time code runs. The host send-receives messages through the net requesting the target to accept parameters changes and to send signal data for graphical displaying and file logging. Local only application comes either by trivially executing the distributed code in "localhost" mode or by exploting the native NET_RPC capability of directly executing in local mode without any change to the executable. This basic scheme can be extended by running on the same host many RTAI-Lab sessions, thus monitoring and interfacing many targets simultaneously. RTAI-Lab architecture has the capability to generate and distribute control systems, by simply providing a library of I/O blocks which embeds RTAI-NET_RPC directly in the above code generators. In such a way the user builds as many Simulink/Scicos models as the number of targets and downloads each one to the corresponding machine. Each model contains the I/O blocks to send to and receive messages from the other parts of the distributed real time controll system thus allowing the execution of syncronised distributed applications in an easy way. It should however be remarked that the user is responsible of implementing his/her own distributed policies by using the conprehensive APIs set provided by NET_RPC. No automatic generation of distributed applications is provided at the moment but will be soon available, at least because of to the quick expansion of RTAI-Lab usage at DIAPM..

3.2 LabVIEW and visual flow programming of hard real time control systems

A well known graphical programming tool is LabVIEW [21]. The binding of RTAI to LabVIEW enables the use of user space hard real time services provided by RTAI directly from within LabVIEW. With such a support a separate LabVIEW execution engine for a Linux task is fired and switched to hard real time mode using RTAI support. Non real time and real time LabVIEW tasks will communicate using RTAI FIFOs. Communication, synchronization and timing between real time LabVIEW/RTAI tasks will be done by calling the appropriate RTAI functions directly from within LabVIEW.

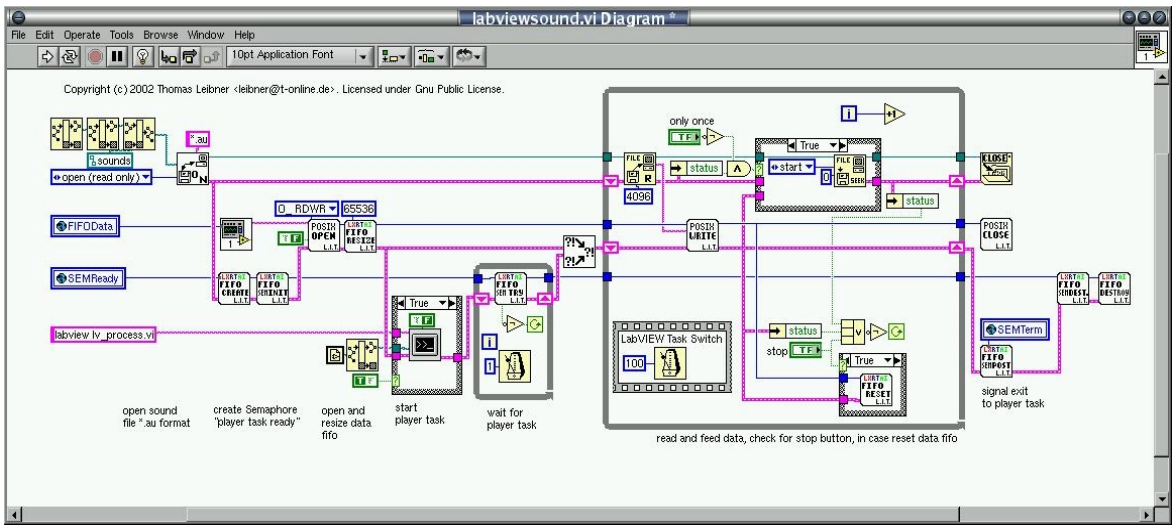

Figure 2: RTAI/LabVIEW flowchart example

The binding is based on LabVIEW Code Interface Nodes (CIN, a service that interfaces graphical programming to self written platform dependant code) sot that a CIN can freely call both RTAI APIs and any other non real time CINs available. Since a CIN is an integral part of a LabVIEW specific program, called Virtual Instrument (VI), the self written code it contains is embedded into the VI itself. LabVIEW implements its own library format, i.e. *.llb and all VIs implementing the LabVIEW/RTAI binding are collected in the library LVL_LXRT.llb. The overhead caused by using LabVIEW as a programming interface is not totally negligible, an unavoidable price to be paid for using a general graphical programming and debugging method for hard real time applications

running on a GPCPUs. However even the least computer power available nowadays allow to use such an approach also for highly demanding controllers.

4. Motion control applications at DIAPM

By interpreting the definition of motion control in a wide sense, i.e. the application of various technologies that use controllable forces (actuators) to achieve useful operations in fluid and/or solid electromechanical systems, this section presents three different motion control applications implemented at DIAPM, naturally using RTAI capabilities and features.

4.1 Tracking control of a flexible manipulator

This is an RTAI based system developed to manoeuvre a manipulator with flexible links along a specified trajectory, with a motion controller integrating active damping of structural vibrations [22]. Precise motions of space manipulators along a desired trajectory is a critical issue for current and future space missions. The structural flexibility of the manipulator links, which usually are large and slender, adds more complexity to that task.
Two control techniques, using a Lyapunov based non linear control and a perturbation approach respectively, were investigated, under the constraint of using as few sensors and actuators as possible to limit weight, complexity and cost, while enhancing reliability. The ultimate goal was to control the motion of the robot end effector without introducing additional actuators besides those naturally located at the manipulator joints. Angular potentiometers at the joints and strain gauges along the links were used. A slewing maneuver was chosen to test the controller performances. Experiments were carried out on a two DOF planar robot model, with a flexible forearm and a rigid arm. The manipulator was mounted horizontally in a cantilevered configuration to simulate the microgravity dynamics on the work plane. The shoulder joint stator was fixed to a vertical support structure. The robot was actuated by two direct drive brushless motors, on the base of measures related to its arms angular positions and to two strain gauge bridges, one at the root and one at the middle of the forearm. The digital controllers were implemented with a scheme based on a “main” program that expands into two POSIX threads of execution and then acts as a user supervisory interface. The first thread, called “controller”, 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 the “main” program and the “linux_server” are soft POSIX real time tasks implementing the Linux scheduler FIFO policy and cooperate in building a suitable user interface. The “main” program takes the burden of user input for supervising the controller parameters, while the “linux_server” provides data logging to a file and monitoring scopes. A precise timed support is provided by the RTAI UP scheduler, used in periodic mode at 500 Hz. The actual hard real-time controller interfaces to sensors and actuators by using an ISA DAQ board, programmed directly in user space.

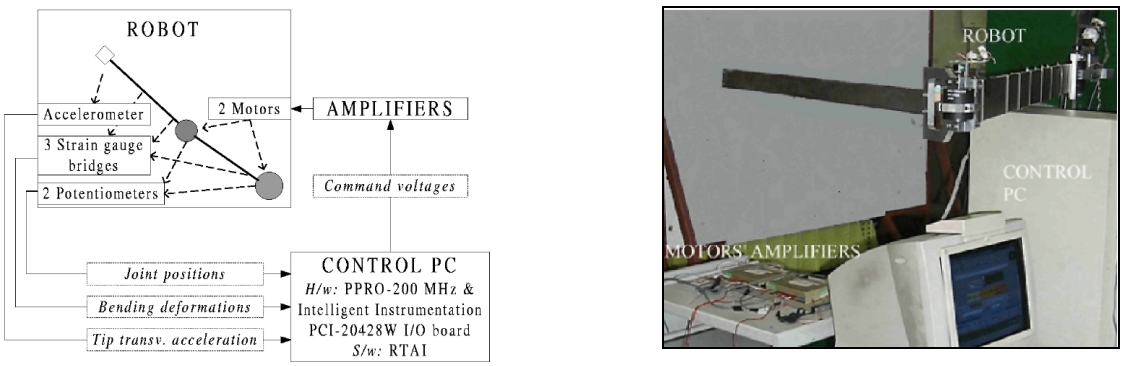

Figure 3: Block diagram and picture of the Space Robot Simulator test-bed

4.2 Fatigue test system

Nowadays almost all commercial fatigue test rigs are realised using digital control systems, with the likely exception of signal conditioning and power units. Such systems are often costly and proprietary and the user is allowed little flexibility of usage, any user specific development requiring being paid, often a lot. So research activities requiring high flexibility in programming arbitrarily complex coordinated multi load histories and a high operational and specimen installation flexibility can be better tackled by in home developed testing systems. In such a view a competitive fatigue test system has been developed at DIAPM using RTAI. It integrates motion control, force control being nothing but a high resolution, high precision, high bandwidth position control, and user interface into a single general purpose PC with two DAQ PCI boards. Such a solution leads to a low cost and easily upgradable system, having all the capabilities of its commercial counterparts at the fraction of its cost. It can control many load channels simultaneously, at the moment up to 64 inputs and 10 outputs can be acquired, adapt to a wide range of actuators and transducers and implement any kind of control policy (force, displacement, hybrid). The control PC does it all: arbitrary load signals generation, test results logging, display and recording, optimisation of test conditions and controllers parameters. The actual fatigue testing rig uses servovaleves controlled hydraulic rams and load cell transducers. The hardware loop controls also a tension to current converter to appropriately drive the servovalves with PID based controllers running at 1000 Hz or more.

Figure 4: Fatigue test rig

The software code is split in two parts, a real time module and a graphical user interface (GUI), once more according to the basic principle to separate what has hard timing constraints from what is not time critical. The hard part is implemented in a kernel module that creates three RTAI tasks. The first task, running at 1000 Hz and at the highest priority, is the “control task” which implements the bank of PID controllers. The transducers signals are acquired by a DAQ board, programmable with different gain for each channel, that are passed through a digital first-order conditioning filter and directly compared with the desired signal for the computation of the PIDs output. It is noticed that all the input transducers have an intrinsic low bandpass response adequate to intrinsically ensuring that no significant aliasing occurs. In any case there is enough computer power, Pentium III 1.8 GHz, to allow to increase the control frequency so that to avoid any aliasing with whatever useful transducer to be used for such an application. Thus any signal conditioning but strain gauge bridges amplification can be easily carried out digitally. A second task, called “surveillance task”, performs the error checking of peak loads and monitors safety sensors alarms. It can automatically suspend the test in case of any alarm occurrence or if any load error overcomes any limit value assigned by the user. The last task plays the role of function generator, sine,

trapezium, triangle, saw-tooth, step and ramp signals, plus any specific user coded function, can be used. Data exchange with the user combines FIFOs sending data from user to the kernel tasks and shared memory for storing the flood of monitoring data to be plotted and logged. The graphical user interface provides also a calibration module to tune the load lines controllers using a semi-adaptive algorithm. A running test supervisory module performs the actual interaction with the hardware, providing a graphical scope like visualisation of any interesting measure and a continuous detailed report of the whole test execution.

4.3 Multi Pulse Width Modulated (MPWM) control of the motion of a large space structure

The last example involves a digital MPWM of on/off thrusters to control the low-frequency modes of a large space structure model in a laboratory facility called Truss Experiment for Space Structure (TESS). The structure is a modular beam-like truss, with a basic cubic bay having a single diagonal on each side, suspended from the ceiling by means of six soft springs. The truss, built with plastic tubes, has an overall length of about 19 m and weighs 75 Kg. TESS is equipped with six pairs of on/off air jet thrusters placed at fixed bays along the truss. The twelve devices are appropriately mounted to operate within the horizontal plane, the jets are co-located in pairs to allow thrusting in opposite directions. Beam motions are sensed by capacitive accelerometers which are positioned in such a way to measure the transverse horizontal accelerations.
Vibration suppression is achieved by means of a linear deadbeat predictive control, where an equivalence is set between a pulse width modulated control law and a parent discrete-time pulse amplitude modulated control. The approach adopted [23] allows more than one pulse to appear within each sampling step, resulting in a multi-pulse-width modulation (MPWM).

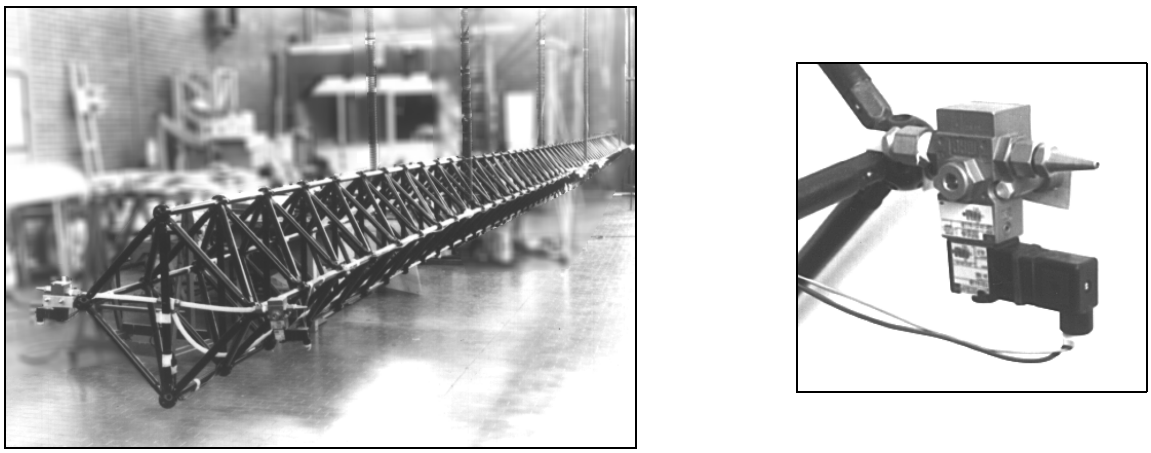

Figure 5: TESS experiment (whole view and details of one jet thruster)

The real time software is implemented on a Dual Pentium III 700 MHz. The use of a dual processor allows distributing the work over two CPUs to profitably exploit their computational resources and obtain an appropriate synchronization and prioritization of all the control activities, which can be divided into three blocks. They are: data acquisition and digital filtering at an oversampling rate frequency (10 KHz), finely timed firing MPWM management, low-frequency (5-10 Hz) controller computations. The oversampling uses dithering and low pass filtering, down to three orders of magnitude, to increase quantisation levels, thus achieving high resolution and a low noise on low level accelerometric signals. It has the added advantage to avoid using any external filter, being it enough to use the low pass accelerometers transfer function to avoid aliased samples. The first two blocks are assigned to the first CPU, the third one to the other to let it exploit the full computational power of the used CPU. The hard time pacing is driven by the DAQ board timer though an RTAI interrupt handler, which simply resets some board registers and fires the analogue data acquisition

task waiting on a semaphore. In turn, at the integer fraction rate chosen for the control action, the acquisition task awakes the control task by sending the conditioned acquired data to a mailbox on which the control task is blocked. So the control task receives sensors measures and carries out the computations needed to determine the amplitude of the control forces. These are then transformed into pulses durations by the referenced algorithm and the related firing times table is used by the firing task to directly set the digital output controlling the solid state relays that drive the opening/closure of the air jets electrovalves. It is remarked that due to the low frequency dynamics of the beam a MPWM firing uncertainty of a couple of tens of microseconds, once more related to the use of GPCPUs, is precise enough to make actuation noise totally negligible.

5. Non-DIAPM motion control applications

The FOSS and totally non commercial nature of RTAI does not allow us to have a complete list of non-DIAPM academic and industrial motion control projects using RTAI. It is nonetheless possible to infer a well established wide usage by monitoring the traffic at the RTAI home site, mailing list and private communications concerning motion control applications. Here is a loose short summary of known applications:

- marble and granite slabs automatic cutter machine;
- ball bearing rings shape control;
- laser cutting machine;
- robot control;
- universal measuring machines;
- electric power conversion;
- electroerosion machines;
- single machine and numerical controlled production centres;
- aerospace flight simulators;
- real time floating point support processor for proprietary axes drives;
- plastic deformation machines.

For further reference to RTAI applications and other Linux real time applications as well, see also [24].

6. Concluding remarks

Since its very beginning RTAI has grown in an impressive way in robustness, services and programming tools provided, thanks also to substantial contributions from non DIAPM developers and users worldwide. It is now a relatively well known hard real time platform within the FOSS RTOSes scenario, with many known academic and industrial applications, including the excerpts related to motion control systems applications cited in this paper, but with likely many more significant applications being undetected. Due to its FOSS nature RTAI might still lack some of the bells and whistles of commercial RTOSes, but it does not fear any performance challenge. It has been somewhat criticised, even within its developers, for too a frenetical development, often without caring too much of documenting it. While we think that its documentation is not so bad, such a situation is typical of its bazaar development style and is common to many FOSS projects. We tend to see it as an indication of strong vitality. After all you can not only use FOSS but feel free of helping in contributing to it otherwise… anybody is totally free to spend any money to buy the best he/she deserves. After paying he/she will get the usual clause “we are not responsible for any damage caused by this software…”, moreover we all know that bugs are always there and will be fixed only if they lead to loosing too much money. With FOSS instead if nobody cares you are free to loose some sleeping and fix bugs yourselves, feeling proud to contribute your fixes back to

help the community. Thus RTAI is just another reminder that FOSS means open knowledge also and no FOSS should ever become a real estate.

References

[1] http://www.kernel.org
[2] http://www.bsd.org
[3] http://www.fsf.org
[4] S. Zeadally, "An Evaluation of the Real Time Performances of SVR4.0 and SVR4.2", *Operating Systems Review*, 1997.
[5] http://www.aero.polimi.it/~rtai
[6] D. Stodolsky, J.B. Chen, and B.N. Bershad, "Fast Interrupt Priority Management in Operating System Kernels", *Proceedings of the USENIX Symposium on Microkernels and Other Kernel Architectures*, San Diego, CA, USA, 1993.
[7] D. Probert, J. L. Bruno, and M. Karaorman. "SPACE: A New Approach to Operating System Abstraction", *Proceedings of the International Workshop on Object-Orientation in Operating Systems*, Palo Alto, 1991.
[8] D. Probert and J. L. Bruno, "Building Fundamentally Extensible Application-Specific Operating Systems in Space", *Technical Report TRCS95-06*, Computer Science Dept., UC Santa Barbara, March 1995.
[9] D. Cheriton, K. Duda, "A Caching Model of Operating System Kernel Functionality", *Proc. Symp. on Operating Systems Design and Implementation*, Monterey, CA, USA, 1994.
[10] D. R. Engler, M. F. Kaashoek, J. O'Toole, "Exokernel: an operating system Architecture for Application-Level Resource Management", *Symposium on Operating Systems Principles*, 1995.
[11] K. Yaghmour, "Adaptive Domain Environment for Operating Systems", available at http://www.opersys.com./adeos
[12] http://www.opersys.com/LTT/index.html
[13] http://www.rts.uni-hannover.de/rtnet
[14] https://cvs.comedi.org/comedi
[15] http://www.port.de/engl/canprod/sw_linux.html
[16] http://www.rtautomation.it
[17] http://www.mathworks.com
[18] http://www.scilab.org
[19] http://A.die.supsi.ch/~bucher
[20] http://www.windriver.com
[21] http://www.ni.com/linux/lin_lv.htm
[22] M. Romano, F. Bernelli-Zazzera, M. Massari, "Numerical and Experimental Results on Closed Loop Tracking Control of Flexible Manipulators", *AIDAA XVI National Congress*, Palermo, Italy, 2001.
[23] L. Dozio, F. Bernelli Zazzera, P. Mantegazza, "Experimental Analysis of the Multi-Pulse-Width Modulation Control of a Large Space Structure using On/Off Jet Thrusters", *Proceedings of the 2002 International Symposium on Active Control of Sound and Vibration (ACTIVE 2002)*, Southampton, UK, 2002.
[24] http://www.realtimelinuxfoundation.org.

Acknowledgments

RTAI developers team, for making it what is now.
P. Gerum, for porting RTAI to ADEOS.
G.L. Ghiringhelli and E. Vigoni, for the fatigue test rig.
F. Bernelli-Zazzera and M. Romano, for the flexible manipulator.