

USING MATLAB-SIMULINK RTW TO BUILD REAL TIME CONTROL APPLICATIONS IN USER SPACE WITH RTAI-LXRT

G. Quaranta P. Mantegazza

Dipartimento di Ingegneria Aerospaziale, Politecnico di Milano,
via La Masa, 34 20156 Milano, Italy
quaranta@aero.polimi.it
mantegazza@aero.polimi.it

Abstract

The paper presents a hard real time Linux environment that can be effectively used to design and implement data acquisition and digital control systems, integrating the functionality of the MATLAB/Simulink/ Real-Time-Workshop suite and RTAI, a freeware hard real time extension for Linux. The development process requires only the ability to use the cited MATLAB suite, almost without any knowledge of C-programming. The resulting software can be run on off the shelf standard personal computers, without requiring overly complex and expensive hardware architectures often associated with specialized real time systems, without any performance loss. Furthermore, by exploiting the RTAI specific LXRT environment, a Linux kernel module allowing to build hard real time applications in user space, there is no need to be involved in any relatively risky kernel module application development. After a brief introduction of the basics for hard real time control in user space and of the design philosophy followed to pursue this project, a few sample applications are presented to demonstrate the ease of use and the performances that can be achieved within the RTAI/LXRT environment.

1 Introduction

The RTAI (Real Time Application Interface, for Linux) project was born at the Dipartimento di Ingegneria Aerospaziale, Politecnico di Milano (DI-APM) as a tool to develop lowcost, effective, PC based complex real time digital control systems [1]. Everything started from the need to have at hand an effective environment for implementing such control systems and carry out real time data acquisition and simulations, with an without hardware in the loop, at the lowest possible cost. Nowadays the project has already reached a significant level of maturity, and has been successfully used in numerous applications [2, 3, 4]. Anyway, until now, the development of the executable code implementing a digital control system required good C/C++ programming skills and, sometimes, also a deep knowledge of the numerical methods involved. An interesting way to overcome such a hurdle is offered by the MATLAB/Simulink/Real-Time-

Workshop suite [5, 6, 7], or simply RTW in the followings. 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 (GUI). The primary advantage of its use lies in the familiar mathematical notation used to express the problem at hand. Furthermore, being Simulink a package of the MATLAB problem solving environment, it shares the same straightforward integration of computation, monitoring and visualization, thus allowing an easy display of any result of a concurrently running parent simulation. By using this tool the user can concentrate on modeling and control issues, as opposed to programming issues. It is then RTW that controls the translation of the blocks in a series of C functions than can be compiled and linked to obtain an executable file. The usual learning time associated with a new software is virtually eliminated, meaning both

a training time/cost saving and the possibility of focusing on more sophisticated issues regarding control strategies. However the primary motivation for this project has been its use in control systems teaching in engineering courses, mated to control laboratory practice as an important part of their syllabus.

The basic choice made while building this new application has been to use what already found in the distributed RTW as it is. So it was decided to develop a hard real time, i.e. providing a strict deterministic response, user space porting under the supervision of RTAI-LXRT [2]. In such a way it has also been possible to avoid duplicating Roberto Bucher's work on the same stuff for RTAI in kernel space [8]. LXRT is an easier, less risky and faster environment for real time applications in user space, allowing a user to develop and test his/her applications without going into the kernel. RTW allows to generate different running codes, depending on the user request to have a code for the tuning and monitoring of the process under control, or a code that can run stand-alone. In the first case is necessary to have a way to exchange informations between the controlling task and Simulink. The solution available in RTW uses TCP/IP sockets for local/remote communications from Simulink to the running code, and vice versa, both to perform on-line parameters tuning during real-time executions and to monitor the behavior of any signal. With LXRT available the easiest choice has been to use the Tornado/VxWorks support distributed with MATLAB. Applying a few simple compatibilities wrappers to the Tornado/VxWorks RTW interface, natively found in MATLAB, it has been possible to use it unchanged [7]. The resulting code allows to exploit all the capabilities of the RTW suite without any added burden. Thus during the design and tuning phase, the generated control code can be run under the supervision of Simulink both locally, meaning with Simulink running on the same PC that is in charge of the controlling process, or remotely. When the developed application fits the user's needs, a stand alone code can be generated, thus getting rid of all socket communications and having available a software that can run correctly on an embedded system.

2 Design Considerations

As already said, the main goal sought by this project has been to improve the ability to create and test a complex digital control system for people that are not acquainted with programming, real time, kernel patching and so on. The aim was a typical student

of a medium to advanced level control course, trying to check his theoretical knowledge and learning from experimental activities in a control laboratory. To make it possible to concentrate on the task of experimenting different control laws, getting a real feel on how all the stuff works, is very helpful to have available a tool that hides all the complexities related to computer software and hardware. Last, but far from least, another requirement is the achievement of the lowest possible cost.

The very first component needed to build this kind of tool is a real time OS. The main problem is that usually this softwares are commercial, proprietary and often very expensive. A great cost/performance advantage is offered by RTAI, a modification and an enhancement of Linux that makes it capable of handling time critical tasks in a predictable way. In fact, RTAI is LGPL and is freely distributed, as much as Linux is. Once you have a suitable OS you have to build the programs, usually by means of C, to be ran on the PC in charge of controlling a process. To do so, the user needs to understand how to deal with operating system calls, create, delete, make, monitor and synchronize real time tasks activities. Usually all this process will require much more time to the control lab student than that effectively given to him/her.

One of the main advantages of RTW consists in its fully configurable code generator called Target Language Compiler (TLC), that specifies how to transform a Simulink blocks model into a C code, allowing to produce a controlling software for virtually any OS and platform on which MATLAB can be run, assuming you build the correct interface files. Furthermore, the standard Graphical User Interface (GUI) provided allows to start and stop the controlling software, perform parameters tuning by modifying them on the fly, and monitor the process by scope blocks, both locally or remotely.

Here it follows the list of hardware and software required for implementing control systems using LXRT and RTW:

- a standard host PC;
- a data acquisition board (DAQ);
- Linux operating system;
- RTAI Linux patch;
- MATLAB version 6.0 Release 12, with Simulink.
- Real Time Workshop.

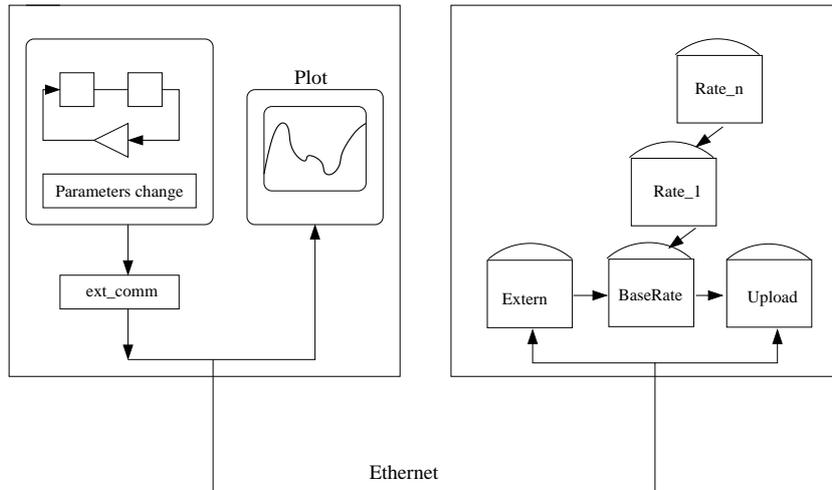


FIGURE 1: *Runtime Code Structure*

To use a specific DAQ board is necessary to have an adapted driver. Right now only a few board drivers are distributed with the RTAI-RTW interface. Anyway a new driver can be easily built following the examples already available in the package. In fact the style adopted, i.e. simple inlines and defines, to be used directly in the related MATLAB drivers, simplify the work of adapting/changing it for any board at hand.

The costlier parts of the presented list are obviously its hardware components and RTW. The latter can be really expensive for non academic applications, but has the advantage of being an extension module of a software widely used within the control systems community, i.e. MATLAB. We are currently following the evolution of the INRIA open source environment competing with MATLAB, called SCILAB [9], and are ready to move to it when it will have an easy to use automatic code generator for its Simulink peer, so that it could be possible to make available a truly free computer assisted control system design (CACSD) environment.

3 LXRT: mimicking RTW-VxWorks interface in RTAI

As already hinted the easier way devised to build an interface between RTW and RTAI, was to reproduce in the RTAI environment what another OS system, Tornado/VxWorks in our case, is able to do. In this way it has been possible to use the VxWorks support, as currently distributed natively within RTW, without any change, with the added benefit of showing how RTAI can easily do also what such an OS does. The VxWorks interface creates the following

tasks that need to run on the real time platform: (i) a certain number of *Rate* tasks that effectively perform the control functions (one or more depending if the model will be compiled as Singletasking or Multitasking [7]). (ii) The *Extern* task that implements the server side of a socket stream connection, used to receive data transferred from Simulink to the real time process; it runs at a lower priority than *Rate* tasks, and allows to update parameters without recompiling the executable code. The parameter update is obviously left to a nonblocking, and non real time, task. (iii) The *Upload* task to enable data collection and display, again by using a socket stream. Data are collected at every base time step and then sent to the host for display when there is time left, that means when there is spare time before the next time step. The execution of all the tasks is controlled by a bunch of semaphores. When a timer interrupt occurs, the interrupt function service calls `semGive`, which awakes a task blocked on a `semTake` routine at the end of previous time step. The host side that communicates with the real-time program is Simulink running in external mode. Using its GUI the user can send new parameters to the real time controller and plot real time histories of the retrieved model data.

To implement the aforementioned architecture within the RTAI environment it is necessary to have access to both Linux (to operate the TCP/IP communications) and RTAI services symmetrically. This capability is offered by LXRT, the module to implement RTAI hard real time tasks in user space. With LXRT you can freely access all RTAI services, meaning about 156 functions for timing, semaphores, messaging and so on, executing both in soft and/or hard real time. Hard real time LXRT in user space allows

full kernel preemption, with the only penalty of a very slight increase in overhead, jitter and latency remaining very close to those measurable for the same applications implemented in kernel space. The only constraint left is that you cannot use Linux kernel services directly. Anyway, thanks to the many inter-task communication mechanisms made available by RTAI, it is trivial to mate each hard real time process to a Linux server that takes up all the kernel services on behalf of its hard real time master. Interestingly enough the same kind of policy is natively used by RTW with VxWorks. That shows that the interaction with non truly real time functionalities should never be performed within any hard real time function, even if the underlying OS allows it, thus turning what some people see as an RTAI/LXRT constraint into a better and safer hard real time programming style.

To access RTAI services from LXRT, a Linux process needs to create a real time kernel task, called the buddy/proxy that is in charge of the execution of real time services, with a call to `rt_task_init()`. Afterward it is possible to start a timer, wait on a semaphore, send intertask messages and so on. To delete the buddy, a simple call to `rt_task_delete()` is necessary. A group of simple interface `.h` files has been written to mimic the VxWorks APIs used in RTW for Tornado/VxWorks. The only module that has been implemented anew has been the one to support the managing of tasks woken up by a timer interrupt, either external or internal. Within RTAI such a module would not be strictly necessary, as it could be easily implemented in many other ways, e.g. trivially by using the timer module found in the user space `tasklets` module, or by setting up a simple periodic parent timing task in kernel space that executes the required semaphore signaling. However a direct implementation in a timer interrupt handler is simpler, more effective and can be easily used with any external timer made available by most DAQ boards. LXRT processes can be run either in hard or soft real time mode; moving from one configuration to the other involves just a call to `rt_make_hard_real_time()`, or vice versa to `rt_make_soft_real_time()`. However to switch between the two a user does not have to know them, but just set a parameter in the execution command line. To clarify the difference, it must be remarked that hard real time tasks and interrupt 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 tasks in kernel space and hard real time processes of higher priority.

Two additional tasks are created in the LXRT-RTW

interface: a mailbox to handle real time log messages that are eventually emitted by the real time tasks, and an *Overrun Monitor*. The latter is used to keep track of the overruns that can eventually happen, since the execution won't stop when an overrun occurs, a possibility left to the operator monitoring the control system. This feature has been provided because sometimes several overrun can happen at the beginning of the control activity under very fast timing, an occurrence that does not affect the correct behavior of the control code in the following phases. LXRT implementation enables any user to freely access hard I/O. So the control system designer need not to be a super user to develop and use these applications; however root permission is still required to install the needed RTAI support modules. In this way an inexperienced user can freely play with the applications without compromising the security of the installed software, an aspect that can be extremely important in a university lab.

4 Building a real-time controller

To set up a new control device the first step is the creation of the Simulink block diagram that represent the algorithm to be used to implement the chosen control strategy. After some simple tests, it is possible to add the DAQ blocks and start the code generation phase. The process is straightforward; the user needs only to choose the right template makefile for TLC, and decide if he/she wants to use the *External Mode*, meaning the mode where there is an exchange of information between Simulink and the code while the process is running and/or the *Data Log Mode*, where the controller runs in a stand alone mode and all the scope signals are just written to a log file that can be subsequently loaded and visualized in MATLAB. If there are different sampling times in the Simulink model the user can generate either a Singletasking or a Multitasking code. The latter consists in creating a task for each sampling time present in the model. The highest priority task is the *Base Rate*, which executes the components of the model code run at the highest sample rate. For each additional sample rate in the system, a separate task, with lower priority, is spawned: *Rate 1*, *Rate 2*, ..., *Rate n*. Those who have read some textbooks on OSes will promptly understand that RTW adopts a Rate Monotonic Scheduling (RMS) policy for its multitasking implementations. TLC creates the source files and the makefile, then proceeds with the compilation process. Before running the executable file to start the control process, it is necessary

to load the required RTAI modules, an operation that is performed by a simple script file. The user can then choose between a soft or a hard real time execution by a simple command line option. The Overrun Monitor will show the overrun frequency, with an assigned time step, if there is any overrun at all. Clearly the soft mode is used only during development to avoid crashes, e.g. due to too demanding timings. After this testing and tuning phase the user can move to the generation of the code in *Standalone* mode, if he/she is planning to use the control device in an embedded system.

5 A simple example

As a first example we present the simple model shown in Figure 2 [8]. A discrete sine signal is generated and output directly to the first channel of the D/A converter. The very same signal goes through a discrete time integrator block, a random noise is added and the resulting signal sent to a second channel of the D/A converter. These two signals are then acquired back by the A/D converter and plotted by means of a scope block. The DAQ board used is a low cost Intelligent Instrumentation model. Even though the proposed scheme is fairly simple, it presents some floating point operations per cycle, mostly related to the random noise generator.

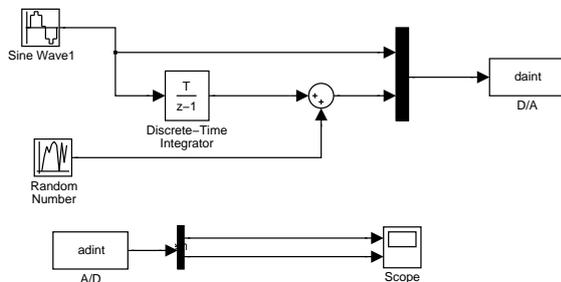


FIGURE 2: *Simulink Block Diagram*

Figure 3 shows a plot of the two signals acquired, when the sine wave and the random signal are generated with a sample rate of 1 KHz. Figure 4 instead shows what happened when we changed on the fly the frequency and the amplitude of the sine wave: from 5 Volt at 10 Hz, to 3 Volt at 30 Hz. All the tests have been ran on an old vintage 200 MHz PentiumPro, while Simulink was running on a different PC. We have increased sampling frequency up to 10 KHz without incurring in any significant overrun other than a few at the very beginning. To test also the Multitasking capabilities, another application has been carried out by lowering the acquisition frequency to 5 KHz, while keeping the main sampling frequency at 10 KHz. Again no problems have been

encountered. The robustness of the system has been proved by running it for over 3 hours continuously. It must be noticed that both applications sent data to be displayed remotely at the very same rate of generation, thus causing a high number of background interrupts on the control computer, without causing any problem to the hard real time activity.

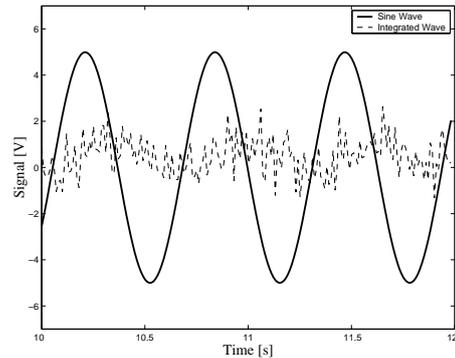


FIGURE 3: *Plot of scope blocks*

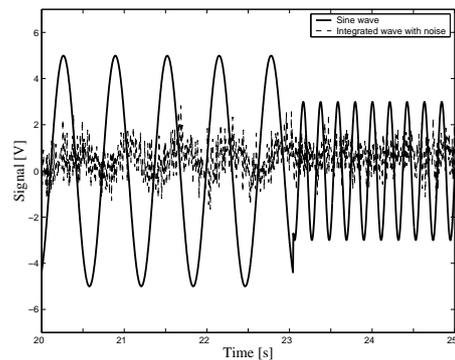


FIGURE 4: *On the fly parameter tuning*

To get a deeper insight on the performances, we have tested another Simulink model made by 9 different sine waves that are sent directly into scope blocks, each one with a different sampling time. The test has been carried out on an SMP Dual PentiumII 350 MHz. The program spawned 14 tasks on both CPU running successfully with the higher sampling rate of 2 KHz. The high number of interrupts that the numerous data exchange required on the control machine, created an intensive overrun during the start up phase, after which the program ran without any significant problems.

Currently the project has been already exploited for the generation of digital control devices for vibration reduction, inverse pendulums, photovoltaic cells, positioning of flexible structures and so on. These results are not shown here for the sake of brevity.

6 Conclusions

A useful application to design and implement real time digital control systems under a freeware environment, without requiring any special programming skill, has been described. It is a very flexible tool affording solutions that can be successfully applied in real word application, being able to sustain a control at 10 KHz on a vintage machine as a PentiumPro, without requiring to go into kernel space. We demonstrated in this way that the overhead, which necessarily comes from generating a code by means of an automatic coding tool such as RTW, should not be an obstacle as far as the performance required are of the order of magnitude presented here.

References

- [1] D. Beal, E. Bianchi, L. Dozio, S. Hughes, P. Mantegazza, and S. Papacharalambous. RTAI: Real time application interface. *Linux Journal*, April 2000.
- [2] E. Bianchi and L. Dozio. Some experience in fast hard real-time control in user space with RTAI-LXRT. In *Real Time Linux Workshop*, Orlando, FL, 2000.
- [3] E. Bianchi, L. Dozio, P. Mantegazza, and G. L. Ghiringhelli. Complex control system, application of DIAPM-RTAI at DIAPM. In *Real Time Linux Workshop*, Vienna, Austria, 1999.
- [4] E. Bianchi, L. Dozio, D. Martini, and P. Mantegazza. Applications of a hard real-time support in digital control of complex aerospace systems. In *AIDAA Congress*, Torino, Italy, 1999.
- [5] MATLAB User's Guide. *The Math Works Inc.*, September 2000.
- [6] Simulink User's Guide. *The Math Works Inc.*, September 2000.
- [7] Real-Time Workshop User's Guide. *The Math Works Inc.*, September 2000.
- [8] <http://A.die.supsi.ch/~bucher>
- [9] <http://www-rocq.inria.fr/scilab>