Xenomai - Implementing a RTOS emulation framework on GNU/Linux

Philippe Gerum

First Edition

April 2004

Abstract

Generally speaking, the Xenomai technology first aims at helping application designers relying on traditional RTOS to move as smoothly as possible to a GNU/Linux-based execution environment, without having to rewrite their application entirely.

This paper discusses the motivations for proposing this framework, the general observations concerning the traditional RTOS directing this technology, and some in-depth details about its implementation.

The Xenomai project has been launched in August 2001. It has merged in 2003 with the RTAI project [http://www.gna.org/projects/rtai/] to produce an industrial-grade real-time Free Software platform for GNU/Linux called RTAI/fusion, on top of Xenomai's abstract RTOS core. Eventually, the RTAI/fusion effort became independent from RTAI in 2005 as the xenomai project [http://www.gna.org/projects/xenomai/].

Table of Contents

1. White paper

1.1. Introduction

A simpler migration path from traditional RTOS to GNU/Linux can favour a wider acceptance of the latter as a real-time embedded platform. Providing emulators to mimic the traditional RTOS APIs is one of the initiative the free software community can take to fill the gap between the very fragmented traditional RTOS world and the GNU/Linux world, in order for the application designers relying on traditional RTOS to move as smoothly as possible to the GNU/

There is a lack of common software framework for developing these emulators, whereas the behavioural similarities between the traditional RTOS are obvious.

The Xenomai technology aims at fulfilling this gap, by providing a consistent architecture-neutral and generic emulation layer taking advantage of these similarities. It is also committed to provide an increasing set of traditional RTOS emulators built on top of this layer.

Xenomai relies on the common features and behaviours found between many embedded traditional RTOS, especially from the thread scheduling and synchronization standpoints. These similarities are exploited to implement a nucleus exporting a set of generic services. These services grouped in a high-level interface can be used in turn to implement emulation modules of real-time application programming interfaces, which mimic the corresponding real-time kernel APIs.

A similar approach was used for the CarbonKernel [http://savannah.gnu.org/projects/carbonkernel/] project [1] in the simulation field, in which RTOS simulation models are built on top of a generic virtual RTOS based on event-driven techniques.

1.2. Porting traditional RTOS-based applications to GNU/Linux

The idea of using GNU/Linux as an embedded system with real-time capabilities is not novel. The reader can refer to Jerry Epplin's article in the October 97 issue of Embedded Systems Programming for a discussion about GNU/Linux potential in the embedded field [2].

Throughout this document, we will use the expression *source RTOS* to indicate the traditional real-time operating from which the application is to be ported, and *target OS* to indicate GNU/Linux or any other free operating system to which the application could be ported.

1.2.1. Limited high-level code modification

Keeping the initial design and implementation of a hard real-time application when attempting to port it to another architecture is obviously of the greatest interest. Reliability and performance may have been obtained after a long, complex and costly engineering process one does not want to compromise. Consequently, the best situation is to have the closest possible equivalence between the source and destination RTOS programming interfaces, as far as both the syntax and the semantics are concerned.

An illustration of this can be taken from the support of a priority inheritance protocol [3] by the mutual exclusion services. These services allow concurrent threads to protect themselves from race conditions that could occur into critical sections of code. The purpose of this discussion is not to argue whether relying on priority inheritance for resolving priority inversion problems is a major design flaw or a necessary safety belt for a real-time application, but only to emphasize that any cases, if this feature is used in the source RTOS, but not available from the target OS, the resource management strategy must be reevaluated for the application, since priority inversion risks will exists.

1.2.2. RTOS behavioural compatibility

During the past years, major embedded RTOS, such as VRTX, VxWorks, pSOS+ and a few others, have implemented a real-time kernel behaviour which has become a de facto standard, notably for thread scheduling, inter-thread synchronization, and asynchronous event management. To illustrate this, let us talk about a specific concern in the interrupt service management.

A well-known behaviour of such RTOS is to lock the rescheduling procedure until the outer interrupt service routine (or ISR) - called first upon possibly nested interrupts - has exited, after which a global rescheduling is finally stated. This way, an interrupt service routine can always assume that no synchronous thread activity may run until it has exited. Moreover, all changes impacting the scheduling order of threads, due to actions taken by any number of nested ISRs (e.g. signaling a synchronization object on which one or more threads are pending) are considered once and conjunctively, instead of disjunctively.

For instance, if a suspended thread is first resumed by an ISR, then forcibly suspended later by another part of the same ISR, the outcome will be that the thread will not run, and remain suspended after the ISR has exited. In the other hand, if the RTOS sees ISRs as non-specific code that can be preempted by threads, the considered thread will be given the opportunity to execute immediately after it is resumed, until it is suspended anew. Obviously, the respective resulting situations won't be identical.

1.2.3. Reevaluation of the real-time constraints

Making GNU/Linux a hard real-time system is currently achieved by using a co-kernel approach which takes control of the hardware interrupt management, and allows running real-time tasks seamlessly aside of the hosting GNU/Linux system [4]. The 'regular' Linux kernel is eventually seen as a low-priority, background of the small real-time executive. The RTLinux [http://www.rtlinux.org] project is representative of this technical path. However, this approach has a major drawback when it comes to port complex applications from a foreign software platform: since the real-time tasks run outside the Linux kernel control, the GNU/Linux programming model cannot be preserved when porting these applications. The result is an increased complexity in redesigning and debugging the ported code.

In some cases, choosing a traditional RTOS to run an embedded application has been initially dictated by the memory constraints imposed by the target hardware, instead of actual real-time constraints imposed by the application itself. Since embedded devices tend to exhibit ever increasing memory and processing horsepower, it seems judicious to reevaluate the need for real-time guarantee when considering the porting effort to GNU/Linux on a new target hardware. This way, the best underlying software architecture can be selected. In this respect, the following, the following criteria need to be considered:

- *Determinism and criticality*.

  What is the worst case interrupt and dispatch latencies needed ?

  Does a missed deadline lead to a catastrophic failure ?

- *Programming model*

  What is the overall application complexity, provided that the higher the complexity, the greater the need for powerful debugging aid and monitoring tools.

- Is there a need need for low-level hardware control ?

  Is the real-time activity coupled to non real-time services, such as GUI or databases, requiring sophisticated communications with the non real-time world ?

1.2.4. Some existing solutions

In order to get whether hard or soft real-time support, several GNU/Linux-based solutions exist [5][6]. It is not the purpose of this paper to present them all exhaustively. We will only consider a two fold approach based on free software solutions which is likely to be suited for many porting taskings, depending on the actual real-time constraints imposed by the application.

1.2.4.1. Partial rewriting using a real-time GNU/Linux extension

Real-time enabling GNU/Linux using RTAI. Strictly speaking Linux/RTAI [7] is not a real-time operating system but rather a real-time Linux kernel extension, which allows running real-time tasks seamlessly aside of the hosting GNU/Linux system. The RTAI co-kernel shares hardware interrupts and system-originated events like traps and faults with the Linux kernel using the Adeos [http://www.adeos.org/] virtualization layer, which in turn ensures RTAI low interrupt latencies. The native RTAI API provides the applications a wealth of useful services, including counting semaphores, POSIX 1003.1-1996 facilities such as pthreads, mutexes and condition variables, also adding remote procedure call facility, mailboxes, and precision timers. Most services are symmetrically available from kernel module and user-space programs.

RTAI 2.x and 3.x provide a means to execute hard real-time tasks in user-space context (x86 only), but still outside the Linux kernel control, which is best described as running 'user-space kernel modules'. This feature, namely LXRT, is a major step toward a simpler migration path from traditional RTOS, since programming errors occuring within real-time tasks don't jeopardize the overall GNU/Linux system sanity, at the expense of a few microseconds more latency.

Ad hoc services emulation. A first approach consists in emulating each real-time facility needed by the application using a combination of the RTAI services. An ad hoc wrapping interface has to be written to support the needed function calls. The benefit of the wrapping approach lies in the limited modifications made to the original code. However, some RTAI behaviours may not be compliant with the source operating system's. For the very same reason, conflicts between the emulated and native RTAI services may occur in some way.

Complete port to RTAI. A second approach consists in fully porting the application over the native RTAI API. In such a case, RTAI facilities are globally substituted from the facilities from the source RTOS. This solution brings improved consistency at the expense of a possible large-scale rewriting of the application, due to some fundamental behavioural differences that may exist between the traditional RTOS and the native RTAI interface.

1.2.4.2. Unconstrained user-space emulations

A few traditional RTOS emulators exists in the free software world. There are generally designed on top of the GNU/Linux POSIX 1003.1-1996 layer, and allow to emulate the source RTOS API in a user-space execution context, under the control of the Linux kernel.

Once one of the most proeminent effort in this area used to be the Legacy2linux project [8]. This project, sponsored by Montavista Software, aimed at providing ["a series of Linux-resident emulators for various legacy RTOS kernels."] Just like Xenomai, [these emulators are designed to ease the task of porting legacy RTOS code to an embedded Linux environment".] Two emulators have been made available by this project, respectively mimicking the APIs of WindRiver's pSOS+ and VxWorks real-time operating systems. However, this project has stalled due to a lack of maintenance and contribution.

The benefits of this approach is mainly to keep the development process in the GNU/ Linux user-space environment, instead of moving to a rather 'hostile' kernel/supervisor mode context. This way, the rich set of existing tools such as debuggers, code profilers, and monitors usable in this context are immediatly available to the application developer. Moreover, the standard GNU/Linux programming model is preserved, allowing the application to use the full set of of facilities existing in the user space (e.g. full POSIX support, including inter-process communication). Last but not least, programming errors occuring in this context don't jeopardize the overall GNU/Linux system stability, unlike what can happen if a bug is encountered on behalf of a hard real-time RTAI task which could cause serious damages to the running Linux kernel.

However, we can see at least three problems in using these emulators, depending on the application constraints:

- First, the emulated API they provide is usually incomplete for an easy port from the source RTOS. In other words, only a limited syntactic compatibility is available.

- Second, the exact behaviour of the source RTOS is not reproduced for all the functional areas. In other words, the semantic compatibility might not be guaranteed.

- These emulators don't share any common code base for implementing the fundamental real-time behaviours, even so both pSOS+ and VxWorks share most of them. The resulting situation leads to redundant implementation efforts, without any benefit one can see in code mutualization.

- And finally, even combined to the latest Linux 2.6 features like fine-grain kernel preemption and low latency efforts, these emulators cannot deliver deterministic real-time performance.

1.3. A common emulation framework

1.3.1. Common traditional RTOS behaviours

In order to build a generic and versatile framework for emulating traditional RTOS, we chose to concentrate on a set of common behaviours they all exhibit. A limited set of specific RTOS features which are not so common, but would be more efficiently implemented into the nucleus than into the emulators, has also been retained. The basic behaviours selected cover four distinct fields:

1.3.1.1. Multi-threading

Multi-threading provides the fundamental mechanism for an application to control and react to multiple, discrete external events. The nucleus provides the basic multi-threading environment.

Thread states. The nucleus has to maintain the current state of each thread in the system. A state transition from one state to another may occur as the result of specific nucleus services called by the RTOS emulator. The fundamental thread states defined by the nucleus are:

- DORMANT and SUSPENDED states are cumulative, meaning that the newly created thread will still remain in a suspended state after being resumed from the DORMANT state.

- PENDING and SUSPENDED states are cumulative too, meaning that a thread can be forcibly suspended by another thread or service routine while pending on a synchronization resource (e.g. semaphore, message queue). In such a case, the resource is dispatched to it, but it remains suspended until explicitly resumed by the proper nucleus service.

- PENDING and DELAYED states may be combined to express a timed wait on a resource. In such a case, the time the thread can be blocked is bound to a limit enforced by a watchdog.

Scheduling policies. By default, threads are scheduled according to a fixed priority value, using a preemptive algorithm. There is also a support for round-robin scheduling among a group of threads having the same priority, allowing them to run during a given time slice, in rotation. Moreover, each thread undergoing the round-robin scheduling is given an individual time quantum.

Priority management. It is possible to use either an increasing or decreasing thread priority ordering, depending on an initial configuration. In other words, numerically higher priority values could either represent higher or lower scheduling priorities depending on the configuration chosen. This feature is motivated by the existence of this two possible ordering among traditional RTOS. For instance, VxWorks, VRTX, ThreadX and Chorus O/S use a reversed priority management scheme, where the higher the value, the lower the priority. pSOS+ instead uses the opposite ordering, in which the higher the value, the higher the priority.

Running thread. At any given time, the highest priority thread which has been ready to run for the longest time among the currently runnable threads (i.e. not currently blocked by any delay or resource wait) is elected to run by the scheduler.

Preemption. When preempted by a higher priority thread, the running thread is put at the front of the ready thread queue waiting for the processor resource, provided it has not been suspended or blocked in any way. Thus it is expected to regain the processor resource as soon as no other higher priority activity (i.e. a thread having a higher priority level, or an interrupt service routine) is eligible for running.

Manual round-robin. As a side-effect of attempting to resume an already runnable thread or the running thread itself, this thread is moved at the end of its priority group in the ready thread queue. This operation is functionally equivalent to a manual round-robin scheduling.

Even if they are not as widespread as those above in traditional RTOS, the following features are also retained for the sake of efficiency in the implementation of some emulators:

Priority inversion. In order to provide support for preventing priority inversion when using inter-thread synchronization services, the priority inheritance protocol is supported.

Signaling. A support for sending signals to threads and running asynchronous service routines to process them is available. The asynchronous service routine is run on behalf of the signaled thread context the next time it returns from the nucleus level of execution, as soon as one or more signals are pending.

Disjunctive wait. A thread is able to wait in a disjunctive manner on multiple resources. The nucleus unblocks the thread when at least one of the pending resources is available.

1.3.1.2. Thread synchronization

Traditional RTOS provide a large spectrum of inter-thread communication facilities involving thread synchronization, such as semaphores, message queues, event flags or mailboxes. Looking at them closely, we can define the characteristics of a basic mechanism which will be usable in turn to build these facilities.

Pending mode. The thread synchronization facility provides a means for threads to pend either by priority or FIFO ordering. Multiple threads should be able to pend on a single resource.

Priority inheritance protocol. In order to prevent priority inversion problems, the thread synchronization facility implements a priority inheritance protocol in conjunction with the thread scheduler. The implementation allows for supporting the priority ceiling protocol as a derivative of the priority inheritance protocol.

Time-bounded wait. The thread synchronization facility provides a means to limit the time a thread waits for a given resource using a watchdog.

Forcible deletion. It is legal to destroy a resource while threads are pending on it. This action resumes all waiters atomically.

1.3.1.3. Interrupt management

Since the handling of interrupts is one of the least well defined areas in RTOS design, the nucleus focuses on providing a simple mechanism with sufficient hooks for specific implementations to be built onto according to the emulated RTOS flavour.

Nesting. Interrupt management code is reentrant in order to support interrupt nesting safely.

Atomicity. Interrupts are associated with dedicated service routines called ISRs. In order for these routines not to be preempted by thread execution, the rescheduling procedure is locked until the outer ISR has exited (i.e. in case of nested interrupts).

Priority. ISRs are always considered as priority over thread execution. Interrupt prioritization is left to the underlying hardware.

1.3.1.4. Time management

Traditional RTOS usually represent time in units of ticks. These are clock-specific time units and are usually the period of the hardware timer interrupt, or a multiple thereof. Since it needs to support both periodic and aperiodic time sources, the nucleus transparently switches from periodic jiffies to time-stamp counter values depending on the current timer operating mode.

Software timer support. A watchdog facility is provided to manage time-bound operations by the nucleus.

Absolute and relative clock. The nucleus keeps a global clock value which can be set by the RTOS emulator as being the system-defined epoch.

Some RTOS like pSOS+ also provide support for date-based timing, but conversion of ticks into conventional time and date units is an uncommon need that should be taken in charge by the RTOS emulator itself.

1.3.2. An architecture-neutral abstraction layer

After having selected the basic behaviours shared by traditional RTOS, we have implemented them in a nucleus exporting a few service classes. These generic services will then serve as a founding layer for developing each emulated RTOS API, according to their own flavour and semantics.

In order for this layer to be architecture neutral, the needed support for hardware control and real-time capabilities will be obtained from an underlying host software architecture, through a rather simple standardized interface. Thus, porting the nucleus to a new real-time architecture will solely consist in implementing this low-level interface for the target platform.

1.3.3. Real-time capabilities

The host software architecture is expected to provide the primary real-time capabilities to the RTOS abstraction layer. Basically, the host real-time layer must handle at least the following tasks:

- On request start/stop dispatching the external interrupts to a specialized handler ;
- Provide a means to mask and unmask interrupts ;
- Provide a means to create new threads of control in their simplest form ;
- Provide support for periodic and aperiodic interrupt sources used in timer management ;
- Provide support for allocating chunks of non-pageable memory.

1.3.4. Benefits

Xenomai aims at helping application designers relying on traditional RTOS to move as smoothly as possible to a GNU/Linux-based execution environment, without having to rewrite their applications entirely. Aside of the advantages of using GNU/Linux as an embedded system, the benefits expected from the described approach is mainly a reduced complexity in designing new RTOS emulations. The architecture-neutral abstraction layer provides the foundation for developing accurate emulations of traditional RTOS API, saving the burden of repeatedly implementing their fundamental real-time behaviours. Since the abstraction layer also favours code sharing and mutualization, we can expect the RTOS emulations to take advantage of them in terms of code stability and reliability.

1.4. Nucleus description

RTOS emulations are software modules which connect to the nucleus through the pod abstraction. The pod is responsible for the critical housekeeping chores, and the real-time scheduling of threads.

1.4.1. Multi-threading support

The nucleus provides thread object (xnthread) and pod (xnpod) abstractions which exhibit the following characteristics:

- Threads are scheduled according to a 32bit integer priority value, using a preemptive algorithm. Priority ordering can be increasing or decreasing depending on the pod configuration.
- A thread can be either waiting for initialization, forcibly suspended, pending on a resource, delayed for a count of ticks, ready-to-run or running.
- Timed wait for a resource can be bounded by a per-thread watchdog.
- The priority inheritance protocol is supported to prevent thread priority inversion when it is detected by a synchronization object.
- A group of threads having the same base priority can undergo a round-robin scheduling, each of them being given an individual time quantum.
- A support for sending signals to threads and running asynchronous service routines (ASR) to process them is built-in.
- FPU support can be optionally enabled or disabled for any thread at creation time.
- Each thread can enter a disjunctive wait on multiple resources.

1.4.2. Basic synchronization support

The nucleus provides a synchronization object abstraction *(xnsynch)* aimed at implementing the common behaviour of RTOS resources, which has the following characteristics:

- Support for the priority inheritance protocol, in order to prevent priority inversion problems. The implementation is shared with the scheduler code.
- Support for time-bounded wait and forcible deletion with waiters awakening.

1.4.3. Timer and clock management

The nucleus needs a time source to provide the time-related services to the upper interfaces. The timer hardware needs to be configured so that a user-defined routine is called according to a given frequency. On architectures that provide a oneshot-programmable time source, the system timer can operate either in aperiodic or periodic mode. Using the aperiodic mode still allows to run periodic nucleus timers over it: the underlying hardware will simply be reprogrammed after each tick by the timer manager using the appropriate interval value.

Each incoming clock tick is announced to the timer manager which fires in turn the timeout handlers of elapsed timers. The scheduler itself uses per-thread watchdogs to wake up threads undergoing a bounded time wait, while waiting for a resource availability or being delayed.

A special care has been taken to offer bounded worst-case time for starting, stopping and maintaining timers. The timer facility is based on the timer wheel algorithm[11] described by Adam M. Costello and George Varghese, which is implemented in the NetBSD operating system for instance.

1.4.4. Basic memory allocation

Xenomai's nucleus provides dynamic memory allocation support with real-time guarantee, based on McKusick's and Karels' proposal for a general purpose memory allocator[10]. Any number of memory heaps can be maintained dynamically by Xenomai, only limited by the actual amount of system memory.