# The XENOMAI Project

**Implementing a RTOS emulation framework on GNU/Linux.**

**Philippe Gerum**

IDEALX, Open Source Engineering

15-17 avenue de Segur, Paris, France

rpm@idealx.com

## Abstract

Xenomai is a GNU/Linux-based framework which aims at being a foundation for a set of traditional RTOS API emulators running on top of a host software architecture, such as RTAI when hard real-time support is required.

Generally speaking, this project 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.

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

The Xenomai project has been launched in August 2001. It is hosted at http://freesoftware.fsf.org/projects/xenomai/.

Linux is a registered trademark of Linus Torvalds. Other trademarks cited in this paper are the property of their respective owners.

# 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 a GNU/Linux real-time environment.

There is a lack of common software framework for developing these emulators, whereas the behavioural similarities between the traditional RTOS are obvious, most variations being only different "window dressings" of well-known operating system concepts (multi-threading, synchronization etc).

The Xenomai project aims at fulfilling this gap. It aims at providing a consistent framework that helps implementing real-time interfaces and debugging real-time software on GNU/Linux. Xenomai comes with a growing set of emulators of traditional RTOS APIs, that ease the migration of applications from these systems to a GNU/Linux-based real-time environment.

The Xenomai project 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 nanokernel exporting a set of generic services. These services grouped in a high-level interface can be used in turn to implement real-time interfaces such as emulation modules of traditional RTOS which mimic the emulated real-time kernel APIs.

A similar approach was used for the 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.

# 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].
Throughough this document, we will use the expression *source RTOS* to indicate the traditional real-time operating system 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.

### 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.

For instance, if the application needs dynamic memory allocation with success guarantee for its real-time threads (which is different from a real-time guarantee), porting it to a GNU/Linux hard real-time extension (such as RTAI or RTLinux) raises the following issues:

- Linux kernel's *kmalloc()/kfree()* services should not be called on behalf of a real-time thread, since these services are not reentrant. Consequently, the needed memory has to be pre-allocated statically or during the application startup, on behalf of the Linux kernel context.
- A dynamic allocator callable from a real-time context is provided by RTAI (i.e. *rt_mem_mgr*), but its services are based on an algorithm *anticipating* the memory starvations using an asynchronous pre-allocation technique, but *not guaranteeing* that no failure could occur. To give a reasonable guarantee of success in allocating memory blocks, i.e. to be sure that valid memory will always be returned to the real-time thread as soon as it is available from the Linux kernel, the calling thread should be put in a wait state until the memory it has requested is available.

In both cases, it may be necessary to adapt the memory management strategy according to these constraints, which could be a quite difficult and error-prone task.

Another example 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 in 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 exist.

**2.2. RTOS behavioral compatibility**

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

A well-known behavior 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 started. 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 the 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.

**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 task of the small real-time executive. The RTAI (http://www.rtai.org/) and RTLinux (http://www.rtlinux.org/) projects are 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 progamming model cannot be preserved when porting these applications. The result is 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 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 catastophic failure ?

*Programming model.*

What is the overall application complexity, provided that the highest the complexity, the greatest the need for powerful debugging aid and monitoring tools.
Is there a 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 ?

### 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 tasks, depending on the actual real-time constraints imposed by the application.

#### 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 is hooked to the hosting system through an hardware abstraction layer (HAL) which redirects external events to it, thus ensuring low interrupt latencies. RTAI provides a fixed-priority driven scheduler to run concurrent real-time activities loaded from dynamic kernel modules. Globally-scoped scheduling decisions are made by the co-kernel which always considers the host Linux kernel as its lowest-priority thread of activity. In other words, RTAI considers the Linux kernel as a background task that should run when no real-time activity occurs, a kind of idle task for common RTOS. RTAI provides a wealth of other 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.
Moreover, RTAI provides a mean to execute hard real-time tasks in user-space context, 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 ocurring 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 behaviors 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 natively to RTAI. In such a case, RTAI facilities are globally substituted for the facilities from the source RTOS. This solution brings improved consistency at the expense of a possibly large-scale rewriting of the application, due to some fundamental behavioral differences that may exist between the traditional RTOS and RTAI.

#### 2.4.2. Unconstrained user-space emulations

A few traditional RTOS emulators exists in the free software world. They 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.

One of the most proeminent effort in this area is the Legacy2linux project [8]. This project, sponsored by Montavista Software, aims 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 are currently available from this project, respectively mimicking the APIs of WindRiver's pSOS+ and

VxWorks real-time operating systems.

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 immediately available to the application developer. Moreover, the standard GNU/Linux programming model is preserved, allowing the application to use the full set of facilities existing in the user space (e.g. full POSIX support, including inter-process communication). Last but not least, programming errors occurring 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 four 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 behavior 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 behaviors, 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 existing Linux kernel patches providing fixed-priority scheduling (Montavista's RTSched) and fine-grain kernel preemption (Ingo Molnar's Linux kernel patches for improved preemptability), these emulators cannot deliver hard real-time performance.

## 3. A common emulation framework

### 3.1. Common traditional RTOS behaviors

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

#### 3.1.1. Multi-threading

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

*Thread states.* The nanokernel 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 nanokernel services called by the RTOS emulator. The fundamental thread states that should be defined 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 explicitely resumed by the proper nanokernel 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 must also be 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 should be given an individual time quantum.

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

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

*Preemption.* When preempted by a more prioritary thread, the running thread should be put at 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 prioritary 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 should be moved at the end of its priority group in the ready thread queue. This operation should be 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 should be implemented.

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

#### 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. In looking at them closely, we can define the characteristics of a basic mecanism which will be usable in turn to build these facilities.

*Pending mode.* The thread synchronization facility should provide a mean 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 should implement a priority inheritance protocol in conjunction with the thread scheduler. The implementation should allow for supporting the priority ceiling protocol as a derivative of the priority inheritance protocol.

*Time-bounded wait.* The thread synchronization facility should provide a mean to limit the time a thread waits for a given resource using a watchdog.

*Forcible deletion.* It should be legal to destroy a resource while threads are pending on it. This action should resume all waiters atomically.

#### 3.1.3. Interrupt management

Since the handling of interrupts is one of the least well

defined areas in RTOS design, a generalized mechanism is provided with sufficient hooks for specific real-time interfaces to be built onto.

*Nesting*. Interrupt management code should be reentrant in order to support interrupt nesting safely.

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

*Priority*. ISRs should always be considered as prioritary over thread execution.

#### 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.

*Software timer support.* A watchdog facility is needed to manage time-bound operations by the nanokernel.

*Absolute and relative clock.* The nanokernel should keep 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.

### 3.2. An architecture-neutral abstraction layer

After having selected the basic behaviors shared by traditional RTOS, we can implement them in a nanokernel 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 nanokernel to a new real-time architecture will solely consist in implementing this low-level interface for the target platform.

### 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:

° Start/stop dispatching on request the external interrupts to an abstraction layer's specialized handler ;
° Provide a mean to mask and unmask interrupts ;
° Provide a mean to create new threads of control in their simplest form ;
° Provide support for a periodic interrupt source used in timer management ;
° Provide support for allocating chunks of non-pageable memory.

When the host software architecture has no direct access to the underlying hardware, such as in a soft real-time user-space execution environment, interrupts may be simulated by POSIX signals, and hard real-time constraints imposed to the services above may be relaxed (e.g. memory can be pageable).

### 3.4. Benefits

The project described herein 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 are:

*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 implementing each time their fundamental real-time behaviors. Since the abstraction layer also favours code sharing and mutualization, we can expect the RTOS emulations to take

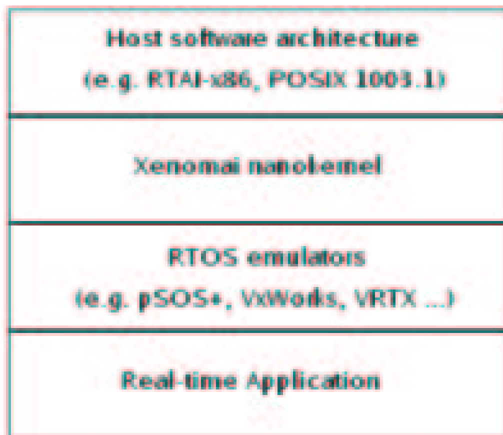
advantage of them in terms of code stability and reliability.

*Generic support for RTOS-aware tools.* One of the most potential show-stopper for a broader use of GNU/Linux in the real-time space is probably the lack of powerful and user-friendly debugging and monitoring tools for real-time applications. However, this gap is about to be filled by the maturation of tools like the Linux Trace Toolkit (LTT) [9] which now offers unprecedented capabilities for inspecting the dynamics of a running GNU/Linux system. Since a version of LTT is available for the 'regular' Linux kernel and Linux/RTAI, the next step will be to take advantage of this toolkit, implementing the proper hooks to support it into the nanokernel internals and interface, in order to provide RTOS-aware tools as soon as possible.

## 4. The Xenomai approach

### 4.1. Xenomai architecture

The common emulation framework precedently envisioned translates in the Xenomai architecture as follows:




### 4.2. Host software architecture

Xenomai's nanokernel relies on an host software architecture to provide the needed hardware control and real-time capabilities.

The nanokernel is connected to the host architecture through a standardized interface. The following services compose the nanokernel-to-real-time subsystem interface:

Depending on the execution environment, some of the above services may be emulated or simply stubbed as soon as they are not needed. However, all of them are needed for porting the nanokernel on top of RTAI. For instance, the interrupt-related services can be emulated by the POSIX signal feature when running a combination of the nanokernel, the RTOS emulator and the (soft) real-time application as a user-space GNU/Linux process. In the same spirit, the real-time context switch routines have no purpose, thus can be empty in such environment.

#### 4.2.1. Using RTAI as the host software architecture

The Real-Time Application Interface (RTAI) is a real-time GNU/Linux extension, which allows running real-time tasks seamlessly aside of the hosting GNU/Linux system. The RTAI co-kernel is hooked to the hosting system through an hardware abstraction layer (HAL). RTAI considers the Linux kernel as a background task that should run when no real-time activity occurs. RTAI applications run in supervisor mode, in the Linux kernel address space.

When running on top of RTAI, the Xenomai framework gains hard real-time capabilities, replacing the standard RTAI scheduler module (namely *rtai_sched*) in order to provide the real-time scheduling subsystem. RTOS emulation modules can then be loaded on top of Xenomai's nanokernel, followed by a client application module using the emulated API.

RTAI port of Xenomai is based on the facilities provided by the core HAL module (namely *rtai*). The nanokernel-to-host software architecture interface is implemented using the real-time services exported by this module. For instance, let us look to the implementation of two critical functions, which respectively allow to enter and exit the RTAI context, thus preempting then reinstating the Linux kernel context.

From the file *xenomai/include/arch/rtai-386.h,*

```
#define INTERFACE_TO_LINUX
#include "asm/rtai_sched.h"
#include "rtai.h"

DEFINE_LINUX_CR0

static inline void xnarch_enter_realtime () {
   rt_switch_to_real_time(0);
   save_cr0_and_clts(linux_cr0);
}
static inline void xnarch_exit_realtime () {
   rt_switch_to_linux(0);
   restore_cr0(linux_cr0);
}
```

**4.2.2. Using the POSIX 1003.1-1996 layer as the host software architecture**

The aftermaths of the real-time constraints reevaluation - that we suggest to conduct when considering a port of a real-time application to a GNU/Linux system - may lead to envision a user-space execution, since soft real-time capabilities may be sufficient to support the requirements.

In such a case, implementing the nanokernel-to-host software architecture interface should be quite straightforward. For instance, the thread-related services can be mapped to the POSIX thread facility, and the periodic timer can be obtained from the POSIX virtual timer facility.

Combined to existing Linux kernel patches providing fixed-priority scheduling and fine-grain kernel preemptability, Xenomai's user-space execution may well deliver the expected soft to firm real-time performance needed while preserving the standard GNU/Linux programming model.

**4.2.3. Using Xenomai's Minute Virtual Machine**

The Minute Virtual Machine (MVM) is one of the real-time layers on top of which the Xenomai nanokernel and even RTAI's UP-scheduler module (rtai_sched) can run.

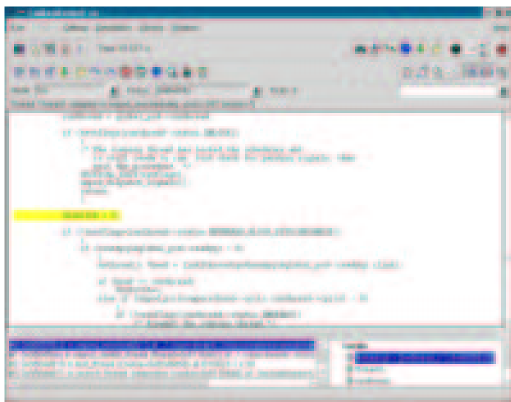
Like the other two real-time layers, the MVM provides the nanokernel (and the RTAI UP-scheduler) with the low-level resources it needs to schedule threads, handle interrupts, manage memory and so on, so it eventually thinks it runs on real hardware.

The MVM comes with a "RTOS-aware" graphical debugger named the Xenoscope that allows tracing the execution of real-time software at source code level in a simulated environment. This tool shows precisely how the multiple threads running in the system work together sharing the resources of a given real-time interface (e.g. who is locking a semaphore, which thread has been readied or suspended by a given system call, and so on).

Using the MVM to run a real-time application has a lot of

advantages: it does not require the cross-development tools, gives extended debugging, monitoring and tracing features and provides an easy way to stress the application under test with run-time situations otherwise barely conceivable on a real target. For instance, one can simulate bursts of interrupts generated at a very unreasonable rate...

Here is a snapshot of a Xenoscope debug session running the Xenomai nanokernel on top of the Minute Virtual Machine:



#### 4.3. Nanokernel description

Xenomai's nanokernel implements a set of generic services aimed at being a foundation for a set of RTOS API emulators running on top of a host software architecture. These services exhibit common traditional RTOS behaviors.

RTOS emulations are software modules which connects to the nanokernel through the pod abstraction. Only one pod can be active at a given time on top of the host software architecture. The pod is responsible for the critical housekeeping chores, and the real-time scheduling of threads.

#### 4.3.1. Multi-threading support

The nanokernel 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 wait on a synchronization resource.

Thread scheduler-related services are the following:

#### 4.3.2. Basic synchronization support

The nanokernel provides a synchronization object abstraction *(xnsynch)* aimed at implementing the common behavior 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.

| xnsynch_init | Initialize a synchronizatior |
|---|---|
| xnsynch_destroy | Flush and destroy a synchronization object |
| xnsynch_sleep_on | Make the running thread on the resource |
| xnsynch_set_ownership | Set a thread as the resou |
| xnsynch_wakeup_one_sleeper | Release the next thread f pending on the resource |
| xnsynch_wakeup_this_sleeper | Release a given thread frc pending on the resource |
| xnsynch_flush | Release all threads from pending on the resource |

#### 4.3.3. Interrupt management

A threaded interrupt model has been chosen in order to:

° Provide a mean to prioritize interrupt handling by software.
° Allow the interrupt code to synchronize with other system code using kernel mutexes, therefore reducing the need for hard interrupt masking in critical sections.

Xenomai's nanokernel exhibits a split interrupt handling scheme, in which interrupt handling is separated into two parts. The first part is known as the Interrupt Service Routine (ISR), the second being the Interrupt Service Task (IST).

When an interrupt occurs, the ISR is fired in order to deal with the hardware event as fast as possible, without any interaction with the nanokernel. If the interrupt service code needs to reenter the nanokernel (e.g. to resume a blocked thread), the ISR may require an associated interrupt service task to be scheduled immediately upon return. The IST has a lightweight thread context that allows it to invoke the nanokernel services safely. A Xenomai interrupt object may be associated an ISR and/or an IST to process each event.

This rather sophisticated scheme allows to easily emulate virtually all RTOS interrupt handling scheme on top of the nanokernel.

#### 4.3.4. Timer and clock management

Xenomai's nanokernel measures time as a count of periodic clock ticks. The periodic source is usually an external interrupt controlled by the underlying host architecture. Under RTAI/x86 for instance, the 8254 chip can be programmed to generate a periodic interrupt which can be hooked to a user-defined handler through the *rt_request_timer()* service. 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.

The nanokernel globally maintains three distinct time values, all expressed in clock ticks:
° The absolute number of elapsed ticks announced since the nanokernel is running
° The last date set by a call to *xnpod_set_date()*.
° The number of clock ticks announced since the last time the date was set.

| | |
|---|---|
| xnpod_tick_announce | Announce a new clock tick to scheduler |
| xnpod_set_date | Set the system date (in ticks) |
| xnpod_get_date | Get the system date (in ticks) |
| xntimer_init | Initialize a timer |
| xntimer_destroy | Stop and destroy a timer |
| xntimer_start | Start a timer |
| xntimer_stop | Stop a timer |

#### 4.3.5. Basic memory allocation

Xenomai's nanokernel provides dynamic memory allocation support with real-time guarantee, based on McKusick's & 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.

The memory chunks are obtained from the underlying software architecture. As far as RTAI is concerned, the memory pages composing the allocation heap are managed using the *kmalloc()/kfree()* Linux kernel routines. As soon as it is called on behalf of a real-time thread, the allocator transparently switches to the Linux kernel context using the RTAI-to-Linux service request feature when needed (i.e. *rt_pend_linux_srq()*). The proposed services are synchronous to the calling thread.

Memory-related services are the following:

| | |
|---|---|
| xnheap_init | Initialize a new memory heap |
| xnheap_destroy | Destroy a memory heap |
| xnheap_alloc | Allocate a variable-size block of memory |
| xnheap_free | Free a block of memory |

## 5. Features overview

The major focus of the Xenomai project is to help creating emulators of traditional RTOS APIs that ease the migration from these systems to a GNU/Linux-based real-time environment. As of now, the following real-time interfaces are available:

° pSOS+ emulator
° VRTXsa emulator
° VxWorks emulator
° uITRON implementation

Aside of the emulators, Xenomai ships with an experimental real-time interface called *Dualion*. *Dualion* is an API running on top of the **Xenomai** nanokernel, based on the direct message-passing paradigm. Such design is aimed at distributing more easily the application workload between kernel-based real-time threads and userland Linux processes by unifying these two domains messaging capabilities in a single and fast one.
*Dualion* uses a specialized driver called DBridge to establish communication channels between the real-time kernel space and the userland process space. *DBridge* stands for Domain Bridge, and can be used independently to have Xenomai threads talk to Linux processes, since it is directly based on both the nanokernel API and the Linux device driver interface.

**Xenomai** also provides a full-featured simulation engine as one of the supported real-time infrastructure. Running **Xenomai** on top of the Minute Virtual Machine (MVM) allows you to debug and stress most of the final real-time code using a RTOS-aware debugger in a comfortable user-space environment, including the application code, the real-time interface and the nanokernel.

An existing port of RTAI's original *rtai_sched* and *rt_mem_mgr* modules to **Xenomai's** Minute Virtual Machine, called 'Virtual RTAI', allows writing the hardware-independent part of a kernel-based **RTAI** application in userland, using a powerful RTOS-aware debugger to trace it. One should note that *VRTAI* only requires the *MVM*, and not the Xenomai nanokernel since the regular RTAI UP-scheduler is in charge of controlling the simulated real-time system.

## References


[1] The CarbonKernel project, at http://freesoftware.fsf.org/projects/carbonkernel/
[2] Jerry Epplin's paper "Linux as embedded operating system", at http://www.embedded.com/97/fe39710.htm
[3] Lui Sha, Raghunathan Rajkumar and John Lehoczky, "Priority Inheritance Protocols", at http://data.uta.edu/~ramesh/cse5326/papers/sha90.html
[4] M. Barabanov and V. Yodaiken paper "Real-Time Linux", at http://www.rtlinux.org/documents/papers/lj.pdf
[5] Montavista's white paper, "Linux for Real-Time: Strategies and Solutions" available from http:// www.mvista.com.
[6] Kevin Dankwardt's article in Linuxdevices "Comparing real-time Linux alternatives", at http://www.linuxdevices.com/articles/AT4503827066.html
[7] The RTAI position paper, at http://www.aero.polimi.it/projects/rtai/position_paper.pdf
[8] The Legacy2Linux project, at http://www.sourceforge.net/projects/legacy2linux/
[9] The Linux Trace Toolkit project, at http://opersys.com/LTT/
[10] "Design of a General Purpose Memory Allocator for the 4.3BSD Unix Kernel" by Marshall K. McKusick and Michael J. Karels, USENIX 1988.
[11] "Redesigning the BSD Callout and Timer Facilities" by Adam M. Costello and George Varghese.