The XENOMAI project Implementing a RTOS emulation framework on GNU/Linux

Company name

Philippe Gerum

First Edition

January 2004

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 application 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 by the free software foundation.
[http://freesoftware.fsf.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 behavioral similarities between the traditional RTOS are obvious.

The Xenomai project 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.

The Xenomai project relies on the common features and behaviors 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 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://freesoftware.fsf.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

Note

Ceci est une petite note sur 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.

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 ne 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 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 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 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 asynchronous 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 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 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 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 taht the highest the complexity, the greatest 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 is hooked to the hosting GNU/Linux through an hardware abstraction layer (HAL) which redirects external events to it, thus ensuring low interrupt latencies. RTAI provides a fixed-priority driver 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 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 serices 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 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 behavioral differences that may exist between the traditional RTOS and RTAI.

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.

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 currenlty 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 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 interprocess 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 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.

1.3. A common emulation framework

1.3.1. 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:

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 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:

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

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 this 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.

Disjunctive wait. A thread should be able to wait in a disjunctive manner on multiple resources. The nanokernel should unblock the thread when at least one among the pended 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 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.

1.3.1.3. Interrupt management

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

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. Interrupt prioritization should be left to the underlying architecture.

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.

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.

1.3.2. 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.

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:

- 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).

1.3.4. 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.

1.4. The Xenomai approach

1.4.1. Xenomai architecture

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

1.4.2. 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:

xnarch_init	Mount the real-time subsystem
xnarch_exit	Unmount the real-time subsystem
xnarch_hook_irq	Attach a handler to an interrupt line
xnarch_release_irq	Detach a handler from an interrupt
xnarch_enable_irq	Enable dispatching of an interrupt
xnarch_disable_irq	Disable dispatching of an interrupt
xnarch_chain_irq	Pass the interrupt request to the handler
xnarch_start_timer	Stop the periodic timer
xnarch_stop_timer	Stop the periodic timer
xnarch_enter_realtime	Switch current context to real-time
xnarch_exit_realtime	Switch current context to non real-time
xnarch_init_stack	Initialize a new thread's stack
xnarch_save_fpu	Save FPU registers for an outgoing thread
xnarch_restore_fpu	Restore FPU registers for an incoming thread
xnarch_init_fpu	Initialize a thread's FPU context
xnarch_set_imask	Set the global interrupt mask
xnarch_sysalloc	Allocate non-pageable memory
xnarch_sysfree	Free memory obtained from xnarch_sysalloc()

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.

1.4.2.1. 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);
}
```

1.4.2.2. Using the POSIX 1003.1-1996 layer as the 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 pro-

gramming model.

1.4.2.3. Using CarbonKernel to support Xenomai's Virtual Architecture

CarbonKernel is a RTOS simulator based on event-driven simulation techniques. The main idea is to provide a consistent framework for building simulation models which mimic the behavior of real-time operating systems on a workstation. The simulated RTOS kernels are built on top of a generic virtual RTOS. CarbonKernel allows tracing the execution of embedded software at source code level, running in a single regular GNU/Linux process, with concurrent target debugging capabilities, dynamic control of kernel resources, UI simulation, and much more. In other words, Xenomai can be thought as CarbonKernel's companion project, the former providing support for emulating RTOS APIs on the real target, whereas the latter simulates the RTOS APIs on a GNU/Linux workstation.

Since CarbonKernel is a powerful tool for understanding the dynamics of a real-time system, it is very well suited for helping in the nanokernel and RTOS emulators development. Moreover, it alleviates the burden of debugging a complex software in a rather 'hostile' kernel environment, especially in the first stages of its development. The idea is to implement a trivial CarbonKernel simulation model standing for a virtual architecture on top of which Xenomai can run. This way, Xenomai's internals could be debugged like mere application code running in a simulated environment.

A RTOS simulation model (called a 'personality') stacked on top of CarbonKernel is able to run, debug and stress an embedded application using the RTOS' native API. Simulating the RTOS behavior in a host-based environment instead of emulating the machine code of the target platform has a lot of advantages: it's faster, 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 easily generate bursts of simulated interrupts at a very unreasonable rate while still being able to observe and analyze the resulting situation.

Xenomai's Virtual Architecture (VA) is implemented as a CarbonKernel model exporting a limited set of services which provides a trivial thread management facility (i.e. basically: creation, deletion, suspension, resuming). The interrupt management is directly handled by the CarbonKernel built-in simulation facilities, whilst raw memory is obtained from the standard ANSI-C malloc routines.

From the file *xenomai/include/arch/va.h,*

```
#include "va/va.h" /* Include the VA model interface file */

typedef struct xnarchtcb { /* Per-thread arch-dependent block
ckhandle_t vahandle; /* The underlying VA thread handle */
void *cookie; /* XENO thread cookie passed on entry */
int imask; /* Initial interrupt mask */
 unsigned stacksize; /* Aligned size of stack (bytes) */
```

```
 int *sp; /* Saved stack pointer - unused */
 int *stackbase; /* Stack space - unused */
 } xnarchtcb_t ;

static inline void xnarch_switch_to (xnarchtcb_t *outtcb, x
        va_thread_resume(intcb->vahandle);
        va_thread_suspend(outtcb->vahandle);
}
```

Here is a snapshot of a CarbonKernel debug session running the Xenomai's nanokernel on top of the Virtual Architecture:

1.4.3. Nanokernel description

Xenomai's Virtual Architecture (VA) is implemented as a CarbonKernel model exporting a limited set of services which provides a trivial thread management facility (i.e. basically: 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.

1.4.3.1. 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 enter a disjunctive wait on multiple resources.

xnpod_thread	Create a new thread (left suspended)
xnpod_delete_thread	Delete a thread
xnpod_start_thread	Start a newly created thread
xnpod_suspend_thread	Make a thread enter a suspended state
xnpod_resume_thread	Resume a thread from a suspended state
xnpod_unblock_thread	Unblock a thread waiting for a resource
xnpod_renice_thread	Change a thread's base priority
xnpod_boost_thread	Boost a thread's current priority (inheritance)
xnpod_lock_sched	Disable the rescheduling procedure
xnpod_unlock_sched	Re-enable the rescheduling procedure
xnpod_activate_rr	Enable the round-robin scheduling
xnpod_deactivate_rr	Disable the undergoing round-scheduling
xnpod_reschedule	Start the rescheduling procedure

1.4.3.2. 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 synchronization object
xnsynch_destroy	Flush and destroy a synchronization object
xnsynch_block_thread	Make a thread pending on the resource

xnsynch_unblock_thread	Release a thread from pending on the resource
xnsynch_flush	Release all threads from pending on the resource
xnsynch_enter_pip	Make a thread own the resource
xnsynch_apply_pip	Prevent priority inversion (inheritance)
xnsynch_exit_pip	Release the ownership on the resource

1.4.3.3. Interrupt management

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 Deferred Service Routine (DSR). When an interrupt occurs, the ISR may be run with interrupts disabled, thus it should run as quickly as possible. To reduce this unpreemptable delay, lengthy processing can be delegated by the ISR to an associated DSR which will run later, outside the interrupt context, as soon as all pending interrupts are processed. This scheme allows for the DSRs to be run with interrupts enabled, thus allowing other potentially higher priority interrupts to be taken and processed.

This rather sophisticated scheme allows to easily emulate virtually all RTOS interrupt handling scheme on top of the nanokernel. Xenomai's interrupt-related services are the following:

xnpod_enter_interrupt	Signal an ISR entry to the scheduler
xnpod_exit_interrupt	xnpod_exit_interrupt

1.4.3.4. 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 the 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
xntimer_do_tick	Signal an incoming clock tick to the timer manager

1.4.3.5. Basic memory allocation

Xenomai's nanokernel 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.

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

1.5. Emulating pSOS+ on top of Xenomai

As a practical example explaining how traditional RTOS emulators can be built on top of Xenomai, let us have a brief look to details of the undergoing implementation of a pSOS+ emulator [http://freesoftware.fsf.org/projects/xenomai/emulators/psos4xeno/].

Eight pSOS+ kernel call families are targeted:

- Task management,
- Task signaling and synchronous service routines,
- Timer management,
- Counting semaphores,
- Event flags synchronization,
- Message queues,
- Memory partition (fixed-size memory block allocator),
- Memory region (variable-size memory block allocator).

The following code fragment shows the implementation of the t_create() service allowing to create a new pSOS+ task. One can observe that a few nanokernel calls are sufficient to obtain the basic support for the new task.

2. Autre section

Mon petit paragraphe.