Real-Time Application Interface

Pasi Sarolahti
Research seminar on Real-Time Linux and Java
University of Helsinki
Department of Computer Science
26th February 2001

Abstract

Real-Time Application Interface (RTAI) is a real-time Linux implementation based on RTLinux. It adds a small real-time kernel below the standard Linux kernel and treats the Linux kernel as a low priority real-time task. RTAI provides a large selection of inter-process communication mechanisms and other real-time services. Additionally, RTAI provides a LXRT module for easy development of real-time applications in user space. LXRT makes it possible to dynamically switch between real-time and non-real-time operation in the user-space. Finally, we give a small example of using RTAI and LXRT.

1 Introduction

Linux operating system is gaining popularity in research and student communities, and lately also in the business world. Linux is a multitasking system, which provides fair, non-preemptive scheduling among several dynamically created and deleted processes. The disadvantage of Linux is that it can not guarantee response times for its processes. However, there are application areas which require real-time response, such as robotic devices, computers used in health care and military, and various embedded systems used in different kinds of devices.

There are several commercial hard real-time operating systems available, such as QNX [Hil], AMX [Kad], RTKernel [On], which are small and convenient to be used in embedded systems, for example. However, some developers and researchers have pointed out that while there are needs for hard real-time, it would be useful to have the various device drivers, networking protocols, and other features offered by Linux also available.

For the above-presented needs, several Real-Time Linux implementations have been presented (KURT [NDH99], Linux/RK [OR99], etc.). Perhaps the best known solution is RTLlinux [BY97] which has been used as a basis for many other real-time Linux solutions. RTLlinux inserts a small real-time kernel below the standard Linux kernel and treats the Linux kernel as one of the real-time
processes. The disadvantage of this approach is that the real-time tasks operate in kernel-space and in case of software bug they may cause substantial harm.

One of the solutions based on RTLinux is *Real-Time Application Interface (RTAI)* developed in the Milan Polytechnic, which we describe in this presentation. Of the available Real-Time Linux solutions it claims to be one of the most actively updated and maintained. Currently RTAI patches exist for Linux 2.2 and Linux 2.4 and new features for RTAI are being constantly developed. The most important features of RTAI include various flexible inter-process communication methods and a symmetrical API allowing creation of real-time tasks in the user-space, avoiding the disadvantage of operating in the kernel-space.

Rest of this presentation is organized as follows. Section 2 describes the architecture and the fundamental concepts of RTAI. Some of the important features of RTAI are briefly described in Section 3. Section 4 describes *LXRT*, which is a common API for both real-time tasks and user-space processes. LXRT provides the means for implementing real-time tasks in user-space, and is hence one of the most important features of RTAI. Before we conclude, a simple example of how to construct a simple real-time application with LXRT and RTAI is given in Section 5.

## 2 RTAI Fundamentals

RTAI [Clo00, Lin00] has similar architecture to *RTLinux* [BY97]. Like RTLinux, RTAI treats the conventional Linux kernel as a low-priority real-time task, which may do its normal operations whenever there are no higher priority real-time tasks running. In the basic RTAI operation the real-time tasks are implemented as Linux kernel modules, similarly to RTLinux. RTAI handles the interrupts from peripherals and dispatches the interrupts to Linux kernel after handling the possible real-time actions triggered by the interrupts.

Figure 1 shows the basic architecture of RTAI, which is rather similar to RTLinux architecture. There are interrupts originating from processor and peripherals, of which processor originated interrupts (mainly error signals such as division error) are still handled by the standard Linux kernel but the interrupts from the peripherals (e.g. timer) are handled by RTAI’s *Interrupt dispatcher*. The RTAI forwards the interrupts to the standard Linux kernel handlers (*Hardware management* in Figure 1) when there no active real-time tasks. The interrupt disabling and enabling instructions (*cli()*/sti()* ) in Linux kernel are replaced by macros that forward the instructions to RTAI. When interrupts are disabled in the standard kernel, RTAI queues the interrupts to be delivered after the Linux kernel has enabled the interrupts again.

Additionally Figure 1 shows inter-process communication mechanisms, which are implemented separately for Linux and for RTAI. The communication mechanisms of RTAI are described in more detail in Section 3. Also the scheduler exists separately for Linux side and for
RTAI side. The RTAI scheduler is briefly described later in this section.

2.1 Hardware abstraction layer

RTAI developers introduce the concept of Real-Time Hardware Abstraction Layer (RTHAL) which is used for intercepting the hardware interrupts and processing them. RTHAL is a structure installed in the Linux kernel which gathers the pointers to the internal hardware related kernel data and functions needed by the RTAI to operate. The purpose of RTHAL is to minimize the number of changes needed to make to the kernel code and thereby improve the maintainability of RTAI and Linux kernel code. With RTHAL the different operations (e.g. the interrupt handlers) are easy to be changed or modified without having to interfere with the Linux implementation. For example, the RTHAL struct contains the interrupt handler table, which lists the functions that are called for handling different interrupts.

When RTHAL is installed on Linux, the function calls and data structures related to hardware interaction are replaced by pointers to the RTHAL structure. Initially the structure contains the pointers to the original functions and data of the Linux implementation, but when RTAI is enabled, the needed replacements are made to the pointers in the RTHAL table.

2.2 Scheduling

The scheduling units of RTAI are called tasks. There is always at least one task, namely the Linux kernel which is run as a low-priority task. When real time tasks are added, the scheduler gives them priority over the Linux kernel. The scheduler provides services such as suspend, resume, yield, make periodic, wait until, which are used in various real-time operating systems.

The scheduler is implemented as a dedicated kernel module (again similarly to RTLinux), which makes it easy to implement alternative schedulers if necessary. There are three different
3 RTAI FEATURES

RTAI features depend on the machine type. Uniprocessor (UP) scheduler is intended to be used on uniprocessor platforms, and it cannot be used with multiprocessor machines. Symmetric Multiprocessor (SMP) scheduler is designed for SMP machines and it provides an interface for the applications to select the processor or the pool of processors on which a given task is run. If the user does not specify any processor for the task, SMP selects the processor based on the processor load status. Multi-uniprocessor (MUP) scheduler can be used with both multi and uniprocessor machines. Unlike with the SMP scheduler, the tasks must be bound to specific processor when MUP scheduler is used. On positive side, MUP scheduler allows more flexible timer mechanisms for the tasks than SMP or UP scheduler.

3 RTAI Features

In order to make the application development as flexible as possible, RTAI developers have introduced a wealth of different mechanisms for inter-process communication (IPC) between real-time tasks and the user-space processes. In addition to IPC mechanisms, RTAI provides services for memory management and Posix compatible threads.

3.1 Inter-process communication

RTAI provides a variety of mechanisms for inter-process communication. Although the Unix systems provide similar IPC mechanisms to the user-space processes, RTAI needs to provide its own implementation for them in order to make it possible for the real-time tasks to use the IPC mechanisms, as they can not use the standard Linux system calls. The different IPC mechanisms are included as kernel modules, which can be loaded in addition to the basic RTAI and scheduler modules only when they are needed by the tasks. An additional advantage of using modules is that the IPC services can be easily customized and expanded.

The oldest, basic communication mechanism of RTAI are FIFOs. FIFO is an asynchronous and unblocking one-way communication channel between a Linux process and a real-time task having a size limit given by the user. No data can be overwritten in a FIFO, but a FIFO can become full in which case new data cannot be written until the old data is consumed. The boundaries of data segments written to FIFO are not preserved, and it is the reader’s responsibility to resolve them if needed. A FIFO appears to the user space as an entry in the /dev directory at the filesystem, in which there is a preallocated pool of 64 FIFO entries (of form /dev/rtfXX).

The FIFO implementation of RTAI is based on the RTLinux implementation, but RTAI provides some features which were not possible in RTLinux. Firstly, RTAI can trigger a signal when there are events on the FIFO (such as writing new data). An user-space process can then create a handler for the signal by the standard Unix mechanisms. However, this mechanism is not nec-
essary needed, as an user process may write to and read from FIFO by using the standard I/O calls such as `select()`). Additionally, there can be multiple readers and writers attached to a FIFO, which was not possible in the RTLinux version on which RTAI is based. Finally, FIFO identifiers can be dynamically allocated based on a symbolic name. Before this enhancement the FIFOs had to be opened using a global identifier from 0 to 63, which might cause problems if there were multiple independent processes and tasks using the FIFOs. By allocating the FIFO identifier dynamically after assigning a literal name for the FIFO this problem can be avoided.

**Semaphores** are the basic inter-process synchronization tools often used in operating systems. Semaphores are counters allocated and released by the tasks and processes. If there are no available resources held in a semaphore, processes are queued to wait for the resources to be released. RTAI provides an API for using semaphores, although each semaphore is technically associated to a FIFO, thus each semaphore uses one entry from the global FIFO pool. In addition to the basic semaphore services, a semaphore can be associated with a timer, which can be used to wake up a process in the semaphore queue even if the semaphore is still reserved.

**Shared memory** provides an alternative IPC paradigm to FIFOs, when a different communication model is required. Shared memory is a common block of memory which can be read or written by any processes and tasks in the system. As the different processes can operate on the shared memory asynchronously, a design issue is to ensure that data in the shared memory is not unintentionally overwritten. As RTAI may pre-empt and interrupt a writing or reading process, for example semaphores are needed to guarantee the mutual exclusion of a memory block.

Shared memory is implemented as a device driver, hence it requires an entry in the `/dev` directory before it can be used. When allocating a block of shared memory, the caller must give an identifier for the shared memory block and the size of the requested block. The first call for the given identifier allocates the memory block, whereas the next calls for the same identifier just return the address of the memory block allocated earlier. The identifier is an integer, which leads to similar allocation problems that were present with FIFOs, if there is no centralized control of the identifier allocation. However, also the shared memory identifiers be allocated dynamically using a symbolic name, in which case the conflicting allocations can be avoided. In addition to the basic shared memory implementation made by RTAI developers there are two enhanced third-party implementations for shared memory provided in the RTAI package.

Finally, perhaps the most flexible IPC method of RTAI are the **mailboxes**. Any number of processes can send and receive messages to and from a mailbox. A mailbox stores messages up to its size limit, and unlike FIFOs, mailbox preserves the message boundaries. There can be an arbitrary number of mailboxes active simultaneously. The mailbox sending and receiving operations can be associated with a timer, which means that if a sending or receiving operation do not get completed in given time, the control can be returned to the caller. Additionally, the caller may specify that if the mailbox can not hold the whole message, only part of it is consumed, or the caller may demand that whole message must fit to mailbox, or else the call will fail.
In addition to the mechanisms described above, RTAI has a facility for a simpler inter-process messaging, which is used as a basis for the Remote Procedure Call (RPC) facility of RTAI. We skip the description of these mechanisms for now.

### 3.2 Memory management

In the early versions of RTAI (and RTLinux) the memory had to be allocated statically, and real-time allocation was not possible. However, the present versions of RTAI include a memory management module, which allows dynamic allocation of memory in the real-time tasks by using an interface familiar from the standard C library.

RTAI preallocates chunks of memory (with configurable size and number) before real-time execution. When a real-time task calls `rt_malloc()`, the requested memory is given from a preallocated chunk. When the amount of free memory in the chunk is less than a threshold value, a new chunk of memory is reserved for future allocations. Similarly, when `rt_free()` is called, the released memory is left on the allocated chunk to wait for future reservations. When the amount of free memory is more than a high water mark value, the memory chunk is released.

### 3.3 Posix threads

RTAI provides `pthreads` module which implements threads according to the POSIX 1003.1c standard. By using the operations specified in the standard, the user may manage the threads with similar properties to the conventional Posix threads. Additionally, RTAI implementation of `pthreads` does not currently have the concepts of joining and detaching.

Threads under the same process share the memory space, thus it is easy for them to exchange information with each other. However, use of the shared memory area needs to be synchronized. In addition to the RTAI mechanisms described earlier, the RTAI pthreads implementation has operations for mutexes to have mutually exclusive locks in the program code and condition variables to help in synchronization.

### 4 LXRT: User-space Interface to RTAI

LXRT is an API for RTAI which makes it possible to develop real-time applications entirely in user-space, without having to create kernel modules. This is useful, because the traditional way of using kernel modules for real-time tasks has some drawbacks. Firstly, the kernel-space memory is not protected from invalid access of a buggy program. Modifying an unwanted location in memory can cause data corruption and cause malfunction of Linux kernel. Secondly, if the
underlying Linux kernel is updated, the kernel modules need to be also recompiled, which makes it impossible to move binary real-time tasks between different kernel versions.

While developing a real-time application in user-space (as a soft real-time process), the programmer may use the standard debugging tools until the application is considered to be bug-free. Moreover, the services provided by the Linux system calls are available to the task. After the task has been debugged, it can be converted into kernel space module as a hard real-time task. However, the Linux system calls are not anymore available for the hard real-time task, therefore the task has to manage with the services provided by RTAI.

The transition of application from user-space process to a real-time task is easy, because LXRT provides a symmetrical API for inter-process communication and other RTAI services. This means that the same API can be used by both kernel-space tasks and user-space processes. The same LXRT API can also be used when two user-space processes or two real-time tasks communicate with each other. This implies that the various timers and messaging systems that LXRT provides could be used by an user-space application even if it would not actually have any real-time requirements.

LXRT allows applications to dynamically switch between soft real-time and hard real-time by using a single function call in the user-space\(^1\). When an application is in soft real-time mode, it uses the Linux scheduler, but requires the process to change its scheduling policy to FIFO scheduling (see `sched_setscheduler(2)` man page), which is meant to be used when time-critical applications need to be run in Linux. FIFO scheduling provides static priorities with preemptive scheduling and hence better control over the scheduler than the default Linux scheduling algorithm with dynamic priorities. Therefore, FIFO scheduling should improve the response time of a process from the default Linux scheduling. However, a FIFO-scheduled process can still miss deadlines for various reasons (e.g. because of running interrupt handlers or tasks scheduled by RTAI).

In order to make it possible for an application to switch to hard real-time, LXRT creates a real-time agent in the kernel-space for each user-space process with real-time requirements. When the process enters hard real-time mode, the real-time agent disables interrupts, takes the process out of Linux scheduler’s `running-queue` and adds the process to RTAI scheduler queue. Now the process can not be disturbed by interrupts or other Linux processes. In order to switch to hard real-time mode the process must also make sure that the memory it uses stays in RAM and hence disable paging of the memory by using the `mlockall(2)` call. The Linux system calls can not be used in the hard real-time mode, but this restriction can be helped by having an assisting soft real-time process, which uses the Linux services on behalf of the hard real-time process. The two processes communicate using the IPC methods provided by RTAI.

---

\(^1\)RTAI developers distinct the real-time processes to soft real-time tasks (e.g. the tasks running under Linux scheduling) and hard real-time tasks. Tasks scheduled by RTAI scheduler are claimed to be hard real-time. RTAI developers do not use the notion of firm real-time.
Although the RTAI developers claim that by using LXRT the hard real-time applications can be developed in user-space, the response times are not as good for the user-space tasks using LXRT than for the real-time tasks running as a kernel module. In particular, according to RTAI developers, context switch takes “typically”\(^2\) under 100 us when using LXRT, but when the real-time tasks are operating as a kernel module, context switch takes under 40 us [Lin00].

### 5 Example of Using RTAI-LXRT

We now provide a simple example of how to use RTAI and the LXRT API according to [BD00]. Our example does not define any concrete action for our real-time process, but is meant to show how hard real-time tasks can be created in the user space. The emphasis is on showing the functionality of the RTAI API.

Figure 2 illustrates the memory space of a Linux host, separating the kernel-space area and the user-space area. A real-time application is launched in the user-space, which eventually creates two additional threads, a hard real-time controller thread and a Linux server which runs in soft real-time mode and is able to use the Linux kernel services. The threads communicate using a dedicated common memory area.

In the beginning the main process allocates memory for common data used by the different threads using RTAI’s own memory management module (rt_malloc()). Then the main module calls rt_task_init() to create the real-time agent in the kernel-space, which is required in order to execute hard real-time tasks in user-space and to use RTAI’s services. Additionally, the caller gives the priority of the task which is used if the task enters hard real-time and to be scheduled by the RTAI scheduler. The main thread issues this command even it does not require hard real-time, because it wants to use the services provided by RTAI and it is not possible without the initialization call.

Then the main module creates the controller thread and the Linux server thread using the Posix threads implementation of RTAI. These two threads do the most of the work in this example. After creating the threads, the main thread can be suspended to wait for the controller and Linux server threads to finish. When the main thread gets a permission to terminate, it has to clean up the internal task data and delete the real-time agent by calling rt_task_delete(). Finally, the main thread releases the memory it has reserved.

The Linux server task sets its scheduling policy (by using sched_setscheduler()) to FIFO in the Linux scheduler, as recommended for time-critical Linux processes. By doing this it can have priority over the standard Linux processes. Alike with the main thread, the soft real-time Linux server issues rt_task_init(), because it needs to use the RTAI services. Finally,
before starting to wait requests from the controller thread, the Linux server thread calls `mlockall()` Linux system call to lock the code and data memory pages of the Linux server thread to RAM. This is another way, in addition to setting the scheduling policy, to avoid some of the unexpected delays.

The controller thread is the only task with hard real-time requirements. In the beginning it also sets the FIFO scheduling policy in Linux scheduler so that it is served timely also in soft real-time operation. Then the controller thread initializes the RTAI task info and creates the real-time agent with `rt_task_init()`, similarly to the main and Linux server threads. Before entering hard real-time mode, the controller thread locks the code and data pages in the RAM memory by calling `mlockall()`.

Controller thread enters hard real-time mode by calling `rt_make_hard_real_time()`. This causes the Linux thread to be suspended and activates the controller’s agent task in RTAI scheduler. When the thread wakes up the next time, it is in hard real-time mode and returns from the `rt_make_hard_real_time()` call to the program that issued the call in the first place. At this point the Linux interrupts are disabled (although RTAI still may intercept them) and none of the Linux processes can preempt the controller thread.
The first call in the hard real-time mode for the controller thread is \texttt{rt_task_make_periodic()}, which makes the task to be scheduled on periodic intervals. After setting the scheduling period, the hard real-time controller enters to the main loop. After each repetition of the loop the controller waits for the next period which was set with the \texttt{rt_task_make_periodic()} call.

When the main loop of the controller is finished, the controller thread issues \texttt{rt_make_soft_real_time()} which suspends it on the RTAI scheduler. The Linux interrupts are enabled again and the thread is activated in the Linux scheduler. When returning from the function, the thread is in soft real-time. Finally, the controller thread releases the real-time agent and the related data in RTAI and unlocks the locked memory so that it can be paged again.

6 Conclusion

\textit{Real-Time Application Interface (RTAI)} is one alternative to incorporate real-time activities and the services provided by Linux operating system. The fundamental principle of RTAI is to provide a small real-time kernel below the standard Linux operating system, which is similar to what used in \textit{RTLinux}. RTAI provides a wide variety of services for task management, inter-process communication, memory management, etc. In addition, the recent versions of RTAI introduce LXRT, which provides a similar API for both normal Linux processes and real-time tasks.

The real-time Linux applications are often required to be run in kernel memory space in order to bypass Linux kernel and its standard interrupt handlers. By doing this, the behavior of a real-time task becomes predictable, and the response times can be bounded. However, developing program code for kernel space is risky, because the memory accesses are not protected and a bug in a program may cause corruption of internal kernel data and a system crash. If application is run in user-space, there are no such problems, but real-time execution of the application is difficult to ensure. LXRT module included in RTAI claims to be able to do this.

The reason for creating RTAI from the foundations of RTLinux was that the developers wanted more versatile features from the real-time Linux. Developing applications is easier if the underlying operating system provides alternative mechanisms for communication or task handling, so that the application developer may choose the method which suits the needs of the application. On the other hand, some developers want to have a simple and deterministic real-time operating system to be able to predict its behavior more closely. The numerous features of RTAI might therefore be too much for a developer who seeks a simple solution to assure real-time for his application.
References


