The Real-Time Application Interface

Karim Yaghmour
karym@opersys.com

Abstract

The Real-Time Application Interface provides Hard real-time capabilities in a Linux environment. It provides FIFO, semaphore, mailbox, message and RPC communication primitives and includes a Posix compliant subsystem. Hooking on to Linux is provided via a Real-Time Hardware Abstraction Layer, hence diminishing kernel intrusion.

Built on top of RTAI, the LXRT module provides for seamless integration of normal Linux processes and RTAI-bound real-time tasks. It enables these two types of tasks to communicate symmetrically while enabling normal Linux processes to enter the real-time execution domain using a single system call and provides memory protection and trap handling for hard real- time tasks. RTAI is supported by various other projects including the Linux Trace Toolkit and RTNet.

1 Introduction

General purpose operating systems (GPOSes) are designed to offer most applications a "fair" share of the system resources. Different mechanisms are implemented to ensure that this sharing is done efficiently. Mainly, sharing is enforced using aging schemes and execution precedence of important system code. This design philosophy is very well suited to the workstation and server environments where it finds its roots. For systems where real-time deterministic response times are critical, this approach is ill adapted and, in essence, does not fit the requirements.

In designing such systems, real-time operating systems (RTOSes) may be useful and are in fact often used to guaranty such things as interrupt response times, context switching and priority inheritance. The problem with these OSes however is that they very often support a much more restricted set of hardware platforms and devices compared to their general purpose counterparts. Not to speak of the cost of quality RTOSes and their restrictive distribution licensing.

It is in this context that a hybrid form of operating systems was born where a GPOS is provided with the control of most hardware resources but is itself subject to the control of a deterministic hard-real-time OS which enforces strict scheduling policies within the context of the GPOS. One such hybrid, the DIAPM Real-Time Application Interface, originated from the work done by Paolo Montegazza in using MS-DOS as the basis of a real-time operating system for the purposes of his research in the '80s. At the time, the real-time OS ran as a TSR[1] program.

With the advent of mainstream 32-bit OSes it became desirable to have such capabilities provided on such systems. As the Linux kernel was becoming more and more popular and its sources were available, an attempt was made to port the previously available functionality to it. Yet, at the time, the Linux 2.0.xx kernel wasn't deemed mature enough to support the RTHAL (Real Time Hardware Abstraction Layer) necessary for the undertaking. But this was soon overcome as the RTLinux project effectively diverted control from Linux for its real-time needs. Hence, a DIAPM-RTL implementation was born which used the NMT patch as the basis of its functionality.

With the release of Linux 2.2.xx in early 1999 it became possible to implement the RTHAL concept since the hardware management interface was properly layered. This enabled minimal modifications to the kernel while maintaining the dynamic loading of the real-time executive as a common kernel module. By March of the same year the first release of RTAI was made. Since then, RTAI has seen many increments and now includes many widely used commu-

[1]Terminate and stay resident.

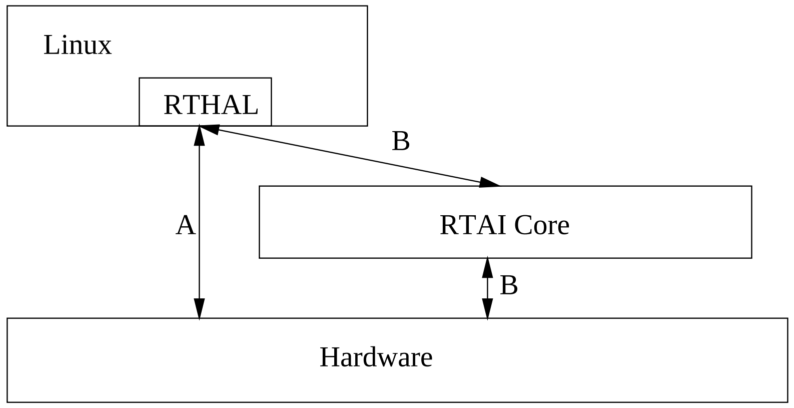

Figure 1: Control flow from and to the hardware interrupt facilities.

nication methods, is supported by a wide-variety of projects and runs on both the i386 and the PowerPC architectures. It is also widely used to implement real-time systems and is supported by a variety of vendors.

This paper will focus on presenting the different capabilities of RTAI, how they work, how they can be used and how they interact. Section 2 will discuss the intrinsics of RTAI's functionality. Section 3 will discuss the different schedulers provided with RTAI. Section 4 will present the different communication facilities and other services implemented by RTAI. Section 5 will discuss the LXRT symmetrical communication facility. Last, section 6 will look at the various extensions being made to RTAI and the various future directions of development.

2 Intrinsics

Real-time systems are designated as such because of their deterministic response times to outside events. Yet, we know that GPOSes are not capable of providing for such requirements. The Linux kernel, however efficient it may be, is no exception. It is, although, possible to provide a RTOS running alongside Linux to provide for real-time needs. To accomplish this hybrid configuration it is necessary to provide for Linux not to divert critical hardware events from the RTOS as these are the events that will dictate the response time of the system.

2.1 Taking control from Linux

As critical hardware events mostly take the form of outside interrupts it is the occurrence of these interrupts that will need to be diverted from Linux without hindering its normal behavior. In addition to diverting the interrupts from Linux it will also be important to ensure that Linux is not in a position to control the occurrence of these interrupts. Hence, the RTOS will have to be provided with means to divert these two control mechanisms: the flow from the interrupts to Linux and the control Linux has of the interrupts. In RTAI, this is done using the RTHAL which is the only addition made to the Linux kernel in the form of a 100 line (approximately) patch.

Figure 1 presents the RTHAL. In the case where RTAI is not loaded, case A, then Linux is directly interfacing with the hardware. When RTAI is loaded, case B, control of the interrupts goes through RTAI's core in either direction. To achieve this, the RTHAL is made up of function pointers which will point to Linux's native functions and tables initially and which will be diverted to RTAI's internal functions and tables when RTAI is loaded.

2.2 Managing the diverted resources

Once in control, RTAI will seamlessly provide interrupt control to and from Linux and, in addition, will provide a number of abstractions and facilities to real-time tasks. These include interrupt allocation and timer control. Also, RTAI provides a way

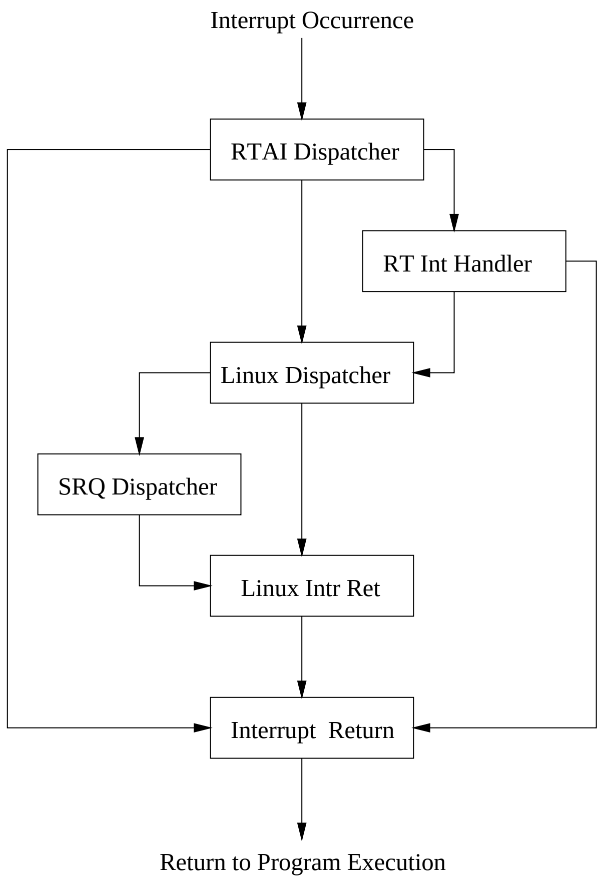

Figure 2: Path followed upon the occurrence of an interrupt in an RTAI/Linux system.

to call on the non-deterministic Linux facilities using System Requests (SRQs). This facility can also be used to dynamically extend the services provided to user-space programs.

To provide virtual interrupt control to Linux, RTAI implements replacements to the *cli()/sti()* couple. These replacements are actually RTAI functions which will set flags within internal maintained structures to keep record of whether Linux would like to be informed of incoming interrupts (sti) or whether he's ignoring them (cli). Since, in normal circumstances, all pending interrupts would be signaled upon the occurrence of a hardware *sti*, RTAI implements such a mechanism by registering all pending Linux interrupts and signaling them at the appropriate time.

Figure 2 illustrates the path an interrupt may follow from the moment it occurs to the moment where control is returned to program execution. When a global interrupt occurs (meaning through the 8259 PIC), the first function encountered is *dispatch_global_irq()*. If a real-time handler is registered for this interrupt, it will be called here. Once done, or if there are no handlers, a verification will be made as to whether Linux is expecting this interrupt or not and whether he has "disabled interrupts" (remember that such requests are caught by RTAI). If interrupts are enabled, the Linux interrupt dispatcher will be called upon. Otherwise, RTAI immediately exits the interrupt context and returns to normal program execution. When the Linux interrupt dispatcher is summoned, it may be followed by the system request dispatcher which will call upon all SRQs which may have been activated by an interrupt handler. It is using this facility that RTAI modules may have access to the standard non-deterministic kernel facilities. Once SRQs are called, if any, RTAI proceeds on jumping to the normal *ret_from_intr* code to finish the normal return sequence.

3 Schedulers

The RTAI schedulers use the facilities and abstractions provided by the core RTAI module described in the previous section to provide high-level programming abstractions to the programmer such as tasks and various communication mechanisms. There are three schedulers that come with RTAI: a uniprocessor (UP) scheduler, an SMP scheduler and a Multi-Uniprocessor (MUP) scheduler. All these schedulers can be used either in one-shot mode or in periodic mode. These modes relate to the way the timer is programmed and take their roots from the PC architecture where the 8254 timer may be programmed in various ways.

On that architecture, the timer may be programmed to emit interrupts at fixed time-intervals, in periodic mode, or may be reprogrammed at each interrupt, in one-shot mode. Periodic would seem ideal as it costs approximately 3 microseconds to reprogram the timer, but it may be ill-adapted to certain situations where intervals may vary. Hence the one-shot mode which has also the advantage of reprogramming intervals using the CPU frequency and not the timer frequency which, on the PC architecture, is considerably lower.

On the PowerPC architecture, the existing timing facility is embodied by the decrementer and provides a mechanism akin the one-shot mode of the 8254 found on the PC. In actuality, the processor's internal counter is used to generate an interrupt at a given time and needs to be reprogrammed upon firing. Periodic mode is therefore attained on this architecture by reloading a constant value.

The following subsections provide details about each scheduler available. Keep in mind that regardless of the differences, the facilities provided by RTAI remain the same and need not be used differently from one scheduler to the next.

3.1 Uniprocessor scheduler

The uniprocessor scheduler enacts a scheduling algorithm to select a task to be run on a single CPU. As such, its operation is very much straight forward: whichever process has the highest priority gets the CPU. In effect, it is a multi-list priority based scheduler with support for priority inheritance. In this scheme, Linux is a real-time task as any other but remains at the lowest priority level.

As for the implementation of the scheduler proper, it is split between two different yet complimentary functions: *rt_schedule()* and *rt_timer_handler()*. The former is invoked by the different facilities to enforce a scheduling change to reflect a modification in the state of a process. The later is exclusively targeted at dealing with the timer interrupt. At first it may seem that these functionalities should be grouped together to form a single scheduling function but given the slightly different approaches and uses, a choice was made to keep them separate.

3.2 SMP scheduler

The SMP scheduler differs from the UP scheduler in that it can schedule tasks to more than one CPU. This involves different degrees of additional services such as the capability to set CPU affinity for a task or to assign an IRQ to be dealt with on a specific CPU. Also, contrary to its UP counterpart on the PC architecture, the SMP scheduler is not limited to the usage of the 8254 as the single source of timer interrupts since the APIC possesses its own timer. The SMP scheduler remains, though, a priority-driven scheduler.

3.3 Multi-Uniprocessor scheduler

As the name suggests, the multi-uniprocessor scheduler views a multiprocessor machine (SMP) as being a collection of many uniprocessors. This means that each CPU can have its timer programmed differently. Hence, one could have a CPU running in periodic mode while another running in one-shot mode. This is very useful for some types of applications but involves setting CPU affinity at task creation, although it may be migrated later using the proper facilities.

4 Communication facilities and other services

One of the main features of RTAI is the wide array of communication facilities and other services made available to the programmer throughout the different schedulers while providing identical interfaces. Some of these services are part of the scheduler modules as they are simple enough in implementation while providing basic communication facilities. Others, more complex or less common, are implemented within their own separate modules. The following subsections discuss each of the services provided by RTAI.

4.1 Mailboxes

Mailboxes provide for a way to exchange data between multiple tasks using a pointer to a mailbox structure as the reference point. Typically, this will be used to send n bytes from a given buffer to a specific mailbox. On the receiving end, the reader can read m bytes from the mailbox into its own buffer. The following is a sample of the services provided by the mailbox facility:

- *rt_mbx_init()* Initializes a mailbox with a given size.
- *rt_mbx_delete()* Deletes the resources used by a mailbox.
- *rt_mbx_send()* Sends a sized message to a given mailbox.

- *rt_mbx_receive()* Receives a sized message from a mailbox.

Timed and conditional versions of the send/receive primitives also exist. These can be used to provide the programmer with greater control over the way his requests are dealt with by the system.

4.2 Messages and RPCs

Unlike mailboxes, messages and RPCs are task-based. One sends a message to or receives a message from a task. There is, therefore, no need to instantiate or initialize any structures or identifiers proper to messages or RPCs other than ensuring that the recipient or source of messages is actually a live task. This is somewhat explicit as all the calls to this facility require the passing of a pointer to the designated task. Hence, if we are waiting for a message from task X, we need to pass the message API the pointer to task X's structure. It is still possible to receive a message from any task by passing a NULL pointer to the message API when asking for such a reception. Also, the messages being transferred are not variable sized message, but are fixed size unsigned integer values. The following is a sample of the services provided by the message and RPC interface:

- *rt_send()* Sends a message to the given task.
- *rt_receive()* Receives a message from a given task.
- *rt_rpc()* Sends a message to a given task and expects a reply.
- *rt_isrpc()* Determines whether the given task is waiting for a response to an RPC.
- *rt_return()* Replies to an RPC from a given task.

Some of these services have timed and conditional variants. It is the case of the send, receive and RPC calls which may need to be used differently by the programmer.

4.3 Semaphores

Semaphores are a basic synchronization mechanism that enables multiple tasks to coordinate their work in a coherent way. In RTAI, semaphores are identified using their structures. Hence, tasks wanting to synchronize their work using a given semaphore will need a pointer to that semaphore's structure. In all other respects, RTAI semaphores behave the same way as conventional semaphores do. Here is a sample of the semaphore API:

- *rt_sem_init()* Initializes a semaphore to a given value.
- *rt_sem_delete()* Deletes a given semaphore.
- *rt_sem_signal()* Signals a semaphore.
- *rt_sem_wait()* Waits on a semaphore.

The wait call has timed and conditional variants which may be useful in some situations.

4.4 FIFOs

Contrary to the three previous facilities, the FIFO facility is a separate module that is optionnaly loaded if useful. As its name implies, this facility provides tasks with a way to put data within a buffer which will then be read on a first-in-first-out basis. The main usage of the FIFO module is the sharing of data from/to user space to/from real-time tasks. When used from within a kernel module, the FIFO API identifies a FIFO using its ID. In user space, this ID corresponds to an entry in the */dev* directory. Fifo 1 is visible as */dev/rtf1* from user space, for example. Using this, a real-time task can collect data in real-time while making this data available to a normal Linux process that is not bound by any real-time constraint. Data acquisition is one of the uses of this facility. From user-space, a task can communicate through this facility by using the conventional open, read, write, close and other services of the Unix file API. To the user application, the FIFO is just another file in the system.

The following is a sample of the API available to modules:

- *rtf_create()* Creates fifo with a given size and ID.
- *rtf_destroy()* Destroys a fifo.
- *rtf_reset()* Empties the content of a fifo.

- *rtf_put()* Puts data in a fifo.
- *rtf_get()* Gets data from the fifo.
- *rtf_create_handler()* Associates a handler to deal with the addition of data to the fifo in an asynchronous way.

In addition to these services, semaphore primitives have been added to provide for the synchronization of the access to the fifos. Also, it is now possible to name the fifos being created in order to increase the flexibility of the facility. Using this, identification of a correspondent depends on knowing the name of the fifo he uses.

4.5 Shared memory

Another way of sharing data between execution domains is through shared memory. To this end, RTAI provides a shared memory facility. Basically, this module provides for the allocation and freeing of memory regions. Identification of these memory regions is done using a name scheme which will ensure that further allocation of the same name will only result in the mapping of the designated region to the process's memory map while providing the caller with a pointer to said region. As with the FIFO facility this is an optional module that, once loaded, is usable by both user-space applications and real-time tasks. Communication of user space requests to the shared memory module is provided by the use of the SRQ mechanism described above.

4.6 Posix

As with other fields of computer science, the real-time field possesses its share of standards. One such standard is the Posix standard for real-time. Actually, there are multiple Posix standards for real-time. The RTAI Posix module implements the 1003.1c pthreads standard and a part of the 1003.1b standard, the message queues. It is not the intention of this paper to discuss the APIs provided by the Posix standard, but the interested reader is invited to take a look at the extensive documentation available on RTAI's web-site.

4.7 Memory management

A memory management facility may be seen as the wrong type of service to provide in a deterministic hard-real-time system, but it has some very practical uses. Least of which, support for higher-level languages like C++ which require the existence of the *new* and *delete* operators. The algorithm used by the memory management unit has been designed to provide for real-time memory allocation. This works by initially reserving a chunk of memory from the kernel using the conventional means. Thereafter, chunks of memory are provided upon request to the callers of *rt_malloc()* using a deterministic algorithm. Freeing of the request memory is done using the *rt_free()* call. The *new* and *delete* mechanisms are based on those basic memory management primitives.

4.8 Watchdog

In an effort to further insure that RTAI is a safe programming environment, a watchdog facility has been implemented. This facility can be used to insure that no one task will freeze the system because of its misbehaviors. Using the watchdog facility, it is possible to ensure that infinite loops and task scheduling overruns[2] do not handicap the system's ability to continue operating by enforcing a configurable reaction to such occurrences. It is therefore possible to suspend offending tasks or even kill them.

5 LXRT symmetrical interface

Of all the services and abstractions provided by the different RTAI modules, LXRT remains the most flexible and the most complex of them all. By providing the programmer with a symmetrical programming interface, LXRT integrates the best of both worlds in the hybrid GPOS/RTOS combination. In effect, it provides user applications with means to communicate transparently with real-time tasks and vice-versa. Figure 3 shows the possible communication interactions between tasks belonging to different execution domains. Note that tests

[2]When a task is rescheduled before it had the time to complete its intended job.

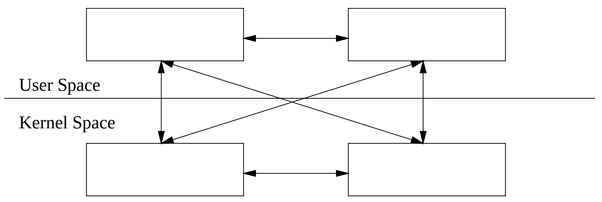

Figure 3: Possible LXRT symmetrical communication.

have shown that communication through LXRT is very fast.

5.1 User space services

Communication from user space to kernel space is possible through a software interrupt handled by LXRT. Using this software interrupt, user space tasks can call on exported RTAI services in very much the same way they call on exported Linux system calls. Among the services exported to user-space using LXRT, we find all the services previously only available to loadable modules such as mailboxes, messages, RPCs and semaphores. In actuality, the LXRT API makes it possible to render usage of these services to be completely transparent to the context. In other words, one can use the same functions and semantics in either user space or kernel space with the same effect. The only difference being the usage of a *main()* function instead of an *init_module()* and *cleanup_module()* interfaces. This makes it possible to effectively test real-time applications in user space prior to inserting them as kernel modules. However, note that user space applications using LXRT to access RTAI services are not hard-real-time tasks, they are only soft-real-time tasks. Although, as we will see in the next section, they can become hard-real-time tasks using LXRT. In any case, prior to using any other LXRT services, the user space applications need to instantiate a real-time shadow task which will be used to maintain coherent data structures within RTAI while dealing with intertask communication and scheduling.

With that said, LXRT is not limited to the predefined set of exported services and may be extended quite easily by providing an alternative function table which includes the initial table while adding to it the extra entries required. Such an extension is used by the RT_COM module to provide real-time com port communication to tasks through LXRT.

5.2 Stealing tasks from Linux

It was originally thought that with the hybrid GPOS/RTOS configuration tasks could either belong to one domain or the other and would be programmed differently depending on the domain they belonged to. The initial user space services provided by LXRT blurred this divide. The addition of a routine enabling normal GPOS tasks to become RTOS tasks takes this further by providing a means for normal Linux processes to become hard-real-time bound tasks through the use of the *rt_make_hard_real_time()* call. Contrary to loadable modules, such real-time tasks run in their own isolated memory space and, hence, provide for memory protection of real-time tasks.

Process stealing is done in two steps. The first part of the transition is done as part of the call made by the Linux process and consists of the following sequence:

1. Disable global interrupts.
2. Set Linux process state to TASK_LXRT_OWNED.
3. Raise the priority of the idle task (this is necessary for the second part of the transition).
4. Enqueue the function dealing with the second step as part of the normal Linux IMMEDIATE tasks queue.
5. Mark the IMMEDIATE bottom half to run.
6. Call the Linux scheduler.
7. Enable global interrupts.
8. Reset idle task to its original priority.

After this first step, the Linux process is in a state of limbo and will remain in this state until the second part of the stealing process is carried out. This second part will come to run within the standard bottom-half framework in Linux and consists of the following:

1. Disable global interrupts

2. Set the real-time task's state as READY
3. Run the LXRT scheduler
4. Enable global interrupts

The LXRT scheduler will take the necessary steps to insure that the task runs in a consistent memory configuration and will interact with the other RTAI modules to provide for scheduling of the task as if it were yet another RTAI task.

Just as it was possible to transition into hard-real-time space, it is possible to return to soft-real-time, as a normal Linux process, using the inverse of the above steps.

5.3 Exception handling

Given the memory protection possible with the process stealing method and the growing need to full (and fool) proof real-time programming, it becomes useful to implement exception handling to enforce protection policies and provide for other capabilities such as debugging. For this purpose, LXRT now handles processor exceptions. As RTAI is the first to receive processor exceptions, it provides for identifying the current execution context and passes exceptions onto the Linux exceptions handlers whenever necessary.

5.4 QNX-like services

As some RTOSes have been widely used and adopted for different uses, it is desirable to being able to use the same functionalities on open real-time kernels. Such is the case with the the synchronous IPC services provided in LXRT akin similar QNX services. This service enables tasks to communicate together synchronously using name schemes to locate recipients. In addition to synchronous communication, this facility also adds raw proxies functionality. Proxies are real-time tasks which can send a predefined message to a waiting task and hence trigger a certain behavior. Proxies may be used within interrupt handlers to signal a certain event to a waiting task; provided that the trigger is the last action taken by the handler.

The following is a sample of the services provided by this facility:

- *rt_Name_attach()* Attaches a name to the current task.
- *rt_Name_locate()* Locates a task identifying itself with the given name.
- *rt_Name_detach()* Detaches a name from a given task.
- *rt_Send()* Send a message to a task and wait for an answer.
- *rt_Receive()* Receives a message from a given task.
- *rt_Reply()* Reply to a received message.
- *rt_Proxy_attach()* Attaches a proxy to a given task.
- *rt_Proxy_detach()* Detaches the proxy of a given task.
- *rt_Trigger()* Triggers the action of a proxy.

5.5 Unix Server

As real-time tasks do not have access to standard Linux services many ways have been provided to circumvent this limitation, the unix server capability from LXRT is one of them. By starting a unix server prior to entering hard-real-time mode, a Linux processor can have access to some of the most commonly used Linux services. Starting a unix server is done through the *rt_start_unix_server()* call. In effect, starting a unix server forks the current process to execute an agent who will be in charge of the non-deterministic communication with Linux. Exchanges between the agent and the real-time tasks are done via a shared memory region to minimize overhead.

The following services are provided by the unix server:

- *rt_scanf()*
- *rt_printf()*
- *rt_open()*
- *rt_close()*
- *rt_write()*
- *rt_read()*

- *rt_select()*
- *rt_lseek()*
- *rt_sync()*
- *rt_ioctl()*

Note that calls to Linux services via the unix server remain non-deterministic and the caller will have to wait for Linux to complete servicing the request before continuing its operations.

5.6 Asynchronous I/O

The asynchronous I/O recent addition to LXRT provides programmers with an asynchronous I/O based on the glibc sources but adapted to provide the same functionality within the LXRT framework. This provides programmers with the following services:

- *aio_read()*
- *aio_write()*
- *aio_open()*
- *aio_close()*
- *aio_return()*
- *aio_cancel()*
- *aio_fsync()*

To deal with I/O requests, threads are started and handled by LXRT to carry out the requested service.

5.7 User library (liblxrt)

In an effort to provide developers with a way to develop their applications without having to run a modified kernel or RTAI, a user library LXRT has been provided. This library provides for standard interfacing with LXRT for development and enables development to be carried out in parallel.

6 Future directions

As RTAI is constantly evolving, there are different future directions which will be investigated while continuing to improve the currently available facilities. The following is a non-exhaustive list of things to come:

- More ports of RTAI (including ports of LXRT) to other architectures. As it stands now, prime targets are the MIPS and the ARM processors, but others are also being considered. As for LXRT, it is currently functional on i386 only and there is a desire to have it running on the PPC too.
- Extensive framework for C++ programming for RTAI. This is not limited to having C++ code run with RTAI, but having a real framework that would be usable both from a loadable module standpoint and from a user application standpoint.
- Real-time RAM filesystem.
- Flash-based filesystem.
- POSIX I/O layer to support filesystem.
- Integration of RTNet and socket layer.
- Integration of Linux Trace Toolkit hooks.
- Using RTAI services on RTLinux.
- Standalone RTAI.
- Port uClibc to RTAI kernel space.
- Fix uClibc to work with LXRT seamlessly.
- Better testing suite.
- Standard real-time development environment.
- Multiple interrupt priorities.
- Latency verification of code paths.
- More advanced memory management.

Many other enhancements are possible and the RTAI development team is open to any suggestions and contributions.

Acknowledgements

RTAI is the collective work of a team of developers which is built on a tradition of openness and cooperation. The rapid development of RTAI under Paolo Montegazza's lead and its mainstream adoption are a testament to this effort. Hence, a special thanks to all the RTAI developers (listed in no particular order, except for Paolo):

Paolo Montegazza
Stuart Hughes
Lorenzo Dozio
Trevor Wolven
Giuseppe Renoldi
Tomasz Motylewski
Pierre Cloutier
Steve Papacharalambous
David Schleef
Ian Soanes
Emanuele Bianchi
Brendan Knox
Erwin Rol
Karim Yaghmour

A great deal of thanks goes out to all the RTAI users and supporters who have made this effort all the while more enjoyable.

References

`http://www.aero.polimi.it/projects/rtai/`