DIAPM-RTAI POSITION PAPER, NOV 2000

Pierre Cloutier
Poseidon Controls Inc.
e-mail: pcloutier@poseidoncontrols.com

Paolo Mantegazza
Dipartimento di Ingegneria Aerospaziale, Politecnico di Milano
e-mail: mantegazza@aero.polimi.it

Steve Papacharalambous, Ian Soanes, Stuart Hughes
Lineo Industrial Solutions Group
e-mail: stevep@lineo.com
e-mail: ians@lineo.com
e-mail: stuarth@lineo.com

Karim Yaghmour
Opersys Inc.
e-mail: karim@opersys.com

Abstract

The development team has perfected and added much functionality to RTAI over the course of the last year. The position paper reviews this progress and focuses on the following areas:

- Improvements in the source tree, installation procedure and manual upgrade.
- Dynamic CPU frequency and bus frequency calibration.
- Port to Linux 2.4: 24.1.x releases and support for the PPC architecture.
- POSIX 1003.1c thread module with mutexes and condition variables.
- New Fifos: dynamic creation of named fifos, signal and semaphore interfaces.
- RT_MEM_MGR module: Dynamic memory management and C++ support.
- LXRT: soft and hard real time modes in user space with symmetrical API.
- Trap handling and memory protection while in plain RT and LXRT modes.
- LXRT-INFORMED: integration of RTAI, trap handlers and Linux at termination.
- RT_LXRT_COM and RT_LXRT_RNET modules: the concept of an extendable LXRT.
- MINI_LXRT: timers and tasklets running in user space.
- Integration of QNX IPC primitives, proxy messages and qBlk's to LXRT module.
- LIBLXRT: efforts to simplify the API for GUI and C++ programmers.
- LINUX_SERVER: access to Linux I/O while in LXRT hard real time mode.
- Linux Trace Tool Kit: support for RTAI including LXRT.

The LXRT module with its fully symmetrical API provides a safe and flexible tool to quickly implement hard real time programs in user space. Once the program is debugged, it can be easily migrated to the kernel for optimal performance if the application demands it. With CPU clocked near the 1 GHz mark, the necessity to execute code in the kernel becomes questionable. Thus, LXRT provides the trust direction of RTAI's future developments.

Introduction

RTAI results from the research done at DIAPM (Dipartimento di Ingegneria Aerospaziale, Politecnico di Milano) in the field of PC based real time control systems.

It all started in the late 80's when RTOS solutions like QNX, RTKernel and UCOS were considered. The decision to develop and in-house RTOS was then made, and DIAPM-RTOS was born. It ran under DOS and used the "terminate and stay resident" (TSR) technique.

The hardware abstraction layer concept was also extended to include all services required for real time applications and thus the term RTHAL (Real Time Hardware Abstraction Layer) that will be found in RTAI's documentation and source tree.

The RTOS needed to evolve to full 32 bits mode and again various options were considered including a port to GNU-DOS and Linux 2.0.xx.

The Linux 2.0.xx kernel was not mature enough to implement the full RTHAL concept, i.e. scattered interactions with the hardware, too many cli.sti's etc. However, the NMT RTL effort introduced a simple real time scheduler very close to DIAPM-RTOS's own real time scheduler. It demonstrated that DIAPM-RTOS could be ported easily to Linux almost unchanged.

The Linux 2.0.xx DIAPM-RTL implementation used the NMT patch and integrated to the real time scheduler services like semaphores, intertask messaging, QNX like synchronous messaging, and timing services. It enabled real time floating point support and also modified heavily the timer interrupt handler in order to have efficient periodic timing and TSC (CPU time stamp clock) based one-shot modes.

In early 1999 Linux 2.2xx made it possible to implement the RTHAL concept which was successfully done by mid-February. By the middle of April, RTAI-0.x was born with support for both UP and SMP under an SMP compiled kernel.

In September 1999, LXRT was introduced to allow faster development of real time applications by debugging the application in user space before porting it to the kernel.

Year 2000 saw the organization of a formal development team of about a dozen developers working together with a cvs tree. Followed an avalanche of improvements, correction of bugs, additional functionality and new modules that this paper will describe and summarize in the following pages.

Real Time Hardware Abstraction Layer

Essentially RTAI's kernel patch installs the RTHAL structure in the Linux kernel. The RTHAL performs three primary functions:

- gathers all the pointers to the required internal data and functions into a single structure, rthal, to allow the easy trapping of all the kernel functionalities that are important for real time applications, so that they can be dynamically switched by RTAI when hard real time is needed;
- makes available the substitutes of the above grabbed functions and sets rthal pointers to point to them;
- substitutes the original function calls with calls to the rthal pointers in all the kernel functions using them.

Linux is almost uneffected by RTHAL, except for a slight (and negigible) loss of performance due to calling cli and sti, and flags related functions, in place of their corresponding original Linux function calls and macros.

About 100 lines of code is all of what is changed or added in the kernel. Clearly, the RTHAL concept will facilitate and simplify a lot the long term support of RTAI.

Structure rt_hal definition in system.h

```
struct rt_hal {
	void *ret_from_intr;
	void *__switch_to;
	struct desc_struct *idt_table;
	void (*disint)(void);
	void (*enint)(void);
	unsigned int (*getflags)(void);
	void (*setflags)(unsigned int flags);
	unsigned int (*getflags_and_cli)(void);
	void *irq_desc;
	int *irq_vector;
	unsigned long *irq_affinity;
	int (*assign_irq_to_cpu)(int, unsigned long);
	void (*ack_8259_irq)(unsigned int);
	int *idle_weight;
```

```
        void (*lxrt_global_cli)(void);
        void (*switch_mem)
       (struct task_struct *, struct task_struct *, int );
        struct task_struct **init_tasks;
};
```

Structure rthal initialisation in irq.c

```
struct rt_hal rthal = {
        &ret_from_intr,
        __switch_to,
        idt_table,
        linux_cli,
        linux_sti,
        linux_save_flags,
        linux_restore_flags,
        linux_save_flags_and_cli,
        irq_desc,
        irq_vector,
        irq_affinity,
        assign_irq_to_cpus,
        ack_8259_irq,
        &idle_weight,
        0,
        switch_mem,
        init_tasks,
};
```

Because Linux uses the pointers in the above structure, it is possible for RTAI to change the functions that Linux uses and that is what the RTAI module does.

The RTAI Source tree

In the last year, RTAI has had a complete makeover of its build system. Using the Linux kernel source tree as a guideline, many improvements have been made.

All Makefiles files now use the toplevel Rules.make file to deduce their build rules. This has led to a more consistent and brief mechanism for developers when writing Makefiles within RTAI.

The first build of RTAI uses the header file dependency utility from the Linux kernel to build up a comprehensive set of dependency files for RTAI. This ensures that rebuilds are conducted correctly in response to changes within the system, or anything the system depends on.

The internal numbering system was changed (e.g 22.2.5), this was done for a number of reasons.

- It is easy to recognize which kernel series the RTAI variant will run on (the first digits, e.g 22 == for the 2.2 kernels). While this is a good idea, we hope to decouple RTAI from being restricted to a particular kernel variant.

- As in Linux, the second number indicates a stable or development version. Even numbers are stable, odd are development.

- The make system auto-generates an include/version.h file. Using this numbering scheme allows simple comparison of version numbers which may be used for feature detection.

RTAI now uses kernel patches rather than file copies to upgrade the kernel. This has helped to keep the system small, and more familiar to most developers. As part of this change, at make time a check is made to see that the patch has been applied, and also that the kernel has been configured for the RTHAL. One very important feature for embedded developers is that ifdefs have been added to the patch so that if CONFIG_RTHAL is not selected, all RTAI code is effectively removed from the kernel build.

The build system now has an install target that puts the RTAI modules into the appropriate /lib/modules/<ver>/misc directory. This means that once installed, the RTAI module stack can be loaded/unloaded with modprobe, without having to reference a specific version. Note also that the convention of having .o extension for modules is now observed. Another feature common to Linux is that it is possible to install in an alternate directory base (e.g for embedded systems) by using the INSTALL_MOD_PATH assignment to the make install command.

In addition to the changes in the build structure, RTAI has added a number of utilities aimed at making things simpler for the user.

To try to provide for compatibility with RTLinux, RTAI includes a header file include/rt_compat.h. This uses a series of wrappers to make it possible to write most applications using the RTLV1/RTAI API and have them build and run on RTL V2 or RTAI. The main benefit is that the application code itself looks much cleaner and so is easier to maintain.

To make it simple for a newcomer to get the flavor of RTAI, there are Perl bindings to LXRT. Using these bindings, you can write a script using the RTAI API

and immediately see your results with no compilation or Makefile woes.

Finally, we have seen the start of regression tests in RTAI (see newfifos) the idea is to give a simple go/no go test so that it is easy for the user (and developer) to determine whether a feature is functioning as expected.

Dynamic CPU frequency and bus frequency calibration

This replaces the CPU_FREQ and APIC_FREQ #defines in rtai.h with values obtained dynamically when the RTAI modules are installed. It is no longer necessary to recompile RTAI for different computers of differing specifications. The same binaries will now automatically calibrate themselves to the computer when they are installed.

- good for binary distributions
- good for a host target development environment

The CPU frequency calibration by default uses Linux's value (which for Pentiums is obtained dynamically at boot time by calibrating the TSC against the 8254 timer). Alternatively the cpu_freq_calibration utility can be run for 20 seconds or so to obtain a more accurate value. This calibrated value can be made to override the default value by using an insmod command line parameter...

insmod rtai.o CpuFreq=<calibrated_value>

The APIC frequency (usually only relevant for SMP machines) by default is read from the APIC timer directly. Alternatively the apic_freq_calibration utility can be run for 20 seconds or so to obtain a more accurate value. This calibrated value can be made to override the default value by using an insmod command line parameter...

insmod rtai.o ApicFreq=<calibrated_value>

RTAI proc Interface

The RTAI proc interface provides status and debug information on the current operating conditions of the RTAI real time operating system through the standard Linux /proc interface. A series of files under the subdirectory /proc/rtai gives information on each of the major active subsystems in RTAI. These files are activated when the associated module is inserted into the kernel. A description of these files and their contents is given below.

/proc/rtai/rtai

This file gives information on the rtai.o module, for example rtai version, interrupt state, etc. A typical output from this file is shown below:

```
-----------------------------------------------------------------
RTAI Real Time Kernel, Version: 22.2.4

RTAI mount count:  1

Global irqs used by RTAI:

Cpu_Own irqs used by RTAI:

RTAI sysreqs in use:  1
```

/proc/rtai/scheduler

This file gives information on the currently loaded rtai scheduler, for example, priority of the currently running real time tasks, state information, etc. A typical output from this file is shown below:

```
-----------------------------------------------------------------
RTAI Uniprocessor Real Time Task Scheduler.

Calibrated CPU Frequency: 333347000 Hz
Calibrated 8254 inter. to scheduler latency: 9027 ns
Calibrated one shot setup time: 1974 ns

Priority  Period(ns)  FPU  Sig  State  Task
---------------------------------------------------
    5      5000000    No   No   0x5    1
    6      8000000    No   No   0x5    2
```

/proc/rtai/fifos

This file gives status information on the real time fifos, how many are in use, the buffer size, etc. A sample output is shown below:

```
-----------------------------------------------------------------
RTAI Real Time fifos status.

fifo No  Open Cnt  Buff Size  malloc type Name
-----------------------------------------------------------------
0      1       20000     kmalloc
```

/proc/rtai/memory_manager

Current status of the dynamic memory manager is given by this file, for example number of memory blocks, amount of memory available in each block, etc. A typical output is shown below:

```
----------------------------------------------------------------
RTAI Dynamic Memory Management Status.

Chunk Size  Address    1st free block  Block size
----------------------------------------------------------
0    65536 0xc50c0000   0xc50c1798       59484
1    65536 0xc5170000   0xc5170010       65508
```

RTAI port to the PPC Architecture

Paolo Mantegazza worked on porting RTAI to PPC because of the interest of an Italian company and Zentropix. After testing RTAI on PII class CPUs, the company found there was no need for them to use the PPC as PII were cheaper and performed well enough.

Zentropix remained interested and supplied a portable G3 Mac.

Paolo's interest was to put the RTHAL concept to a hard real life test and verify to what extent RTAI is entangled into the ix86 architecture.

It took some spare time to study the CPU, using a well structured standard Motorola manual (The Programming Environments for 32-Bit Microprocessors), and Linux native code hacking and copying. After that, the beta porting of RTAI kernel space part, including FPU support, was done in three weeks working evenings and weekends only.

LXRT compiles but has not been tested. Hard real time mode in user space was never studied nor attempted.

DIAPM do not plan to use the PPC, and therefore Paolo's interest in going further is limited.

The Linux Implementation

The Linux PPC architecture is many miles behind its ix86 brother. We hope things are better on other archs, they say so for alphas. We feel PPC-Linux is much worse than advertised, if compared to the maturity, features and progress of the Linux ix86 architecture.

Relevant hardware summary:

PPC is a many, symmetrically usable, registers RISC CPU. It has no true stack, no push/pop instructions. The stack is emulated by using an index register. Gcc argument passing is mostly done using registers. Paolo did not check if asmlinkage declarations can change things. It defaults to big endian mode, even if it can work also in small endian mode.

Its timing source is an internal counter, called decrementer, guaranteed to be paced at the same frequency of the CPU time base (the equivalent of Intel TSC). The time base pacing is a small fraction of the CPU. The time base frequency does not come from slicing the CPU, but that does not matter much to the software. The decrementer cannot be programmed as periodic. It simply counts down then wraps around to full 32 bits and goes on counting down. At wrapping around it generates a specific interrupt. To have it periodic the decrementer must be reloaded so the PPC is natively a one-shot hardware timer to all practical effects.

External interrupts have just one vector, so it is up to the irq handler to dispatch and find the source by interacting with PICs. Software interrupt have also only one source, called trap, used also for some fault specific trap. There is a separated supervisor specific trap for OS system calls.

Software (Linux) summary

Because of the above, interrupts/traps dispatching is native in Linux Kernel and cli/sti equivalents are already in pointers to function. For an RTAIler that simply means that PPC Linux is natively based on the RTAI RTHAL concept. So there is not much to be patched. In fact RTL does no patching but Paolo did patch a few lines because he wanted to do a few thing differently.

Some technical notes on the port

It is only for UP.

Almost every thing of RTAI proved easily portable as it was. It could not be difficult because RTHAL is in Linux already and I was used to it. The main thing to be adapted where:

- the timer,
- knowing the external interrupt source,
- setting up an emulation of ix86 IDT table for soft irqs to save the RTAI srq concept and flexibility.

The first two had to work for sure to allow running kernel space applications. The third one is essential for shared memory, fifos and LXRT.

The RTAI timer is always a one-shot one. Periodic mode is simply supported by a fix reloading of the decrementer, after reading the time base, with the count required to insure the next interrupt corresponds to the fixed periodic tick. So all timed function always call for the decrementer update at each decrementer expiration. The difference being that an appropriate variable count is loaded in one-shot, while in periodic mode the count is changing just to account for the time elapsed to acknowledge the related interrupt.

After recalling the full 32 bits wrap around, it is noted that Linux does something silly. It reads the decremeter to get the counts from the interrupts, then keeps reading it to wait for another decrement and at that time calculates the new count to be loaded. It is sure that no decrementer variation will occur as the decrementer runs at a fraction of the CPU speed. However on slower CPUs, waiting for the decrementer causes the loss of some time.

The RTAI approach in periodic mode is simpler. At each interrupt the variable keeping the time is incremented by the period to calculate the base time of the next interrupt. The algorithm then reads the current time base, make the difference and loads the decrementer.

Finding the external interrupt source is done by simply using the pointer to the related function made available by native PPC RTHAL. Here the Linux source code was modified slightly as that pointer is passed to the Linux dispatcher for its soft interrupt. The dispatcher is in charge of finding the irq source when RTAI is not mounted, and the RTAI two lines of code patch saves it from doing it uselessly when RTAI is mounted.

Soft irqs for RTAI srq are emulated by causing a trap after loading registers with the interrupt number. The Linux trap handler has been patched so that it can understand if RTAI is mounted. In such a case it passes the trap to the RTAI handler that, by looking at the registers, understands if it is his or Linux, and then acts accordingly. Clearly in such a way one can also intercept Linux traps the way we have done in ix86. It should be OK as shared memory and fifos work well already. All the RTAI kernel space examples seem also to work well.

Status of the RTAI port

As explained above LXRT compiles but does not run. You have seen how I had to change the related call to lxrt_resume to get the right arg. However, there can be problems in the way the long long returned from the lxrt_handler are packed and recovered since they must match the endian mode.

As said above we stopped working on it because of the lack of interest and we were already amused and satisfied with what had been accomplished. The exercise did demonstrate the usefulness of the RTHAL concept.

The RTAI Module

It is a module that once installed lies in a dormant state ready to overtake Linux. The function init_module() does a few important things:

- initializes all of its control variables and structures;
- makes a copy of the idt_table and of the Linux irq handlers entry addresses;
- initializes the interrupts chips management specific functions.

Also, file rtai.h contains basic defines and inlined functions that perform some important RTAI services like timers services including support for 8254 timer and APIC timers.

Mounting RTAI

The execution of function **rt_mount_rtai**() (usually done by the schedulers or the fifos module) mounts RTAI services and fully traps the hardware.

A specific lock service is implemented (Linux spinlocks are no more protected by disabling the interrupt flags as Linux hold just soft flags, while RTAI needs true disables):

```
unsigned long flags; spinlock_t lock;
rt_spin_lock(&lock);
/*
   Critical code isolation in Linux,
   can be preempted by RTAI.
*/
rt_spin_unlock(&lock);

rt_spin_lock_irq(&lock);
/*
   Same as above but soft (flags) disabled only.
*/
rt_spin_unlock_irq(&lock);

flags=rt_spin_lock_irqsave(&lock);
```

```
/*
Critical code isolation in RTAI with interrupt disabled.
*/
rt_spin_lock_irqrestore(flags,&lock);
```

A global lock service to obtain atomicity across CPU's is implemented:

```
unsigned long flags;
rt_global_cli();
/* Critical code, interrupts disabled for all CPU's. */
rt_global_sti();

flags = rt_global_save_flags_and_cli();
/*
Critical code, Linux is already preempted. On SMP, a single global lock that locks out the other CPU's. Allows recursive calls within it from the same cpu. Works on UP boxes.
*/
rt_global_restore_flags(flags);
```

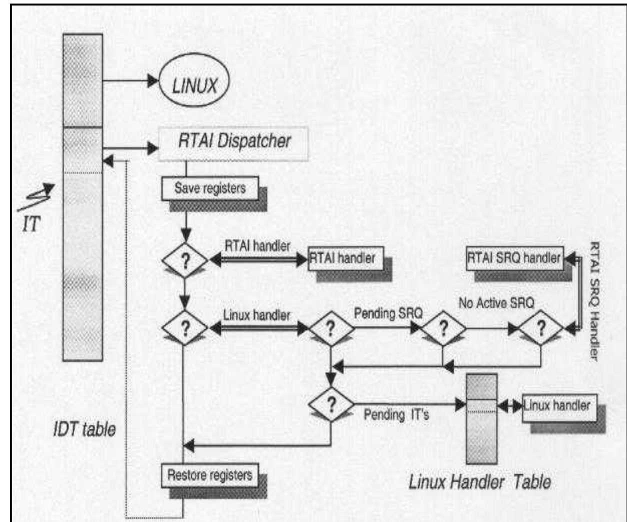
Also, RTAI needs a special form of hard lock disable across CPU's:

```
unsigned long flags;
flags=hard_lock_all();
/*
On UP boxes is the same as rt_global_save_flags_and_cli() above. On SMP locks out all the other CPU's.
*/
hard_unlock_all(flags);
```

The function rt_mount_rtai() actually intercepts all the hardware:

- Sets up the locks described above.
- sets up the global hard lock handler;
- hard locks all CPU's;
- redirects rthal interrupts enable/disable and flags save/restore to its internal functions doing it all in software;
- recovers from rthal a few functions to manipulate 8259 PIC and IO_APIC mask/ack/unmask functions;
- redirect all hardware handler structures to its trapped equivalent;
- changes the handlers functions in idt_table to its dispatchers;
- releases the global hard lock.

When rt_mount_rtai() returns, Linux appears working as nothing had happened but it is no longer the machine master. Function rt_umount_rtai() reverses the process decribed above and returns the system to its original state.



What happens when an interrupt comes in while RTAI is mounted?

While in RTAI real time mode a Linux interrupt is flaged as pending and its execution is delayed until RTAI switches to the Linux context again. Similarly, real time tasks may pend Linux service requests and those are flagged as pending as well.

When RTAI switches to the Linux context, function linux_sti() gets executed immediately after the switch as part of the global lock release algorithm. The function will dispatch for execution all the pending service requests and all the pending Linux interrupts before actually returning control to Linux.

While in the Linux context, a real time interrupt preempts Linux and gets executed immediately. If the interrupt is chained to a Linux interrupt, the Linux handler will also be dispatched immediately. If a service request was pended by the real time interrupt handler, the service request will be executed before control is returned to Linux.

What does the SMP hard_lock_all() do?

With an SMP compiled kernel the hard_lock_all() function protects itself using the global lock, then acquires a specific hardware spinlock and sends an Inter Processors Interrupt (IPI) message to all the other CPU's, using a vector dedicated to such a purpose. It then begins busy waiting on an agreed global volatile variable to be set by all other CPU's. The other CPU's are then interrupted by the message sent to them, set the agreed global variable and spin on the hardware specific lock with their interrupts disabled, thus blocking any activity on their CPU. When the agreed variable indicates that all the CPU's are locked the one in charge of the processing carries out its work and unlocks the global lock. At this point all the blocked CPU's acquire the specific lock in turn and return from interrupt. Notice that it is also possible to force the slave CPU's to execute a service function before returning.

The scheduler Modules

RTAI has a UniProcessor (UP) specific scheduler and two for MultiProcessors (MP). In the latter case you can chose between a symmetricMultiProcessor (SMP) and a MultiUniProcessor (MUP) scheduler.

The UP scheduler

The UP scheduler can be timed only by the 8254 timer and cannot be used with MPs.

The SMP scheduler

The SMP scheduler can be timed either by the 8254 or by a local APIC timer. In SMP/8254 tasks are defaulted to work on any CPU but you can assign them to any subset, or to a single CPU, by using the function:

- **rt_set_runnable_on_cpus**().

It is also possible to assign any real time interrupt service to a specific cpu by using functions:

- **rt_assign_irq_to_cpu**()
- **rt_reset_irq_to_sym_mode**()

Thus a user can statically optimize his/her application if he/she believes that it can be better done than using a symmetric load distribution. The possibility of forcing any interrupts to a specific CPU is clearly not related to the SMP scheduler alone and can even be used in interrupt handlers.
Note that only the real time interrupt handler is forced to a specific CPU. That means that if you check this feature by using "cat /proc/interrupts" for a real time interrupt that is chained to Linux, e.g. the timer when rtai_sched is installed, you can still see some interrupts distributed to all the CPUs, even if they are mostly on the assigned one. That is because Linux interrupts are kept symmetric by the RTAI dispatcher of Linux irqs.

For the SMP/APIC based scheduler if you want to statically optimize the load distribution by binding tasks to specific CPUs it can be useful to use the function **rt_get_timer_cpu**() just after having installed the timer, to know which CPU is using its local APIC timer to pace the scheduler. Note that for the one-shot case that will be the main timing CPU but not the only one. In fact which local APIC is used depends on the task being scheduling out, and that will determine the next shooting.

SMP schedulers allow to chose between a periodic and a one-shot timer, not to be used together. The periodic ticking is less flexible but, with the usual PC hardware much more efficient. So it is up to you to choose the mode in relation to the applications at hand.

measured on the basis of the CPU time stamp clock (TSC) and neither on the 8254 chip nor on the local APIC timer, which are used only to generate oneshot interrupts. The periodic mode is instead timed by either the 8254 or the local APIC timers. In the oneshot mode the time is hen the 8254 is used slow I/Os to the ISA bus are minimised as much as possible with a sizable gain in efficiency. The oneshot mode has just about 15-20% more overhead than the periodic one. The use of the local APIC timers leads to a further improvement and substantially less jitter.

Remember that local APICs are hard disabled on UPs, unless you are using just one CPU on an MP motherboard. Experience with local APIC timers shows that there is no performance improvement for a periodic scheduling, except for a marginal reduced jitter, while the oneshot case gain is the sizable 10-15% mentioned above. In fact by using the TSC just two outb() calls are required to reprogram the 8254, i.e. approximately 3 us, against almost nothing for the APIC timer. However you have to broadcast a message to all the CPU's in any case, and that is more than approximately 3 us. The APIC bus is an open drain 2 wires one and is not very fast. Note that the performance loss of the 8254 is just a fraction of the overall task switching procedure, which is always substantially heavier in the oneshot case than in periodic mode.

If you have an SMP motherboard, or a local APIC enabled, you should definitely use the APIC SMP scheduler. Note however that in this case we have chosen not to bound the timer to a specific CPU. Nonetheless, as explained above, you can still optimise the static binding of your task by using the function rt_get_timer_cpu() which allows you to find which local APIC is timing your application so that you can use the function rt_set_runnable_on_cpus() to bind any task to the "timing" CPU. See the README file in the smpscheduler directory.

Older 80486 machines

Since the TSC is not available on 486 machines, we use a form of emulation of the read time stamp clock (rdtsc) assembler instruction based on counter2 of the 8254. So you can use RTAI also on such machines. Be warned that the one-shot timer on 486 is a performance killer because of the need to read the TSC, i.e. the 8254 counter2 in this case, 2/3 times. That can take 6-8 us, i.e. more than it takes for a full switch among many tasks while using a periodic timer. Thus only a few kHz period is viable, at most, for real time tasks if you want to keep Linux alive. No similar problems exist for the periodic timer that need not use the TSC at all. So, compared to the 20% cited above, the real time performance ratio of the one-shot/periodic timer efficiency ratio can be very low on 486 machines. Moreover it will produce far worse jitters than those caused on Pentiums and upward machines. If you really need a one-shot timer buy at least a Pentium. However, for periodic timing 486s can still be more than adequate
for many applications.

The MUP scheduler

The MUP scheduler instead derives its name by the fact that real time tasks MUST be bound to a single CPU at the very task initialization. They can be afterward moved by using functions:

- **rt_set_runnable_on_cpus**()
- **rt_set_runnable_on_cpuid**()

The MUP scheduler can however use inter CPUs services related to semaphores, messages and mailboxes. The advantage of using the MUP scheduler comes mainly from the possibility of using mixed timers simultaneously, i.e. periodic and oneshot, where periodic timers can be based on different periods, and of the possibly of forcing critical task to remain in the CPU cache. With dual SMP machines we cannot say that there is a noticeable difference in efficiency. MUP has been developed primarily for our slowt (a few khz) PWM actuators, and BANG-BANG air jet thrusters, coupled to a periodic scheduler. All the functions of the UP and SMP schedulers are available in the MUP scheduler.

A new Fifo Module

The new fifo implementation for RTAI maintains full compatibility with the basic services provided by its original NMT-RTL counterpart while adding many more.

It is important to remark that even if the RTAI fifo API appears as before, the implementation behind them is based on the mailbox concepts, already available in RTAI and symmetrically usable from kernel modules and Linux processes. The only notable difference, apart from the file style API functions to be used in Linux processes, is that on the module side you always have only non blocking put/get, so that any different policy should be enforced by using appropriate user handler functions.

With regard to fifo handlers, it is now possible to install also one with a read/write argument (read 'r', write 'w'). In this way you have a handler that can know what it has been called for. It is useful when you open read-write fifos or to check against miscalls. For that you can have a handler prototyped as:

- int x_handler(unsigned int fifo, int rw);

that can be installed by using:

- **rtf_create_handler**
  (fifo_numver, X_FIFO_HANDLER(x_handler).

see rtai_fifos.h for the X_FIFO_HANDLER macro definition.

The handler code is likely to be a kind of:

```
int x_handler(unsigned int fifo, int rw);
{
        if (rw == 'r') {
                // Reading code.
        } else {
                // Writing code.
        }
}
```

Even if fifos are strictly no longer required in RTAI because of the availability of LXRT, they are kept

both for compatibility reasons and because they are very useful tools that can communicate with interrupt handlers since they do not require any scheduler to be installed. In this sense you can see this new implementation of fifos as a kind of universal form of device drivers since once your interrupt handler is installed you can use fifo services to do all the rest.

The new implementation made it possible to add some new services. One of these is the possibility of using asyncronous signals to notify data availability by catching a signal set by the user. It is implemented in a standard way, see the function:

rtf_set_async_sig(int fd, int signum) (default signum is SIGIO);

and standard Linux man for fcntl and signal/sigaction, while the others are specific to this implementation.

A complete picture of what is available can be obtained from a look at rtai_fifos.h prototypes.

Support for multiple readers and/or writers

It is important to remark that now fifos allow multiple readers/writers so the select/poll mechanism to synchronize with in/out data can lead to unexpected blocking. For example: you poll and discover that data is available and then another user preempts you and steals all your data with the result that when you finally ask for it the data is gone and you get blocked. So, make sure that you cannot be blocked when you read or write data. To avoid such problems you have available the functions:

- **rtf_read_all_at_once**(fd, buf, count);

that blocks untill all count bytes are available;

- **rtf_read_timed**(fd, buf, count, ms_delay);
- **rtf_write_timed**(fd, buf, count, ms_delay);

that block just for the specified delay in milliseconds but are queued in real time Linux process priority order. If ms_delay is zero they return immediately with all the data they could get, even if you did not set O_NONBLOCK at fifo opening.

So by mixing normal read/writes with their friends above you can easily implement blocking, non blocking and timed IOs. They are not standard and therefore not portable, but far easier to use than the select/poll mechanism. The standard llseek() is also available but it is equivalent to calling rtf_reset(), whatever fifo place you point at in the call.

For an easier timing you have available also:

- **rtf_suspend_timed**(fd, ms_delay).

To make them easier to use, fifos can now be created by the user at open time. If a fifo that does not exits already is opend it is created with a 1K buffer. Any subsequent creation in the kernel side resizes it without any loss of data. Again if you want to create a fifo from the user side with a desired buffer size you can use:

- **rtf_open_sized**(const char *dev, perm, size).

Since they had to be there already to implement the mailboxes we have also made available binary semaphores. They can be used for many things, e.g. to synchronize shared memory access without any scheduler installed instead of using blocking fifos read/writes with dummy data.

Semaphore services

The semaphore services available are:

- **rtf_sem_init**(fd, init_val);
- **rtf_sem_wait**(fd);
- **rtf_sem_trywait**(fd);
- **rtf_sem_timed_wait**(fd, ms_delay);
- **rtf_sem_post**(fd);
- **rtf_sem_destroy**(fd);

Note that fd is the file descriptor. A semaphore is always associated to a fifo and you must get a file descriptor by opening the corresponding fifo.

Naturally the above functions are symmetrically available in kernel space except for rtf_sem_trywait() and rtf_sem_post() which are only available in user space, because as explained above, you only get non blocking services in the kernel.

Dynamic creation of named fifos

To make it easier to keep track of which fifo to use and in order to avoid fifo number clashes beween separate real time tasks, it is now possible to

dynamically create named fifos on an unused fifo number. Existing named fifos can have their name looked up in order to find which fifo number they occupy. The named fifo services available are:

- **rtf_create_named**(name);
- **rtf_getfifobyname**(name);

Technical Notes

These functions are symmetrically available in kernel and user space. Both return the allocated fifo number. In user space note that these calls will not automatically open the fifo device for you. Instead one must append the returned fifo number onto the end of '/dev/rtf' and then open the fifo device as normal.

The maximum length of a fifo's name is defined as RTF_NAMELEN. This is currently set to 15.

When using rtf_create_named() from user space, the first fifo created is assigned a fifo number of 1 rather than 0. This is because /dev/rtf0 is used to communicate with the kernel driver module (where the name to number mapping is kept), and so at the time of calling fifo number 0 is not free. This should not cause any problems. The same thing does not happen when rtf_create_named() is called from kernel space.

If you want to monitor the fifo name to number mapping two choices are available. Either look in /proc/rtai/fifos or use the new RTF_GET_FIFO_INFO ioctl. Take a look in the test program regression.c and rtai_fifos.h to see a (slightly contrived) example of using this ioctl. Sample /proc interface output...

fifo No	Open Cnt	Buff Size	malloc type	Name
0	1	1000	kmalloc	kernel_FIFO_345
1	2	1000	kmalloc	user_FIFO_12345

Future implementations may employ SRQs rather than /dev/rtf0 for the name resolution.

The pthread Module

The RTAI pthread module implements a thread package to the POSIX 1003.1c standard. The module includes calls for thread creation and destruction, mutual exclusion, and condition variables. This gives the real time programmer the ability to program the application using a standard threads API.

pthreads

The module provides for the dynamic creation and destruction of threads, so the number of threads does not have to be known until runtime. POSIX threads use attribute objects to represent the properties of threads. Properties such as stack size and scheduling policy are set for a thread attribute object. A thread has an id of type pthread_t, a stack, an execution priority, and a starting address for execution. In POSIX, threads are created dynamically with the pthread_create function which creates a thread and puts it in a ready queue.

During its life a pthread can be in any one of four states; Ready, Running, Blocked, and Terminated.

A pthread is in the ready state when it is able to run, but is waiting for a processor to become available. Usually it is in the ready state on creation, when it has been blocked or pre-empted by another pthread or task.

A pthread is in the running state when it is executing on a processor. A pthread is blocked when it is unable to run because it is waiting for an event. Typical examples of this include waiting to lock a mutex, suspending execution for a time period, or waiting for an I/O operation to complete.

A pthread is terminated when it returns from its start function or by calling pthread_exit. Under real time Linux, this state is very short as the concepts of pthread joining and detaching are not currently implemented. Hence, when pthreads are terminated they are recycled immediately.

Synchronization

In the majority of cases, applications that are written using pthreads will have a requirement to share data between pthreads and ensure that certain actions are performed in a coherent sequence. This requires that the activity of the pthreads is synchronized when accessing the data in question to avoid incorrect operation and undesired effects. Under RTAI the synchronization functions that are available for applications are mutexes and conditional variables.

Mutexes

The most common method of synchronizing access to a resource that isshared between multiple pthreads is to use a mutual exclusion, abbreviated to mutex. A mutex is used to serve as a mutually exclusive lock

which permit pthreads to control access to sections of data and code requiring atomic access. In these circumstances only one pthread can hold the lock and hence access the resource that the mutex is protecting. Mutexes can also be used to ensure exclusive access to sections of code or routines; these are known as critical sections of code.

Condition Variables

One of the main differences between a mutex and a condition variable is that a mutex allows threads to synchronize by controlling access to data, whereas a condition variable allows threads to synchronize on the value of the data. A condition variable provides a method of communicating information regarding the state of shared data. For example this information could be a counter reaching a certain value, or a queue becoming empty.

The Memory Management Module

The dynamic memory module for RTAI gives real time application programs the ability to be able to dynamically create and free memory using the standard UNIX programming API calls. Before this module, real time applications which needed dynamic memory management had to use the standard Linux kernel calls: kmalloc() and kfree(). This is potentially very dangerous as these calls can block, and if this were to occur from a real time task, the result is usually a total system lock up. The situation is made worse as this can lead to intermittent bugs as real time applications can appear to work using these calls, but fail under varying load conditions and circumstances.

The dynamic memory manager module pre-allocates blocks (chunks) of memory from the Linux kernel which is available for use by real time tasks using a "UNIX like" API: rt_malloc and rt_free. The manager allocates and frees memory from these blocks of memory, and also monitors the amount of free memory that is available in these blocks. When the amount of available memory falls below a low water mark a request for another block of memory is made pending. Similarly, when the amount of available memory is greater than a high water mark, a request to free a block of memory is made pending. These pending requests are carried out using the standard kmalloc and kfree calls at a safe time, i.e. when the real time system becomes idle, just before control is handed back to Linux. Using this mechanism, the memory manager balances the memory requirements of the real time application with the need to keep as much memory as possible available to the Linux kernel.

The dynamic memory manager can be configured for the number of memory blocks that are kept available, and the size of the blocks. This means that the module can be configured to meet the specific requirements and operating conditions of a real time application, allowing the application to be programmed using the flexibility of dynamic memory allocation, whilst minimizing the memory resource burden on the Linux kernel. Another key feature provided by the module is the ability to create real time threads from other real time threads which is an essential feature for many applications. For this purpose, RT_TASK ***rt_alloc_dynamic_task**(void) has been added to the schedulers.

Dynamic memory allocation for real time tasks is supported by the implementation of the following functions:

- void ***rt_malloc**(unsigned int size);

rt_malloc() allocates size bytes and returns a pointer to the allocated memory. If the allocation request fails a NULL is returned.

- void **rt_free**(void *ptr);

rt_free() frees the memory space pointed by ptr which must have been returned by a previous call to rt_malloc(). rt_free returns no value.

The default configuration of the dynamic memory manager is:

Memory block size: 64 KBytes
Number of free blocks kept available: 2

These parameters can be changed if required by using the following module parameters:

Memory block size: granularity
Number of free blocks kept available: low_chk_ref

For example to change the size of the memory blocks to 32 Kbytes and the number of free blocks kept available for allocation to 4:

Insmod rtai_sched_up.o granularity=32768 low_chk_ref=4

RTAI C++ Support Built Into The Module

Real time C++ support is provided with the implementation of the operators:

- void* operator **new**(size_t);
- void* operator **new []**(size_t);
- void operator **delete**(void*);
- void operator **delete []**(void*);

These operators use the rt_malloc() and rt_free() primitives and thus make it possible to execute real time C++ written modules. The LXRT directory provides an example on how to do this. Notice that C++ support is limited as programs must be compiled with the -fno-exception g++ option. Also, an abstract base class that defines a pure virtual function should implement an empty function otherwise the compiler will generate the **__pure_virtual**() call which will result in an unresolved symbol at insmod time:

```
class Wathever {
public:
        virtual void foo() = 0;      // Pure virtual.
        virtual void Better { }      // Empty function.
};
```

Member function foo causes the problem whereas empty member function Better() tricks g++.

Alternatively you could choose to implement your own __pure_virtual() function with something like this:

```
extern "C" void __pure_virtual();
void __pure_virtual()
{
RT_TASK *t;
rt_printk( "%X calling a pure virtual\n", t = rt_whoami());
rt_task_suspend(t);
}
```

Notice that if you attempt to compile with the -fPIC option, you will see another unresolved symbol: _GLOBAL_OFFSET_TABLE_.

Also, you cannot instantiate global objects because nothing actually does the global initialization. This is normally done before the program enters main and you need to link with the library files crtbegin.o and crtend.o to do that. However, linking with those two files will introduce you to two other unresolved symbols:

- __register_frame_info
- __deregister_frame_info.

To resume, C++ support is limited in that exception handling and global instantiation services are not available in the kernel due to a lack of library support. It is possible to trick the compiler in order to avoid the __pure_virtual unresolved symbol.

There is now in the source tree an example almost entirely written in C++ to help users get started. The example is simple and yet sophisticated in that it illustrates many aspects of the language like derivation, composition, templates, virtual functions and, of course, provides a Makefile that compiles a C++ written kernel module.

LXRT - the symmetrical API concept

Paolo Mantegazza's objective when he started to think about LXRT was to implement in user space the message handling and remote procedural call functions that are integrated to the RTAI schedulers.

Programmers familiar with RTAI would then be able to do IPC in user space without having to master all the details and intricacies of System V IPC and libc6. This way staff and students could focus on their research in the field of aerospace and spend less time learning Unix.

The next step was simple. Given that RTAI messaging functions would be available in user space, it would be usefull if the functions internals allowed to cross the kernel/user space boundary. This way a user space program could send a message to a kernel task and vice versa using the same function call prototype.

The initial development of the LXRT module implemented all the RTAI scheduler services in user space with very few exceptions and changes to the API functions prototypes. Moreover, the problem of crossing the kernel/user space boundary was surmounted and the API functions could be used inter and intra space for both kernel and user space.

The internals of the first LXRT implementation

A number of problems were resolved in the first LXRT implementation of the symmetrical API. Let's follow the flow of control as if we were actually making a call from user space:

First, it is necessary to create a RTAI real time task agent. The agent will enter the real time space and actually execute the native API functions if and when required. The function rt_task_init() creates the agent. In a similar way, the function rt_task_delete() releases the resources required to instantiate the real time agent. The agent real time task structure, stack and messaging buffers are allocated dynamically.

Usually real time kernel tasks have statically declared task structures and it is therefore easy to share pointers to those structures. Any task can initiate a messaging procedure if the name of the variable that declares the global task structure of the receiver is known. Clearly user space programs would have to use a different approach. A name registry algorithm was developped and enables kernel and user space programs to register a unique name up to six characters long. Any task that knows a registered name can find the address of the real time task structure of the registered task and therefore can initiate a messaging procedure with it. The registry algorithm also supports mail boxes, semaphores and proxies.

A simple approach using static inline functions in header file rtai_lxrt.h was used to copy the function arguments onto the stack before executing the software interrupt. With the help of macros most of the API functions where quickly implemented in user space with two lines of code each. The register arguments of the system call encode (on the stack) the size of the argument structure as well as the function ID number and a pointer to the argument structure.

LXRT installs in init_module() an interrupt handler that saves the registers, calls lxrt_handler() after pushing the register arguments onto the stack, and restores the registers at the end of the system call. The handler save/restore are done the same way Linux does it thus keeping the possibility and flexibility to return with a ret_from_intr() although at first a simple iret instruction was sufficient.

Once in the Linux context, the function ID number is decoded and used as an index into a matrix of structures that contain the pointer to the native API function as well as information as to what to do next. About 20% of the native functions do not need to enter RTAI real time context. In that case the arguments are copied from the user space stack with copy_from_user() and the native function simply called.

When a context switch is required, LXRT calls lxrt_resume() to prepare the agent before the context switch can be accomplished. Function arguments may have to be copied from user space to message buffers using dedicated pointers in the real time task structure. The stack of the agent has to be initialised and the address of the native API function copied onto it among other things. The context switch will transfer control to the stub function lxrt_rtai_fun_call() that will in turn disable global interrupts, call the native API function and then automatically call rt_suspend(). Function rt_suspend() always calls rt_schedule() to force a context switch (the return to Linux) and may also pend a Linux service request to wakeup the the user program in the Linux context if need be. After having done this initialisation, emuser_trxl() is called to do the context switch to kernel real time mode.

The user program agent task wakes up in kernel real time mode executing the desired native API function. At this point two things can happen. The agent could exit the function immediately and start unwinding things to go back to user space, or, the function could decide to block and call rt_schedule() to switch to another real time task.

In the first case, the native API function exits and the stub function calls rt_schedule(). At some point in time the RTAI scheduler restarts Linux. The user program wakes up in kernel mode and continues to execute lxrt_resume(). Global interrupts are re-enabled and, if required, data is copied back to user space. The system call then completes and control returns to user space.

In the second case, the agent is blocked in the real time kernel when the RTAI scheduler restarts Linux. The user program wakes up in kernel mode, re-enables global interrupts and immediately suspends itself by setting state to TASK_INTERRUPTIBLE followed by a schedule() call. Linux then continues with another process.

At some point in time, the user program real time agent eventually exits the native API function. The stub function described above calls rt_suspend(). A service request to wakeup the kernel component of the user space program is pended before the rt_schedule() call to do a context switch. This forces the execution of the required wake_up_interruptible() call in the Linux context before the current process actually resumes. A Linux context switch occurs and our user program wakes up again in lxrt_resume() and completes the system call as described above for the first case.

The need to cleanup at process termination time

With the first release of LXRT it became apparent that a soft real time task that terminated abnormally could not be re-started because the registr() call in rt_task_init() failed. Nobody had deregistered the task name at termination. LXRT needed to be informed by Linux of the termination event in order to do required house cleaning. Deemed acceptable in the context of laboratory research more needed to be done to graduate to industrial status.

The objective behind the development of LXRT-INFORMED was to have a system that could recover after the crash of a linux task with a real time LXRT agent.

A few lines of code were added to function do_exit() to allow the detection of real time agents that need to be deleted by LXRT at process termination time. Function do_exit() calls a special handler installed (and de-installed at cleanup_module() time) by the LXRT module. In order to minimise the overhead of that additional call in do_exit() it was decided that do_exit() would only do the call for POSIX processes. Thus LXRT-INFORMED works only for processes that change their Linux scheduling policy to SCHED_FIFO or SCHED_RR.

When a POSIX process terminates do_exit() calls linux_process_termination() and the following actions are taken in sequence and as follows:

- Disable global interrupts.
- Try to find an agent for the current process.
- Verify in the registry if current registered semaphores. If so, and for each, call rt_sem_delete() and rt_free() to release the allocated memory, and also erase the registry entry.
- Verify in the registry if current registered mail boxes. If so, and for each, call rt_mbx_delete() and rt_free() to release the allocated memory, and also erase the registry entry.
- Verify in the registry if current attached proxy messages, if so, and for each, call rt_Proxy_detach(), and also erase the registry entry.
- If an agent task was not found above enable global interrupts and return, otherwise continue. Notice here that this approach works if the user forgot to release the resources during a normal exit.
- Go through the list of RTAI real time tasks and try to find those that may be SEND, RPC, RETURN, or DELAYED blocked on the agent task found above. For each task found, unblock it and force a context switch to RTAI.
- Call rt_task_delete(), and then rt_free() to release the messaging buffers as well as the task structure itself. Remember that the structure was allocated dynamically.
- Finally deregister the task name and enable global interrupts.

Notice that mail boxes cause a particular problem here because they are connection less. In other words, it is not possible for a zombie (a former agent task about to be deleted) to detect that another real time task is MBX (mail box) blocked specifically for a message from him. The solution here is to anticipate this possibility at the system design stage and to use the rt_mbx_receive_timed() function with a timeout value and check the return value to detect the error.

Extensive testing was carried out with test programs using the server (SRV) client (CLT) model and doing synchronous IPC transactions to validate the algorithm under various conditions:

- SRV does a divide by zero error in user space,
- throw SIGINT at SRV with kill -s INT pid,
- throw SIGINT at SRV from real time context,
- Hit Control C,
- SRV exits without deleting a SEM,
- SRV exits without deleting the agent real time task,
- SRV exits normally without deleting anything,
- SRV does a divide by zero in user space while a rt_task is RPC blocked on CLT,
- Control C while a real time task is RPC blocked on CLT,
- SRV does a divide by zero in user space while a real time task is RPC blocked on SRV

The return values from failed rt_rpc() calls were verified to the correct value of 0 when the real time task RPC blocked on either CLT or SRV was unblocked by linux_process_termination(). Furthermore, the tests were carried out under both the SMP and UP schedulers.

Performance and benchmarks

Intertask communications with LXRT are about 36% faster than with old FIFO's. Testing inter Linux (Linux <-> Linux) communications with int size msg and replies (using the native rt_rpc() function) on a P233MMX the following results were obtained:

LXRT	12,000 cycles RTAI-0.9x
Fifo	19,000 cycles RTAI-0.8
Fifo new	22,300 cycles RTAI-0.8
SRR	14,200 cycles

The results speak for themselves. Notice that the new fifo implementation provides much more flexibility than the original implementation but at a small price in performance. The SRR package implements the QNX 4 IPC Send/Receive/Reply primitives with a standard kernel module using ioctl(). LXRT, because it bypasses ioctl() altogether is more efficient and provides with the symmetrical API inter space IPC (kernel <-> user space).

LXRT Switching from soft to hard real time mode

A user space process calls rt_make_hard_real_time() to switch to hard real time mode. Once in that mode, the process can no longer make system calls or use a library function that could lead to a system call. However, the rich family of RTAI messaging services can be used. Thus, any system call can be relayed by the real time process to a soft real time agent. Anyone who has worked on the QNX platform is familiar with such an approach. The Unix server example shows exactly how it is done.

The rt_make_hard_real_time() call enters the kernel and first waits on semaphore steal_give_sem. The purpose of this semaphore is to exclude all other processes from trying to enter or leave hard real time mode at the same time. Those two procedures handle only one process at a time.

When rt_sem_wait() returns, steal_from_linux() is called. Notice that when the process executed rt_task_init() the pointer to the RT_TASK structure of the real time agent task was stored in the task structure of the Linux process in variable this_rt_task[0].

The function steal_from_linux() adds the real time agent to the list of processes that lxrt_schedule() is concerned with. It sets the state of the Linux process to TASK_LXRT_OWNED so Linux will not try to restart it while in hard real time mode. It increases the goodness priority of the idle process because we need to use it later and we want it scheduled as soon as possible. It queues a bottom half function to execute lxrt_do_steal(). Finally, it calls schedule(). The process is now in limbo.

Notice that in the initialization described conceptually above, flag exec_sigfun was set. The first transition from false to true changes a lot of things. In init_module() LXRT sets the signal function of the real time task representing the current process to execute lxrt_sigfun(). Recall that the signal function is always executed immediately after the context switch which means that lxrt_sigfun() gets executed whenever the real time scheduler re-starts Linux. When flag exec_sigfun is false, lxrt_sigfun() is an empty function. When exec_sigfun is true, lxrt_sigfun() calls lxrt_schedule() and then disables hard interrupts. This means that as soon as there is an agent task in the task list of lxrt_schedule(), lxrt_schedule() gets called whenever rt_schedule() re-starts Linux. We will come back to lxrt_schedule() later. Let's now worry about the process still in limbo.

At some point, the bottom half algorithm executes lxrt_do_steal() that will re-schedule itself in the bottom half until three conditions are true: the running CPU is not already in hard real time mode, the state of the process in limbo is equal to TASK_LXRT_OWNED and the current process is an idle task. When the conditions are met, lxrt_do_steal() disables global interrupts, sets the pstate of our process to READY and calls lxrt_schedule(). The next time the process wakes up, it will be running in hard real time mode. Notice that pstate is the state variable used by lxrt_schedule().

When the process finally wakes up global interrupts are enabled, the goodness priority of idle is reset to its normal value, the semaphore steal_give_sem is released and the rt_make_hard_real_time() call returns to user space with a simple iret instruction. Notice that ret_from_intr() calls are not allowed while in hard real time mode.

LXRT Scheduler

As we have seen above, the function lxrt_schedule() is a signal function that gets called every time Linux gets re-started by the real time scheduler when at least one process is in hard real time mode.

The LXRT scheduler behaves in a similar way as the real time scheduler except that it has its own task list and is concerned with task state variable pstate. When

the function executes, it disables global interrupts and then tries to find a task with pstate equal to the READY state.

If it finds none, it restarts Linux if it was not the previous task, and resets lxrt_hrt_flags (if it was previously set) thus enabling system calls return through ret_from_intr().

If it finds one, it sets the pstate variable of the RT_TASK representing Linux to READY (i.e. it stops it because it was also RUNNING) if Linux was the previous task, it re-starts the hard real time process and sets lxrt_hrt_flags (if it was not set) thus disabling system calls return through ret_from_intr().

Like the Linux scheduler, lxrt_schedule() executes rthal.switch_mem() immediately before the context switch in order to load the local descriptor table and the cr3 register etc. If need be, the coprocessor stack is saved and restored immediately before and after the context switch.

Notice the usefulness of the hardware abstraction layer concept: rthal.switch_mem is simply a pointer to the native function Linux uses to switch the memory to the next process.

On exit, the scheduler enables global interrupts.

LXRT Switching from hard to soft real time mode

Typically, a user space process calls rt_make_soft_real_time() to switch back to soft real time mode in order to exit normally with the exit() function (a system call).

The rt_make_soft_real_time() call enters the kernel and first waits on semaphore steal_give_sem. Again, the purpose of this semaphore is to exclude all other processes from trying to enter or leave hard real time mode at the same time.

When rt_sem_wait() returns give_back_to_linux() is called. This function removes the process from lxrt_schedule's task list, decrements nr_linux_rt_process if non zero, schedules the execution of lxrt_do_give_back() in the bottom half queue, and then calls lxrt_schedule() to force a context switch. Again, the process is in limbo.

Notice that when nr_linux_rt_process reaches zero things start to come back to normal. Each time lxrt_sigfun() is executed, the flag exec_sigfun is set equal to nr_linux_rt_process. Thus, the lxrt_schedule() call in give_back_to_linux() will be the very last one if nr_linux_rt_process reached zero when it was decremented.
At some point, lxrt_schedule() will restart Linux and the bottom algorithm will execute lxrt_do_give_back(). This function acts in a similar way as its counterpart lxrt_do_steal_from_linux(). It will re-schedule itself in the bottom half until three conditions are true: the running CPU is not already in hard real time mode, the state of the process in limbo is equal to TASK_LXRT_OWNED and the current process is an idle task. When the conditions are met, lxrt_do_give_back() will set the process state to TASK_INTERRUPTIBLE and simply call wake_up_interruptible(). The process will eventually be scheduled to run by Linux.

When the process finally wakes up global interrupts are enabled, the goodness priority of idle is reset to its normal value, the semaphore steal_give_sem is released and the rt_make_soft_real_time() call returns to user space with a ret_from_intr() call as for any other Linux system call.

Notice the trap handling special case of a hard real time task that must be terminated because of an exception. If the task does a division by zero in the kernel we cannot use any Linux function that references current because it is not defined. "current" is an inline function that will return garbage whenever in RTAI real time mode. Thus, the bottom half setup procedure is avoided and the process is waken up in the Linux context via a standard RTAI service request that eventually calls wake_up_interruptible().

We decided to steal from and give back to an idle process because it was easier to implement at first and allowed the validation of the LXRT concept. It does cause a short interruption of the program flow which is not a major problem as user's are not expected to seesaw between the two modes. Future development may look into the possibility of stealing from and giving back to any process.

Finally, the need for the steal_give_sem semaphore is clearly seen when comes the time to give two processes back to Linux. If they are given back at the same time, the wake_up_interruptible() calls are executed back to back with the result that the second one succeeds and the process that should have been started by the first one falls in limbo and stays there for ever. Without the semaphore, we could infringe the cardinal rule: for each processor, there is to be only one process in the kernel at a time.

LXRT QNX like Synchronous IPC Services

Raw synchronous messaging has always been there in the RTAI schedulers with the rt_rpc(), rt_receive() and rt_return() primitive functions.

Using the raw proxies functionality added last year and the existing messaging primitives the basic QNX like messaging primitives were implemented in LXRT to obtain a symmetrical API:

- pid_t **rt_Alias_attach**(*name);
- pid_t **rt_Name_attach**(*name);
- pid_t **rt_Name_locate**(*host, *name);
- int **rt_Name_detach**(pid);
- int **rt_Send**(pid, *smsg, *rmsg, ssize, rsize);
- pid_t **rt_Receive**(pid, *msg, maxsize, *msglen);
- pid_t **rt_Creceive**(pid, *msg, maxsize, *msglen, delay);
- int **rt_Reply**(pid, *msg, size);
- pid_t **rt_Proxy_attach**(pid, *msg, nbytes, priority);
- int **rt_Proxy_detach**(pid);
- pid_t **rt_Trigger**(pid);

Again, full API symmetry means that one can use the above functions to do synchronous messaging within the kernel, within user space, or between the kernel and user space. The memcpy() function is used and therefore the implementation is not as efficient as one designed to use shared memory (like the Unix server example). However, the memcpy() function will allow to extend the functionality of the QNX like primitives over the network.

Notice that the pid_t pid's returned by the functions above have nothing to do with the standard Linux pid's. Think of them more as handles as they are managed internally by the implementation. Also, pid's are encoded in the lower 16 bits only, and therefore can be differentiated from small negative error numbers.

Recall that natine RTAI names are 6 characters long (because they are encoded into 32 bits). The function rt_Name_attach() is meant to be used by kernel task that do not automatically have a native name. In user space one would use rt_Alias_attach() passing a pointer to a null as the argument in order to obtain the pid. The pointer can point at an optional 31 characters long string holding an alias name. Function rt_Name_locate() looks for equivalence with both the native and the alias name if any.

Raw proxies

Raw proxies are real time tasks ready to send a pre-canned messages (created by an owner task) to the owner task. In practice, the proxy is the task pointer of a real time proxy agent task sitting there doing nothing, always ready to send the pre-defined proxy message.

A real time task or an interrupt handler that knowns the proxy can use the function rt_trigger() to wakeup the proxy agent who in turn will send the proxy messages to the owner of the proxy. The number of messages that will be sent is equal to the number of times rt_trigger() will have been called. rt_trigger() does not block. It does not wait for a reply.

API Function Prototypes

pid_t **rt_Name_attach** (char *native_name);

Registers a native name for the calling task with LXRT and returns the pid of the task. Once this call has been made, the task can use the family of synchronous IPC functions.

Native names can be up to 6 characters long excluding the null at the end. Acceptable characters are numeric and upper case alphabetic. The additional characters '$' and '_' are also valid. This design constraint results from the fact that native names are encoded into a four bytes unsigned long.

The function is not available in user space.

If native_name points to a null string, the function will automatically create a name of the form "T_XXXX" where XXXX is the hex ASCII representation of the returned pid. As the pid is unpredictable, the rt_Name_locate() function is meant to be used for names agreed upon up front and registered with rt_Name_attach().

pid_t **rt_Alias_attach**(char *any_name);

The implementation allows the user to register an alias names that can be up to 32 characters long including the null at the end. The rt_Name_locate() function searches through the list of native names and also checks for the alias names if any.

User space program first use rt_task_init(nam2num(name), ...) to initialise the real time agent. In so doing, they supply a native name automatically. The program obtains the pid from the returned value of the function. If an alias name is not

required the argument should be a pointer to a null string. The usage of the function is mandatory in user space.

The function returns the pid_t if successful. Otherwise returns an error code:

- EBUSY - name already exists.
- EAGAIN - name space used up.
- ENOMEM - no memory to fulfill request.
- EINVAL - illegal null pointer.

pid_t **rt_Name_locate**
(conts char *host, const char *name);

Locates a process that has registered its name with rt_Name_attach() or rt_Alias_attach(). If host is null the search is made locally. If host is not null then a network search occurs. If the name is located on another computer, the initial VC (virtual circuit) buffer size will be equal to a default size of 512 bytes. VC buffers grow dynamically. Notice that network communications are not yet implemented. The function returns a process id if successful, otherwise it returns zero.

int **rt_Name_detach**(pid_t pid);

Removes the registered name and deregisters the process from LXRT. The pid parameter must be the same as the one returned by rt_Name_attach(). When a process dies, its name is be detached from the system and all real time resources created by LXRT will be freed. The function returns zero if successful, otherwise return an error code.

pid_t **rt_Proxy_attach**
(pid_t pid, char *data, int nbytes, int priority);

Creates a canned message of length nbytes pointed to by data. The proxy will be attached to process pid. If pid is zero, the proxy will be attached to the calling process. The proxy can be assigned a priority. A value of -1 defaults to the priority of the calling process. The proxy acts as a messenger always ready to send its message. A proxy can send a zero byte message by setting nbytes to zero. The function returns a process id on success. On error, the function returns a negative error code:

- EAGAIN - no free process entries.
- ENOMEM - not enough memory.
- ESRCH - pid does not exist.

int **rt_Proxy_detach**(pid_t proxy);

Releases the proxy previously created by the calling process. Returns zero on success. Otherwise, the function returns a negative error code:

- EPERM - you are not owner of the proxy.
- ESRCH - proxy does not exist.

pid_t **rt_Trigger**(pid_t pid);

Trigger the proxy agent to send a message to the process which owns the proxy. The calling process will not block. If more than one trigger occurs before the proxy message is received, that number of messages will be received. The function can be called from an interrupt handler provided it is the last call the handler does. The owner of the proxy can trigger the proxy to himself. The function returns the process id of the task who owns the proxy. On error, it returns a negative error code:

ESRCH - pid does not exist.
EINVAL - pid is not a proxy.

pid_t **rt_Receive**
(pid_t pid, void *msg, size_t maxsize, size_t *msglen);

Waits for a message from process pid. If pid is zero, waits for a message or proxy from any process. If a message is waiting, up to maxsize bytes are copied into msg. If a message is not waiting, the process will enter the RECEIVE blocked state. Messages are queued in priority order. RTAI allows to change this to FIFO time order by removing the MSG_PRIORD define in the scheduler source code. If you specify to receive from a task in particular and that task dies while you are RECEIVED blocked on it then the function returns -ESRCH. The number of bytes transferred will be the minimum of that specified by both sender and receiver and will be copied into msglen. The maximum number of bytes that can be transferred is unlimited as the messaging buffers grow dynamically. Receive changes the state of the sender from RPC to RETURN blocked. The function returns the pid of the sender on success, otherwise it returns a negative error code:

ESRCH - Process pid does not exists.

int **rt_Send**(pid_t pid, void *smsg, void *rmsg, size_t ssize, size_t rsize);

Sends the message pointed to by smsg to the process identified by pid. Any reply will be placed in the buffer pointed to by rmsg. The size of the sent message will be ssize while the size of the reply will

be truncated to a maximum of rsize bytes. The number of bytes transferred will be the minimum of that specified by both the sender and the receiver. After sending a message, the task will block in the RPC state waiting for a reply. If the receiving process is RECEIVED blocked and ready to receive a message, the transfer of data into its address space will occur immediately and the receiver will be unblocked and made ready to run. The sending process will enter the RETURN blocked state. If the receiver is not ready to receive the message, the sender enters the RPC blocked state. The transfer will not occur until the receiver executes a rt_Receive() call. The function returns the actual number of bytes received in the reply message (zero is allowed), otherwise the function returns a negative error code:

- EINVAL - message length invalid.
- ENOMEM - insufficient memory to grow buffer.
- ESRCH - process pid does not exist, or died.

int **rt_Reply**(pid_t pid, void *msg, size_t size);

Replies size bytes of data to the process identified by pid. The number of bytes sent will be the minimum of that specified by both the replier and the sender. The data transfer occurs immediately and the replier does not block. Reply changes the state of the sender from RETURN blocked to READY. The function returns zero on success otherwise returns a negativeerror code:

- EINVAL - message length invalid.
- ENOMEM - insufficient memory.
- ESRCH - process pid does not exist.

pid_t **rt_Creceive**(pid_t pid, void* msg, size_t size, RTIME delay);

A non blocking form of rt_Receive(). The function returns zero if no messages from any pid are available for an immediate transfer when delay is set to zero. When delay is non zero, the function will wait up to delay tics for a message to transfer. The functions returns either a pid if a transfer occurred or zero at the expiration of the delay.

LXRT queue blocks (qBlk's)

qBlk's are simple structures that contain a pointer to a function and the time at which the function must be executed. The qBlk's are linked into a list. A family of functions are provided to manage them.

The functions are of the type:

- void (***handle**r)(void *data, int event)

and therefore the simple structures also include the arguments data and event. The application may or may not use any of the arguments. qBlk's use a dynamically allocated root structure called the tick queue. The tick queue is created with the **rt_InitTickQueue**() function. Any task that will use qBlk's must initialize the tick queue.

The tick queue uses an elementary structure called a QueueHook on which qBlk's are linked to form a queue. Queue management functions are provided to manage queues of qBlk's.

qBlk's are usually managed within the task. When a qBlk executes it is guaranteed that it can manipulate the task data atomically. A qBlk function is like a mini-thread that wakes up when the task is blocked waiting. Scheduling functions are provided to control how the qBlk's will be executed.

Dynamic qBlk's

The tick queue can reference both static and dynamic qBlk's. Plain RTAI kernel real time task can use both static and dynamic qBlk's. LXRT soft and hard real time tasks must use dynamic qBlk's only. qBlk's are always managed and executed in plain RTAI hard real time context even if the code of the qBlk function is in user space.

Dynamic qBlk's are one-shot object. They are initialized from a pool of free qBlk's and they are automatically returned to the free poll before they are executed. The only way to get a dynamic qBlk to repeat is to schedule it with the **rt_qBlkRepeat**() function. In fact, **rt_qBlkWait**() forces a single-shot execution and is usually used with static qBlk's.

The function **rt_qDynInit**() takes the qBlk from the free list if one is available. Otherwise it calls **rt_malloc**() to create one. At completion time, the dynamic qBlk is returned to the free list which gets cleared by calling **rt_qDynFree**().

Plain real time task should not attempt to free the memory themselves. Rather, they should call **rt_qDynFree**(-1) to empty the free list completely. This minor constraint leaves the possibility to trade qBlk's among tick queues in the future.

qBlk functions can re-enter LXRT

For LXRT soft and hard real time tasks, the qBlk function can re-enter LXRT as long as the function

type (as defined in struct rt_fun) is not greater than 1. This constraint will disappear in the near future.

Also, while re-entering the task cannot block in the real time scheduler because the Linux context cannot resume until the qBlk function completes.

qBlk management functions

void **rt_qBlkWait**(QBLK *qblk, RTIME tics);

Insert a qBlk in the Tick queue, after dequeuing it if need be (if it was already queued), to be executed after the given number of ticks have expired. Specifying a tick count of 0 is the normal way of inserting a qBlk after all currently expired qBlk's.

void **rt_qBlkRepeat**(QBLK *qblk, RTIME tics);

Insert a qBlk in the Tick queue, after dequeuing it if need be, to be executed after the given number of ticks have expired. After completion, the qBlk is reinserted in the queue with the same delay if it is not cancelled or dequeued within the qBlk function itself. Notice that a tick count of 0 does not repeat.

void **rt_qBlkSoon**(QBLK *qblk);

Insert a qBlk at the head of the Tick Queue after dequeuing it if need be. The qBlk will be executed before any already expired qBlk.

void **rt_qBlkDequeue**(QBLK *qblk);

Unhook a qBlk from a queue and notify (i.e. execute the QHOOK cancel function) the queue manager. The qBlk is not released. If the qBlk was not queued this function does nothing.

void **rt_qBlkCancel**(QBLK *qblk);

Dequeue if it was hooked, and release a qBlk.

qBlk scheduling functions

void **rt_qLoop**(void);

The application waits for the execution of all pending qBlk's. The function returns when the tick queue is empty.

void **rt_qReceive**(void);

The application receives messages and/or proxies from other tasks while excuting pending qBlk's at the same time.

void **rt_qStep**(void);

The application needs to manage the synchronisation itself as other things may be more important than the pending qBlk's. The return value tells the application if a qBlk is pending or not and if so, when the next qBlk should be executed.

void **rt_qSync**(void).

The application was doing something very important and now needs to execute and release all expired qBlk's in the Tick Queue. qBlk's that expire during this process will also be completed.

Queue mangement functions

It is also possible to create hooks to which qBlk's can be linked to be scheduled later on. This feature can be used to implement a bottom half like mechanism to execute less important functions at a more appropriate time. qBlk's can be used by a real time interrupt handler provided it uses the tick queue of a co-operating real time task.

QHOOK ***rt_qHookInit**
(QHOOK **lnk, void (*cancel)(void *, QBLK *), void *);

Allocate and initialise a QHOOK. Returns zero on error or returns the pointer and sets the link lnk if any.

void **rt_qHookRelease**(QHOOK *qhk);

Release and free the memory of a QHOOK.

void **rt_qBlkBefore**(QBLK *cur, QBLK *nxt);

Insert the qBlk before another qBlk in a queue. If it was hooked it will first be unhooked.

void **rt_qBlkAfter**(QBLK *cur, QBLK *prv);

Insert the qBlk after another qBlk in a queue. If it was hooked it will first be unhooked.

void **rt_qBlkAtH**ead(QBLK *cur, QHOOK *hook);

Insert the qBlk at the head of a queue. If it was hooked it will first be unhooked.

void **rt_qBlkAtTail**(QBLK *cur, QHOOK *hook);

Insert the qBlk at tail of a queue. If it was hooked it will first be unhooked.

QBLK ***rt_qBlkUnhook**(QBLK *qblk);

Remove the qBlk from a queue. The qBlk is not released and the cancel function not called.

LXRT Exception Handling

The LXRT module now mounts a handler to deal with processor generated exceptions. These exceptions or traps use the lower 32 vectors of the IDT. The previous approach was to simply ignore trap handling. It worked as long as the real time code was bug free. Often, the slightest mistake would crash or reboot the computer or damage something that came back later to make the user's life miserable.

A default trap handler is installed by the schedulers. The API function **rt_set_rtai_trap_handler**() is provided to change the default trap handler algorithm that simply suspends a faulty RTAI real time task.

LXRT implements a more sophisticated trap handler to deal adequately with exceptions in the following cases:

- A soft real time task is running in user space. No special action taken and the exception is simply passed to the Linux handler. As explained above, function do_exit() will call linux_process_termination() to delete the agent task and release any real time resources the process may have registered.

- A plain real time task is running. As for the default handler the task is suspended. We choose not to delete the task to make it possible for a module to dump the task structure and its stack after the fact.

- The agent of a soft real time task is running in the kernel. A service request is pended to send the signal SIGKILL to the Linux task then the function lxrt_suspend() is called to stop the agent. Notice that control never returns to the trap handler.

- A hard real time task is running in user space. In this case we call function give_back_to_linux() from the trap handler to return the process to soft real time and then do_exit(SIGKILL) is called to terminate the process.

- A hard real time task is running in the kernel. Here function lxrt_suspend() is called to return to the Linux context followed by a give_back_to_linux() call to come back to soft real time mode and finally a do_exit(SIGKILL) call is done to kill the process. Function linux_process_termination() completes as explained above.

- A special case occurs if a qBlk is executing a user space function. In this case it is necessary to reload the local descriptor of the Linux current process that was changed by the the rthal.switch_mem() call in function exec_func(). After the reload (i.e. a second call of function rthal.switch_mem() with appropriate arguments) the exception is dealt with similarly to the plain real time task case described above.

Notice that there is not much that can be done if an exception like a division by zero occurs in an interrupt handler. However, that remains true for all systems.

The CPL (current privilege level) is not checked if the exception is generated by the processor. Thus trap and interrupt descriptors have a DPL (descriptor privilege level) of zero. The IF (interrupt enabled flag) flag is not affected by processor generated exceptions.

The DPL level of a trap descriptor can be changed to 3 to allow calling the trap with the int $n instruction from user space. Notice that there is no error code pushed on the stack in that case.

A stack switch occurs if the handler's privilege level is smaller than the CPL of the interrupted procedure.

Page fault exceptions occur all the time in the Linux context. Usually the user space program needs a VMA not currently mapped in physical memory. An error can occur if the kernel tries to access user space memory with a bad pointer argument. The Linux page fault handler deals nicely with that problem as explained in file exception.txt in the kernel documentation.

The LXRT extendability concept

The original LXRT used an array of structures rt_fun_entry called rt_fun to hold the function pointers so that one of the system call register argument could be used to figure out the matrix index of the function pointer and the number of word long arguments to pass to the function.

The matrix rt_fun is loaded in memory as global data when LXRT is started by insmod. Some other module

could also define a similar matrix of rt_fun structures. Clearly, if LXRT used a pointer to access the base matrix of rt_fun structures, it would be able to access a different matrix using a simple pointer management indexing scheme.

This is the concept behind the RT_LXRT_COM module. To support extendability, LXRT now uses a matrix of 16 different pointers to arrays of rt_fun_entry structures called rt_fun_ext. The pointer to LXRT's base matrix is located at index 0. Hence, up to 15 other modules can use the LXRT system call inteface.

The end result is that RT_COM kernel module functions like:

- **rt_com_setup**(…)
- **rt_com_set_param**(…)
- **rt_com_read**(…)
- **rt_com_write**(…)
- **rt_com_bout_free**(…)
- **rt_com_clr_in**(…)
- **rt_com_clr_out**(…)
- **rt_com_set_mode**(…)
- **rt_com_rd_modem**(…)
- **rt_com_wr_modem**(…)
- **rt_com_error**(…)

where implemented easily with a simple header file and a trivial module as part of the symmetric API without modyfing LXRT at all. In other words, they can be used in user space (both soft and hard real time modes).

RTAI MINI-LXRT Tasklets Support Module

The MINI_RTAI_LXRT tasklets module that is explained hereafter adds an interesting new feature along the line of a symmetric API (pioneered by DIAPM-RTAI) of all real time services inter-intra kernel and user space both for soft and hard real time. As a result, you have an even wider spectrum of development and implementation options, allowing maximum flexibility with uncompromised performances. And of course, all LGPL open source.

New services: tasklets and timers

The new services provided can be useful when hard real time tasks, both in kernel and user space, do not need any RTAI scheduler services that could lead to a task block. This **critical** constraint should be clearly understood.
Such tasks are called **tasklets** and can be of two kinds:

- A simple tasklets,

- Timed tasklets (timers).

It must be noted that only timers need to be made available both in user and kernel space. In fact, simple tasklets in kernel space are nothing but standard functions that can be directly executed by simply calling them, so there is no need for any special treatment. However, in order to maintain full usage symmetry, and to continue to allow the possibility of porting applications from one address space to the other, tasklets functions have been implemented so they can be used in whatever address space.

Note that the Linux kernel offers similar services. They are not exactly the same because of the RTAI symmetrical API implementation, but the basic idea behind them is fairly similar.

It should be clear that for such tasks the standard hard real time tasks available with RTAI and LXRT schedulers can be a waist of resources and the execution of simple, possibly timed, functions can often be more than enough.

Examples of such applications are timed pollings and simple Programmable Logic Controllers (PLC) like sequencing services. Obviously, there are many other instances that justify the use of tasklets, either simple or timed. In general, such an approach can be a very useful complement in controlling complex machines and systems, both for basic and support services.

The implementation

The MINI-LXRT implementation of timed tasklets relies on a server support task that executes the related timer functions, either in one-shot or periodic mode, on the base of their time deadline and according to their user assigned priority.

As explained above, plain tasklets are just functions executed from kernel space. Their execution needs no server and is simply triggered by calling the user specified tasklet function at due time, either from a kernel task or interrupt handler in charge of their execution when they are needed.

Once more it is important to recall that only non blocking RTAI scheduler services can be used in any tasklet function. Services that can block must absolutely be avoided, as they will deadlock the timers server task, executing task or interrupt handler, whichever applies, with the result that no other tasklet functions will be executed.

User and kernel space MINI-LXRT applications should cooperate and synchronize by using shared memory.

It has been called MINI-LXRT since it is a kind of light hard real time server that can substitute both RTAI and LXRT, if the constraints explained above are satisfied. The MINI-LXRT module can be used in kernel and user space, with any RTAI scheduler.

Its implementation has been very easy to accomplish, as it is nothing but what its name implies. LXRT provided all the needed tools. In fact, it duplicates a lot of LXRT so that its final production version will be fully integrated with it. However, at the moment, it cannot work with LXRT.

As already done for shared memory and LXRT, the function calls for Linux processes are inlined in the file **mini_rtai_lxrt.h**. This approach has been preferred to a library since it is simpler and more effective. The calls are short and simple so that even if it is likely that only a few calls are used for a typical process, they do not add significantly to the size of the program.

MINI-LXRT Services

The services made available by the MINI-LXRT module (functions, macros and variable names are self explanatory, see also example test.c) are:

struct rt_tasklet_struct ***rt_tasklet_init**(void)

void **rt_tasklet_delete**(void)

int **rt_insert_tasklet**(struct rt_tasklet_struct *tasklet, void (*handler)(unsigned long), unsigned long data, unsigned long id, int pid)

void **rt_remove_tasklet**
(struct rt_tasklet_struct *tasklet)

struct rt_tasklet_struct ***rt_find_tasklet_by_id**
(unsigned long id)

void **rt_tasklet_exec**(struct rt_tasklet_struct *tasklet)

struct rt_tasklet_struct ***rt_timer_init**(void)

void **rt_timer_delete**(void)
int **rt_insert_timer**(struct rt_tasklet_struct *timer, int priority, RTIME firing_time, RTIME period, void (*handler)(unsigned long), unsigned long data, int pid)

void **rt_remove_timer**
(struct rt_tasklet_struct *timer)

void **rt_set_timer_priority**
(struct rt_tasklet_struct *timer, int priority)

void **rt_set_timer_firing_time**
(struct rt_tasklet_struct *timer, RTIME firing_time)

void **rt_set_timer_period**
(struct rt_tasklet_struct *timer, RTIME period)

#define **rt_fast_set_timer_period**(timer, period)

int **rt_set_timer_handler**
(struct rt_tasklet_struct *timer, void (*handler)(unsigned long))

#define **rt_fast_set_timer_handler**(timer, handler)

void **rt_set_timer_data**
(struct rt_tasklet_struct *timer, unsigned long data)

#define **rt_fast_set_timer_data**(timer, data)

void **rt_tasklets_use_fpu**(int use_fpu)

RT_TASK ***rt_timers_server**(void)

The rt_fast... timer related macros can be safely used in kernel space as alternative to their standard equivalents when the related data and timer structure address are available.

Remember to always include the header file **rtai_timers.h** found in the module directory. It defines struct rt_tasklet_struct and all the tasklet functions prototypes and macros.

The functions **rt_tasklet_init**(), **rt_timer_init**(), **rt_tasklet_delete**() and **rt_timer_delete**() are meant to be used in user space only because the timer structure must be allocated dynamically in kernel space. They become empty macros in kernel space where one must allocate the tasklet structure.

FPU Support and other technicalities

The timers server task assumes that timer functions never use the Floating Point Unit (FPU). Otherwise, the function **rt_tasklets_use_fpu**() should be used to enable the use of the FPU if it is needed by **any** timer function, both in kernel and user space. The same applies to simple tasklets in user space.

In the kernel, the task, or interrupt handler, executing any tasklet must enable the FPU by appropriately using the fsave and frestore, and clearing clts (see rtai.h for the related macros).

The function **rt_timers_server**() returns the pointer to the timer server itself. It has been useful during development and it is maintained as an undocumented back door feature. Recall the basic rule that one should **never** do a blocking call.

The timers server runs on a 2K stack, which should be enough to run most timers tasklet functions in kernel space. If one needs a larger stack, one should either recompile **mini_rtai_lxrt.c** after setting the macro STACK_SIZE in mini_rtai_lxrt.h to what you want, or simply load the timers server module using "ldmod ./rtai_timers StackSize=<xxxx>", where <xxxx> is the new stack size.

In user space, the tasklet (or timer function) runs within the memory of the process owning the handler function so no problem should arise. Note however that you must lock all the process memory (and pre-grow the stack) so that it cannot be swapped out. So pre-grow all the memory the process will need, see mlockall usage in Linux manuals or use the lxrtlib function **lock_all**().

There are also many very useful test cases that demonstrate the use of most services, both in kernel and user space (see directory tests and related run files).

Clearly, this module is a beta release and there is still work to be done.

LXRT Unix server

This example introduces the basic frame of a generalized, per process, UNIX server to be used by hard real time LXRT applications that want to access Linux IO services.

Clearly, hard real time tasks are timed by Linux while using the server. So the garantee to satisfy hard real timing constraints vanishes, even if they remain under control of the LXRT scheduler (i.e. Linux cannot schedule them directly). There is a partial loss of efficiency with respect to plain Linux usage. In fact we do not like such a solution very much and prefer an application specific server. However users who want to have it simple will find it useful, especially during development.

How does it work?

The module shows the power of remote procedure call as a unified inter tasks communication and synchronization mechanism. There are two switches per Linux service request, a standard micro kernel way of working. The context switches and the need to copying some data, are responsible for most of the penalty the user has to pay for using this server. The response time is not so bad anyhow. Those with QNX experience will understand the concept easily.

The function **rt_start_unix_server**() called before the switch to hard real time mode forks the program **./unix_server** who will act as an agent to execute the IO functions in soft real time mode (POSIX with SCHED_FIFO scheduling). Shared memory is used to avoid using **memcpy**() and thus minimize call overhead.

All the native IO calls are used by the agent and their return values returned "as is" to the real time task.

The function **rt_end_unix_server**() call instructs the server to release the shared memory, rt_task_delete() his real time agent, and exit normally. Any open files are not closed automatically before exiting for now.

Once more if you need faster response time use your own server. Recall that Linux should not be your main concern while you are running hard real time in user space. It should be needed just for some support services to be executed sporadicaly when hard real time service is not requested.

API Functions prototypes

The API is pretty much standard except for the function name rt_ prefix to indicate that the call can be made while in hard real time mode. Refer to your libc6 manual if you need to understand how the underlying Unix calls behave. The following functions have been implemented so far:

void **rt_start_unix_server**(void *task, int rt_prio, int shmsize)

int **rt_end_unix_server**(void)

int **rt_scanf**(const char *fmt, ...)

int **rt_printf**(const char *fmt, ...)
int **rt_open**(const char *pathname, int flags, mode_t mode)

int **rt_close**(int fd)

int **rt_write**(int fd, void *buf, size_t count)

int **rt_read**(int fd, void *buf, size_t count)

int **rt_select**(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)

off_t **rt_lseek**(int fd, off_t offset, int whence)

int **rt_sync**(void)

int **rt_ioctl**(int d, int request, unsigned long argp, int size)

LXRT liblxrt

Efforts were made to provide GUI programmers with the possibility to carry out their work without having to install copies of the kernel and RTAI source trees. Many excellent GUI programmers are not interested in and/or do not care about kernel internals. The new header file **rtai_lxrt_user.h** allows to do that.

There is also the issue that GUI applications are often C++ based: KDE, Qt and QpTreads are packages that come to mind. File **rtai_lxrt_user.h** also makes life easier for the g++ compiler. A C++ example was added to the source tree to show how a C++ kernel module can be written.

Tomasz Motylewski contributed the file touchall.c that provides the function **lock_all**(). This function grows the stack and locks all the program memory pages. The usefulness of a library arose from this contribution as all user space programs that use LXRT should call lock_all() - or at least do the equivalent.

Notice that any exception 14 while in hard real time mode is interpreted as a program error and the program terminated by the trap handler. No attempts were made to map pages while in hard real time mode. It would have contradicted basic real time principles.

The library builds both static and shared objects and thus provides the application developer with all the flexibility he or she expectd to find in the Linux environment.

Linux Trace Toolkit

Modern software systems are ever more complex. Systems based on RTAI are no exception. Its real-time nature and the fact that it takes control of the Linux kernel make nothing to diminish this complexity. Hence, understanding the dynamic behavior of RTAI based systems can be difficult, even to the best of insiders. The RTAI extensions to the Linux Trace Toolkit take this complexity away by providing developers with the capability of tracing and reconstructing dynamic system behavior.

To accomplish this, trace statements are inserted in the execution path of key system code. Each trace statement indicates the event that occurred on the corresponding path and provides a concise description of the event. When Linux is compiled with RTAI trace support, the corresponding RTAI trace statements will generate calls to the RTAI trace facility. This facility is the primary link between the instrumented components of RTAI and the trace driver which takes care of logging the traced events. If Linux is compiled without RTAI trace support, the trace statements are void and result in no calls at all (RTAI remains unmodified).

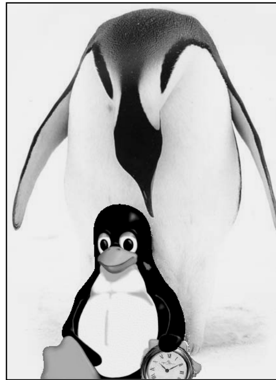
The trace driver's primary role is to buffer event descriptions into its buffers. To increase flexibility, the driver's behavior can be modified through the ioctl() interface. Taking care of this configuration and taking care to commit the data buffered, the trace daemon acts as the primary link between the developer and the trace system. By invoking it with the adequate parameters, the developer has total control on the trace process, from it's duration to the events traced and the trace buffer sizes.

Once the trace process is launched, all specified events are traced and committed to a trace file. Once the tracing is complete, this trace file is then used by the visualization and analysis tool provided with LTT to reconstruct the system's behavior during the trace. This tool can be used both as a command-line tool and as a graphical tool. In the later form, it provides the developer with a control-graph view of the system which enables him to see the different transitions in control that occurred during the trace and the reasons of their occurrence. Furthermore, the tool uses the information collected to provide exact statistics about different components of system performance. Contrary to other means of instrumentation, LTT makes no approximations and does not rely on

samples. Rather, it provides an exact description of the system's behavior.

Using the information provided, the developer can isolate performance bottlenecks, solve synchronization problems and confirm his understanding of the system's behavior. The commercial RTOS world is no stranger to such a capability as many RTOS vendors provide such a capability for their products. It is worth noting that such a trace system was used by the JPL (Jet Propulsion Laboratory) engineers to find the reason why the Mars Pathfinder constantly reset and, consequently, implement a solution. See the link below for the full story.

LTT is available at the Opersys Home Site (see the link below) and is distributed under the terms of the GPL. Apart from and prior to providing RTAI trace capability, LTT was designed to provide trace capability to the Linux kernel. This capability remains available and is independent of the capability of tracing RTAI, though both capabilities can be combined.



Acknowledgements

Acknowledgements to the RTAI developers, listed below in any order, who have contributed towards the fast production of this position paper. It is a pleasure to be part of such a team were co-operation and consensus building are solid foundations for greater achievements to come.

Lorenzo Dozio	(dozio@aero.polimi.it)
Stuart Hughes	(stuarth@lineo.com)
Brendan Knox	(brendank@lineo.com)
David Schleef	(ds@schleef.org)
Ian Soanes	(ians@lineo.com)
Pierre Cloutier	(pcloutier@poseidoncontrols.com)
Paolo Mantegazza	(mantegazza@aero.polimi.it), Maintainer
Steve Papacharalambous	(stevep@lineo.com)
Karim Yaghmour	(karym@opersys.com)
Trevor Woolven	(trevw@lineo.com)
Giuseppe Renoldi	(grenoldi@usa.net)
Tomasz Motylewski	(motyl@stan.chemie.unibas.ch)
Emanuele Blanchi	(blanchi@aero.polimi.it)

Special acknowledgements are in order for DIAPM and Paolo Mantegazza in particular for his dedication and commitment to RTAI. Paolo's innovative concepts like LXRT take root in more than 15 years of research and experimental work in the field of PC based control systems.

Finally, acknowledgements are also in order for the founders of real time under Linux, Victor Yodaiken and Michael Barabanov.

References and Usefull URL's

- DIAPM-RTAI Home and Download Site,
  http://www.aero.polimi.it/projects/rtai
- Real Time Linux,
  http://www.realtimelinux.org
- Linux Trace Toolkit Home and Download Site,
  http://www.opersys.com/LTT
- Real Time and Embedded HOWTO,
  http://www.mech.kuleuven.ac.be/%7Ebruyninc/rthowto/index.html
- Reeves, G., "What really happened on Mars?",
  http://research.microsoft.com/~mbj/Mars_Pathfinder/Authoritative_Account.html
- SRR -- QNX API compatible message passing for Linux.
  http://www.holoweb.net/~simpl/srr_sam.html