DIAPM
RTAI Programming Guide 1.0

disclaimer

Lineo, Inc. makes no representations or warranties with respect to the contents or use of this manual, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Lineo, Inc. reserves the right to revise this publication and to make changes to its content, at any time, without obligation to notify any person or entity of such revisions or changes.

Lineo, Inc. makes no representations or warranties with respect to any Lineo software, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Lineo, Inc. reserves the right to make changes to any and all parts of Lineo software, at any time, without any obligation to notify any person or entity of such changes.

copyrights

Lineo, Inc.
390 S. 400 W.
Lindon, Utah 84042
USA

http://www.lineo.com

RTAI Programming Guide 1.0
September 2000

The DIAPM RTAI homepage can be found at: http://www.rtai.org

Contents

chapter 3 RT Task Programming Basics

preface

About This Guide

Acknowledgments

Each of the individuals listed in the table below has contributed significantly to this document in some form or another. It is a very difficult task to construct an all inclusive list of those who have contributed to specific projects, and we apologize to those who think that their contribution has gone unnoticed.

Contributor	Primary Contributions
Paolo Mantegazza, DIAPM	RTAI Chief Architect
E. Bianchi and L. Dozio, DIAPM	Implementation, Testing, Documentation
Mike Angelo, Lineo, Inc	Technical writer
David Beal, Lineo, Inc.	Technical writer
Pierre Cloutier, Poseidon Controls Inc.	LXRT architecture, Dynamic Memory Allocation Algorithm
Stuart Hughes, Lineo, Inc.	Maintenance of portable FIFOs, LXRT PERL bindings, run-time data debugger, kgdb enhancement, common API implementation and examples.
Gabor Kiss, Computer and Automation Institute of Hungarian Academy of Sciences	API documentation updates and revisions
Tomasz (Tomek) Motylewski	Shared Memory
Steve Papacharalambous, Lineo, Inc.	RTAI /proc system, POSIX 1003.1c pthreads, mutexes and condition varible implementation, RTAI dynamic memory allocation implementation.
David Schleef, Lineo, Inc.	Real-Time Ethernet, COMEDI real-time Linux device driver toolset, rt_printk

Ian Soanes, Lineo, Inc.	Named FIFOs
Phil Wilshire, Lineo, Inc.	RTAI evaluation tests, technical review, code contributions
Trevor Woolven, Lineo, Inc.	POSIX 1003.1b pqueues.
Karim Yaghmour, OperSys Inc.	Linux Trace Toolkit, various RTAI bug fixes,
Victor Yodaiken, Michael Barabanov, et. al.	Originators of RTLinux

As RTAI is indeed a collective work representing many hours of effort by each of the individuals noted above, the authors request full and visible attribution which includes the table shown directly above, to each of these contributors on **any fractional or whole reprint or modification** of this document.

Support and Training for RealTime Linux

As can be seen above, Lineo's talent pool includes several developers who have been critical to RTAI's core services and growing feature set. These same individuals provide corporate level and development team support for design teams who are now developing products based on real-time Linux (either the RTLinux or RTAI implementations).

Lineo's Industrial Solutions Group team members are experts in areas such as:

- Real-time Linux development tools
- Real-time networking under Linux
- Industrial shared memory solutions
- Real-time device drivers
- Code porting from proprietary RTOS to RealTime Linux
- Porting RealTime Linux to new CPUs

For information on our many options for support services and on our one and four day RealTime Linux training courses, please contact us at:

Lineo, Inc.
390 South 400 West, Lindon, UT 84042
Phone: (801) 426-5001, Fax: (801) 426-6166
info@lineo.com

chapter 1

Introduction

This RealTime Linux Programmer's Guide is intended to supplement, rather than replace, the documentation provided with the RTAI distribution. In-fact, you will note that we have reprinted numerous portions from RTAI's documentation set. This document is designed to provide a complete top to bottom understanding of programming hard real-time tasks under the Real-Time Applications Interface for Linux, for those who are familiar with standard Linux programming.

Preamble: What is Real-Time?

The industrial and military sectors require varying levels of 'real-time' computer response depending on the specific nature of each task to be performed. Consequently, three different definitions of 'real-time' can be illustrated by a battlefield scenario where soldiers in the field provide 'real-time' data which is ultimately sent to the commander's 'real-time' tactical display which provides information used to determine that a missile (using a 'real-time' computer system) should be launched.

The 'real-time' data from the troops can be compared to the now familiar 'real-time stock quote', providing information that was current within the last few seconds or perhaps minutes. This can be referred to as 'human real-time' since short delays in the tactical data provided from the field are obscured by the much longer human delays associated with sorting and correlation.

The video display observed by the commander illustrates 'soft real-time', where the loss of an occasional frame will not cause any perceived video degradation, provided that the *average* case performance remains acceptable. Although techniques such as interpolation can be used to compensate for missing frames, the system remains a soft-real time system because the actual data was missed, and the interpolated frame represents derived, rather than actual data.

‘Hard real-time’ is illustrated by the control system of a high-speed missile because it relies on *guaranteed and repeatable* system responses of thousandths or millionths of a second. Since these control deadlines can never be missed, a hard real-time system cannot use average case performance to compensate for worst-case performance. Thus, hard real-time systems are required for the most technically challenging tasks.

Since an embedded system often performs only a single task, the differences between soft and hard real-time for these applications are not as critical as one would think. However, as true multi-tasking operating systems, such as Linux, are adopted for use in increasingly complex systems, the need for hard real-time often becomes apparent.

To further confuse the real-time issue, the general term “Real-Time Operating System (RTOS)” is used to refer to one that can provide either hard or soft real-time capabilities but not necessarily both. Thus all operating systems labeled as “RTOS” are not created equally.

The Real-Time Linux Solution

The real-time Linux scheduler treats the Linux operating system kernel as the idle task. Linux only executes when there are no real time tasks to run, and the real time kernel is inactive. The Linux task can never block interrupts or prevent itself from being preempted. The mechanism that makes this possible is the software emulation of interrupt control hardware. When any code in Linux tries to disable interrupts, the real time system intercepts the request, records it, and returns it to Linux. In fact, Linux is not permitted to *ever* really disable hardware interrupts, and hence, regardless of the state of Linux, it cannot add latency to the interrupt response time of the real time system. When an interrupt occurs, the real time kernel intercepts the interrupt and decides what to dispatch. If there is a real time handler for the interrupt, the appropriate handler is invoked. If there is no real time interrupt handler, or if the handler indicates that it wants to share the interrupt with Linux, then the interrupt is marked as pending. If Linux has requested that interrupts be enabled, any pending interrupts are enabled, and the appropriate Linux interrupt handler invoked - with hardware interrupts re-enabled. Regardless of the state of Linux: running in kernel mode; running a user process; disabling or enabling interrupts; the real-time system is always able to respond to an interrupt.

Real-time Linux decouples the mechanisms of the real time kernel from the mechanisms of the general purpose Linux kernel so that each can be optimized independently and so that the real-time kernel can be kept small and simple.

From a maintenance perspective, this de-coupling allows the Real-Time Linux kernel to be easily and quickly adapted to follow changes in the mainstream Linux kernel.

Real-time Linux has been designed so that the real time kernel never waits for the Linux side to release any resources. The real time kernel does not directly request memory, share spin locks, or synchronize data structures, except in tightly controlled situations. For example, the communication links that are used to transfer data between real time tasks and Linux processes are non-blocking on the real time side. There is never a case where the real time task waits to queue or dequeue data.

One of the fundamental design philosophies of Real-time Linux is to let the Linux operating system do as much as is practicable. Typical examples include system and device initialization, and blocking dynamic resource allocation. Any thread of execution that can be blocked when there are no available resources cannot have real time constraints. Real-time Linux relies on the Linux loadable module mechanism to install components of the Real-Time system, which keeps it extensible and modular. Loading a Real-Time module is not a real-time operation, and so Linux can do it. The primary function of the Real-Time kernel is to provide direct access to the raw hardware for real time tasks so that they can execute with minimal latency and maximal processing resource, when required.

There are two primary variants of hard real-time Linux available: RTLinux and RTAI. **RTLinux** was developed at the New Mexico Institute of Technology by Michael Barabanov under the direction of Professor Victor Yodaiken. **Real-Time Application Interface (RTAI)** was developed at the Dipartimento di Ingeneria Aerospaziale, Politecnico di Milano by Professor Paolo Mantegazza.

Under both RTLinux and RTAI, all interrupts are initially handled by the real time kernel and are passed to Linux only when there are no active real time tasks. Changes to the Linux kernel are minimized by providing the kernel with a software emulation of the interrupt control hardware. Thus, when Linux has disabled interrupts, the emulation software will queue interrupts that have been passed on by the real time kernel. This is achieved by installing a layer of emulation software between the Linux kernel and the interrupt controller hardware, and replacing all occurrences of *cl*i, *st*i, and *iret* in the Linux source code with emulating macros. When the Linux kernel would normally disable interrupts, this event is logged by the emulation software, but interrupts are not actually disabled. When an interrupt occurs, the emulation software checks to see whether Linux has interrupts enabled, if so the interrupt is delivered to the Linux kernel. If not, the interrupt is held pending the Linux kernel re-enabling interrupts. In this way, the Linux kernel does not have direct control over

interrupts, and cannot delay the processing of real time interrupts, as these interrupts do not pass through the emulation software. Instead, they are delivered direct to the real time kernel. This also means that the scheduling of real time tasks cannot be delayed by Linux.

Fundamentally, RTAI, RTLinux and applications written to take advantage of them operate in the same way. The Real Time kernel, all their component parts, and the real time application are all run in Linux kernel address space as kernel modules. As each kernel module is loaded it initializes itself ready for system operation. The kernel modules can be removed from the kernel on completion of the real time system operation. Kernel modules can be loaded and unloaded dynamically, either by an application or by taking advantage of the automatic module loading features of Linux itself.

The advantages of running the real time system in Linux kernel address space is the task switch time for the real time tasks is minimized, and Translation Look-aside Buffer (TLB) invalidation is kept to a minimum as are protection level changes. Another advantage of making use of kernel loadable modules is that it aids system modularity. For example, if the scheduler is unsuitable for a particular application, then the scheduler module can be replaced by one that meets the needs of the application.

One of the main disadvantages of running in Linux kernel address space is that a bug in a real time task can crash the whole system, as there is no separate protected memory space for an individual RT task.

As mentioned earlier, there are two current ‘flavours’ of Real-Time Linux: RTAI and RTLinux. The remainder of this document is devoted to describing RTAI.

The RTAI Solution

RTAI provides deterministic and preemptive performance in addition to allowing the use of all standard Linux drivers, applications and functions.

RTAI was initially developed by The Dipartimento di Ingeneria Aerospaziale Politecnico di Milano (DIAPM) as a variant of the New Mexico Institute of Technology's (NMT) RTLinux, at a time when neither floating point support nor periodic mode scheduling were provided by RTLinux. Since then, RTAI has added many new features without compromising performance.

RTAI's growing list of features now includes:

- POSIX 1003.1c compatibility (Pthreads, including mutexes and condition variables)
- POSIX 1003.1b compatibility (Pqueues only)
- Traditional RTOS IPCs including: Semaphores, mailboxes, FIFOs, shared memory, and RPCs
- Dynamic Memory Allocation - non-blocking in the Real-Time domain.
- PERL Bindings – which allow scheduling of soft real-time tasks from the PERL scripting language, without the need to know any C or compiler specific information.
- /proc interface – which provides information on the real-time tasks, modules, services and processes extending the standard Linux /proc file-system support.
- LXRT – which allows the use of the RTAI system calls from within standard user space
- UniProcessor, Multi-UniProcessor and Symmetric Multi-Processor (SMP) support
- FPU support
- One-shot and periodic schedulers

RTAI's performance is very competitive with the best commercial Real Time Operating Systems (such as VxWorks, QNX etc), offering typical context switch times of 4 uSec, 20 uSec interrupt response, 100 KHz periodic tasks, and 30 KHz one-shot task rates. The main limitation being imposed by the system hardware, rather than the real-time software itself.

Like standard Linux, RTAI is open source and thus it can be downloaded from the internet and manually patched into a Linux system, or, alternatively it can be easily installed using the Lineo Industrial Solutions Group, Embedix-RealTime installation CD which allows real-time Linux features, services, and tools to be applied on top of an existing and already configured Linux system.

Notes on RTAI's Release Numbering Scheme

Previously, RTAI releases followed a simple incremental pattern: rtai-0.9, rtai-1.0, rtai-1.1 etc. This scheme has now been altered to reflect the way releases of RTAI follow release versions of the Linux kernel. This document has been based upon RTAI-22.2.4, which is explained as follows:

22=>Linux kernel version 2.2.x

2=>‘Stable’ release (3 => development release)

4=>Sub-version 4

When first envisioned in 1996, RTAI’s real-time technique required three fundamental problems to be solved. First, the interrupts must be captured. Then, the real-time schedulers and services must be implemented. Finally, the real-time application needs a means to interface with RTAI’s schedulers and interrupt handlers. The following chapters describe how RTAI has achieved these aims.

Architecture

The Real-Time Hardware Abstraction Layer (RTHAL)

The first problem – that of capturing and redirecting the interrupts, was addressed by creating a small Real-Time Hardware Abstraction Layer (RTHAL) which intercepts all hardware interrupts and routes them to either standard Linux or to real-time tasks depending on the requirements of the RTAI schedulers. Interrupts which are meant for a scheduled real-time task are sent directly to that task, while those interrupts which are not required by any scheduled real-time task are sent directly to standard Linux where they are handled according to normal needs. In this manner, the RTHAL provides a framework onto which RTAI is mounted with the ability to fully pre-empt Linux.

A key component of the RTHAL is the Interrupt Descriptor Table (IDT), which provides a set of pointers, which defines to which processes each of the interrupts should be routed. The IDT gives the ability to easily implement or disable all RTHAL services, allowing the developer to easily track low level bugs under the same kernel configuration, but with and without the HAL in place.

The RTHAL structure, which makes it easy to replace any of the standard Linux interrupt handling and enable/disable functions with alternative ones, is shown below.

```
struct rt_hal {
struct desc_struct *idt_table;
void (*disint)(void);
void (*enint)(void);
unsigned int (*getflags)(void);
```

```
void (*setflags)(unsigned int flags);
void (*mask_and_ack_8259A)(unsigned int irq);
void (*unmask_8259A_irq)(unsigned int irq);
void (*ack_APIC_irq)(void);
void (*mask_IO_APIC_irq)(unsigned int irq);
void (*unmask_IO_APIC_irq)(unsigned int irq);
unsigned long *io_apic_irqs;
void *irq_controller_lock;
void *irq_desc;
int *irq_vector;
void *irq_2_pin;
void *ret_from_intr;
struct desc_struct *gdt_table;
int *idle_weight;
};
```

Upon activation of RTAI within the RTHAL, the following actions occur:

- Generation of a duplicate copy of the standard Linux interrupt descriptor table and interrupt handlers. This new IDT becomes the valid table when RTAI is activated.
- Redirection of the RTHAL interrupts, interrupt enable/disable functions, and flags save/restore functions to the trapped RTAI equivalents. Placement of all the appropriate functions and data into a single structure, the RTHAL, makes it easy to trap these items so that they can be dynamically changed from the standard Linux entries to entries for the hard real time kernel.
- Change of the handler functions in the new *idt_table* to the RTAI dispatchers so that it takes control of the system hardware and its interrupts.
- Provision of a timer and locking services for the real time domain. The timers services provide control of the 8254 and APIC timers.

Under these conditions, Linux is run only as the idle task to the real time domain. Some readers may wonder what happens to the Linux "real-time clock" when Linux itself is inactive due to pre-emption by a RTAI task. RTAI handles this properly by passing up the true clock value upon re-activation of Linux, thus preserving the correct time settings and keeping Linux oblivious to its presence.

Advantages and Disadvantages of the RTHAL

The main advantages of using a RTHAL approach compared to a relatively kernel intrusive implementation such as that used by RTLinux include:

- The changes needed to the standard Linux kernel are minimal, a few lines in eleven source files plus configuration additions to three files in the build structure, (Makefile, configuration files etc). This lower intrusion on the standard Linux kernel improves the code maintainability, and makes it easier to keep the real time modifications up-to-date with the latest release of the Linux kernel.
- The real time extensions can easily be removed by replacing the interrupt function pointers with the original Linux routines. This is especially useful in certain debugging situations when it is necessary to remove the extensions, and when verifying the performance of standard Linux with and without the real time extensions.

n o t e : The developers of RTAI are encouraged by the architecture of the recent pre-beta releases of RTLinux v3.0 that have now adopted an approach very similar to RTAI's HAL.

The Linux kernel suffers a slight, but essentially negligible, performance loss when RTAI is added due to the indirection through pointers to the interrupt mask, unmask and flag functions.

In consideration of both strengths and weaknesses, this technique has shown itself to be both efficient and flexible because it removes none of the capability of standard Linux, yet it provides guaranteed scheduling and response time for critical tasks.

Real-Time Service Implementation

The second problem -- that of providing real-time schedulers and services -- is addressed by leveraging the existing Linux module loading capability to provide the real-time schedulers, FIFOs, shared memory, and other services as they are needed. This module-based architecture yields a system that is easily expandable and customizable according to only the services that you require. Thus, if you don’t need shared memory or POSIX, you don’t load those modules.

Real-Time Task Implementation

The third problem is solved, again by using the module loading services of Linux. Recall that the HAL and real-time modules of RTAI effectively run standard Linux as the lowest priority task, with the ability (see below) to insert your application-specific real-time task at a higher priority. In general real-time Linux tasks run with kernel modules (although extended LXRT is changing this requirement) where they have direct access to the HAL and RTAI service modules.

Those of you who are familiar with the Linux kernel will note that neither the memory space occupied by the kernel or its associated modules, is provided any protection against undesired read/write access. Thus, an improperly implemented kernel module can over-write critical areas of system memory. This memory over-write will generally require a system reboot, but it can, in exceptional circumstances require a complete re-installation of the operating system.

Portable Real-Time Task Implementation

While the issues related to unprotected system memory space are mostly relevant during the development phase of a kernel module it certainly is a major inconvenience during development, as once the system has been “damaged”, it is difficult to debug and may only become operable again by rebooting.

To aleviate this, the developers of RTAI have produced the LXRT (Linux Real-Time) module that provides the ability to develop real-time applications from within standard user space using the same RTAI API.

RTAI now also includes the ability to provide hard real-time tasks from userspace using the extended-LXRT feature. Although this provides a protected MMU context for the real-time task, it still lacks the necessary trap handler to provide fault recovery during development. This is being address as part of the on-going development

Since Linux, like all UNIX systems requires that kernel modules be recompiled for each version of the kernel that it will be linked to, real-time tasks are generally not portable across machines running different kernel versions. This requirement means that kernel module style real-time tasks can only be deployed on machines that run the same kernel configuration as the development platform. Consequently, if you wish to provide a kernel module based real-time application, you must either:

- Provide the application's source code so that the end user can compile and then install the modules as required, or

- In addition to the application, you must provide an associated kernel (of the correct version) to be compiled and installed by the user.

While either of these solutions is often technically acceptable, the use of LXRT helps one avoid the kernel dependency issue altogether because it allows a real-time task to run from standard user space – provided that the deployment platform includes (and loads) RTAI and the LXRT features. In this case, it is an easy matter to load the LXRT modules (in addition to those required for standard RTAI services) and to provide the application as a standard user space task.

RTAI's Real-Time Services

RTAI's full feature set can be broken down into a set of basic services – such as the schedulers, FIFOs, and shared memory, and a set of advanced features such as POSIX and dynamic memory allocation.

Both basic and advanced services are provided via kernel modules, which can be loaded and unloaded using the standard Linux *insmod* and *rmmod* commands. Although the *rtai* module is required every time any real-time service is needed, all other modules are necessary only when their associated real-time services are desired.

For example, if you want to install only interrupt handlers, you only have to load the *rtai* module. If you also want to communicate with standard Linux processes using fifos, then you would then load the *rtai_fifos* module in addition to the *rtai* module. These modules can be dynamically loaded and unloaded – however it is necessary to pay attention to the order in which they are loaded and unloaded, as some modules require the services of other.

Altenatively, if the modules are installed in a directory known to modprobe (e.g /lib/modules/<xxx>/misc) and depmod is run, your real-time module along with all the RTAI modules it depends on may be loaded by a single 'modprobe' of your application module.

RTAI's basic services are provided by four modules, which allow hard real-time, fully preemptive scheduling based on a fixed priority scheme. These four key modules are:

- rtai - the basic RTAI framework, plus interrupt dispatching and timer support**.**
- rtai_sched – the real-time, pre-emptive, priority-based scheduler, chosen according to the hardware configuration.
- rtai_fifos – FIFOs and semaphores

- rtai_shm - shared memory (note that you can also use the ‘mbuff’ module for access to shared memory).

The advanced features of RTAI such as LXRT, Pthreads (POSIX 1003.1c) and Pqueues (POSIX 1003.1b), can be added with these modules:

- lxrt - LXRT
- rtai_pthread.o (Pthreads - loaded for POSIX 1003.1c support)
- rtai_pqueues.o (Pqueues – loaded for POSIX 1003.1b message queues support)
- rt_mem_mgr – dynamic memory management for real-time (note that this is most often simply built-in to the scheduler).

RTAI Schedulers

Although only one type of scheduler can be *insmod'ed* at any time, RTAI includes several different types – each uniquely suited to a specific combination of hardware and tasking requirements. It is generally not necessary for the user to manually install the proper scheduler because the installation process is usually able to determine the appropriate scheduler from the hardware configuration of the target machine. It then copies and links the appropriate scheduler so that it is called by the generic *rtai_sched* reference. However, in cases where the developer wants to investigate other RTAI schedulers, or when he is determining which scheduler should be installed onto the target platform, an understanding of each option is required.

The RTAI distribution includes three different priority based, pre-emptive real time schedulers: the Uni-Processor (UP) scheduler; the Multi Uni-Processor (MUP) scheduler; and the Symmetric Multi-Processor (SMP) scheduler, which each incorporate standard RTOS scheduling services like resume, yield, suspend, make periodic, wait until etc. The implementation and functional usage for each of these schedulers is described below.

- UP scheduler (located in the upscheduler directory)
- SMP scheduler (located in the smpscheduler directory)
- MUP scheduler (located in the mupscheduler directory)

UP scheduler (located in the upscheduler directory)

This scheduler is intended for uni-processor platforms where the timer is 8254-based and supports either one-shot or periodic scheduling but not both simultaneously. This should not be used on an SMP machine.

SMP scheduler (located in the smpscheduler directory)

This scheduler is intended for multi-processor machines but can support either the 8254 or APIC based timers and supports either single-shot or periodic scheduling but not both simultaneously. Tasks can run symmetrically on any or a cluster of CPUs, or be bound to a single CPU.

Depending on the hardware's architecture, either the 8254 or the local APIC timer schedulers are chosen for SMP operations. The chosen one will be built and then installed as the ‘generic’ *rtai_sched.o* module.

RTAI supports true Symmetric Multi Processing (SMP) architectures by providing dynamic task loading and IRQ management similar to Linux' SMP operations. RTAI contrasts sharply to other real-time Linux implementations, which do not support standard SMP load balancing techniques.

Under RTAI, by default all tasks are defined to run on any of the CPUs and are automatically moved between CPUs as the system's processing and load requirements change. However, to accommodate situations where manual task distribution is able to manage the task loading more efficiently than the automatic load distribution services, the developer also has the ability to assign individual tasks to any single CPU or to a CPU subset. Additionally, any specific real-time interrupt service can be assigned to any specific CPU, and because the ability to force an interrupt to any specific CPU is not related to the SMP scheduler, these two operations can be performed independently.

To assign individual tasks to any subset of the CPU pool – including just a single CPU, use:

- *rt_set_runnable_on_cpu*s or
- rt_set_runnable_on_cpuid

rt_set_runnable_on_cpus allows you to specify a set (pool) of CPUs (may be just one) to run the task on. Only one will be selected, however. rt_set_runnable_on_cpuid allows you to specify only one CPU, but note that if none of the chosen CPUs are available, the calls will select another one if they can.

To assign any real time interrupt service to a specific CPU, use:

- rt_assign_irq_to_cpu

and

- *rt_reset_irq_to_sym_mod*e.

Hence, a user can statically optimize their application in instances where it is thought to be better than using a symmetric load distribution.

n o t e : To determine whether there is an APIC available, type "cat /proc/cpuinfo" and search for "apic" in the flags field. If an APIC is available, then it is recommended to implement this scheduler. If it is not, then the 8254 scheduler should be inserted.

n o t e : These SMP schedulers can be used on hardware which is physically uni-processor but whose Linux kernel has been compiled for SMP configuration.

MUP scheduler (located in the mupscheduler directory)

The Multi-Uniprocessor scheduler is for multiprocessor platforms only and supports both single-shot and periodic scheduling simultaneously.

RTAI's Multi Uni-Processor (MUP) real-time scheduler implementation is suitable for those architectures, which include the Intel APIC functionality (i.e. Pentiums and equivalent). The MUPS can be effectively used on MP machines with just one CPU mounted on the motherboard.

The MUP is used in a multi-processor but non-true SMP environment where real-time tasks are bound to a single CPU at task initialization. The tasks can be moved to a different CPU by using the function *rt_set_runnable_on_cpu*s, but only to one specific CPU (i.e. not to a pool as can be done under the SMP schedulers). However, like the SMP schedulers, the MUP can use inter-CPU services related to semaphores, messages and mailboxes. The main advantage of the MUP scheduler comes from the ability to be able to use mixed timers simultaneously, i.e. periodic and one-shot, where periodic timers can be based on different periods.

note: In the future it is expected that the MUP will provide the ability to force critical real-time tasks onto the CPU cache on Pentium IIIs.

Readme (RTAI UP Scheduler readme file)

Here you'll find an implementation of an UP realtime scheduler to be interfaced to the RTAI module. Do not use it on an SMP machine.

Readme (RTAI SMP Scheduler readme file)

Here you'll find an implementation of an SMP realtime scheduler to be interfaced to the RTAI module. It can use either the 8254 or the local APIC timer. Be warned that the APIC based scheduler cannot be used for UP, unless you have the local APIC enabled, i.e. an SMP machine with just one CPU mounted on the motherboard.

It is a fully symmetric scheduler, where at task init all real time tasks default to using any available cpu. However you can chose either forcing a task to a single cpu or to let it use any subset of those available by calling the function "rt_set_runnable_on_cpus". That function set the task structure variable "runnable_on_cpus" with the bit map of the usable cpus, which is defaulted to "cpu_present_map", meaning that any available cpu can be used, at task initialization. Thus a user can statically optimize is application if he/she believes that can For the APIC timer based scheduler if you want to statically optimize the load distribution by binding tasks to specific cpus it can be usefull to use "rt_get_timer_cpu()" just after having started the timer, to know which cpu is using its local APIC timer to pace the scheduler. Note that for the oneshot case that will be the main timing cpu but not the only one. In fact which local APIC is shot depends on the task scheduling out, as that will determine the next shooting.

For the 8254 timer based scheduler a statically optimized load distribution could bind the 8254 interrupt to a specific cpu by using "rt_assign_irq_to_cpu" and "rt_reset_irq_to_sym_mode", and then assign tasks in appropriate way to any cpu or cpu cluster.

Actually there are two schedulers: the pessimistic one keeps the global lock throughout any scheduling, while the optimistic one releases the lock

immediately after the task switch.Significant differences in performance should be seen only if you are lucky to have more than 2 CPUs.

The scheduler you decide to adopt must be copied in rtaii_sched.c to be compiled by using "make". Then just do "make instapic" or "make inst8254" at your choice. Clearly if you have APIC enabled that is the best choice. The 8254 can be used also on truly UP.

Readme (RTAI MUPS Scheduler readme file)

Here you'll find an implementation of an MultiUniProcessor (MUP) realtime scheduler to be interfaced to the RTAI module. It is based on the local APIC timers. It can be profitably used on MP machines with just one CPU mounted on the motherboard.

The MUP scheduler derives its name from the fact that real time tasks MUST be bounded to a single CPU at the very task initialization. They can be afterward moved by using the function "rt_set_runnable_on_cpus". The MUP scheduler can however use any 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 possibly forcing critical tasks on the CPU cache on PIIIs, in the future. With dual SMP I cannot say that there is a difference in efficiency. MUP has been developped primarely for our not so fast, a few Khz, PWM actuators, BANG-BANG air jet thrusters, coupled to a periodic scheduler.

All the fuctions of UP and SMP schedulers are available in the MUP scheduler, and MUP specific functions can be used under UP-SMP. Some default action is implied in scheduler-specific features. The main difference can be seen for functions whose name ends with "_cpuid". Such functions imply the specification of a CPU number and came into play with the MUP scheduler whenever a cpuid had to be declared. Typical examples are: task init and time conversions, when time formats differ. Please note that there is a difference between "cpuid", i.e. the CPU number, and "cpu_map", i.e. 1 << cpuid. Thus if you use task init with a cpuid on UP-SMP schedulers you have it assigned to the only CPU available or mapped to the declared one, while if you just task init on MUP your task is assigned to the CPU loaded with less tasks, and so on. Have a look at rtai_sched.h to see the new functions and at the schedulers to verify the default actions.

Be carefull in relation to time conversion under MUP with hetereogeneous timers otherwise you'll put on the scheduler blames that are due only to your misunderstanding on how it works.

Clearly no problem arises if the same kind of timers are used on all CPUs, and with the same period if they are periodic. However the advantage of the MUP scheduler is really the possibility of having a periodic and a oneshot timer, or two periodic timers with different periods, simultaneously, and you must use it for that case. Only exsercize due care while timing at initialization.

For this reason some test examples have be regrouped under the directory mups_examples. It contains only meaningfull examples with more than one task. You can run MUP specific examples with UP-SMP schedulers as well as all the UP-SMP examples can be run with the MUP scheduler. Using "cat /proc/rtai/*" can help in seeing what happened.

The timing relies on the RTAI support functions:

```
- void rt_request_apic_timers(void (*handler)(void),
  struct apic_timer_setup_data *apic_timer_data)
```

and

```
- void rt_free_apic_timers(void)
```

The "struct apic_timer_setup_data { int mode, count; };" allows you to define the mode and count to be used for each timer as:

```
- mode: 0 for a oneshot timing, 1 for a periodic
  timing;

- count: is the period in nanosec you want on that
  timer.
```

It is in nanosec to ease your programming in relation to what said above. It is used only for a periodic timer as for the oneshot case the first shot defaults to the Linux tick. You should care of that in starting periodic task not in advance of that time. The start of the timing should be reasonably sinchronized internally. However you must not call the above functions directly but use the usual start_rt_timer which defaults to the same timer on each LOCAL APIC or start_rt_apic_timers taht allows you to use struct apic_timer_setup_data directly. Note that the latter uses nanosecs, and not internal counts, for the apic count. So you do not have to care for the conversion. If start_rt_apic_timers is used with UP-SMP the single timer is set as periodic only if all the requested APIC timers are periodic with the same period, oneshot otherwise.

It is not the only way to do the all stuff but is the one that suits our needs right now. Suggestions and comment to improve it are welcomed.

To use it you have just to do "make" and "make install."

chapter 2

Development Fundamentals

As there are many texts that cover general Linux topics, it is not our goal to instruct the reader on the fundamentals of code development under standard Linux – although we will provide examples as appropriate. Instead we assume that the reader is familiar with this and with standard terms and techniques required for the design of real-time system architectures. Under these guidelines this chapter will attempt to provide information specific to general issues associated with development for RTAI.

Chapter Summary:

- **Module Basics** – insmod, rmmod, lsmod, init_module(), cleanup_module()
- **Building a Kernel Module** – Because real-time tasks are implemented as kernel modules, the correct compiler and linker flags must be used.
- **Passing data to a kernel module** - Using global variables and insmod.
- **IDEs** – The developer is free to use the development tool chain to which he/she has become accustomed.

Kernel Module Basics

The operating system and application code that is executed under Real-Time Linux are loaded as kernel loadable modules, so it is important that the mechanics of kernel loadable modules are properly understood. Below, we purposefully avoid the details associated with RTAI, but instead discuss basic kernel module operations

A Linux kernel module is a portion of code which when loaded, effectively becomes an integral part of the Linux kernel, having access to the same

memory space occupied by the monolithic portion of the kernel. Kernel modules are not a concept new to real-time Linux programming, but are commonly used by Linux to provide load-on-demand device drivers – e.g. network card driver. Two primary methods of loading and unloading kernel modules are available under Linux: "manual" and "on-demand" loading.

Manual loading is cumbersome, as the operations must be carried out as the super user and remembering to load/unload modules is tedious and prone to error. Manual loading requires that the kernel module be inserted and removed using commands provided for these operations.

The command *insmod* loads modules into the kernel, and *rmmod* removes modules from the kernel. *lsmod* lists the kernel modules that are currently loaded, the number of memory pages occupied by the module, and the processes using it. (Note: see the section on /proc regarding the availability of additional module system usage information)

Those requiring detailed information on *insmod*, *rmmod*, or *lsmod* should consult either the system man pages, or any of the general Linux books that are now available.

On-demand loading enables automatic loading of kernel modules according to demand. Its installation requires activation of the option Kernel module loader (CONFIG_KMOD) when the kernel is configured. This option is valid for the 2.2.x series of kernels and is a replacement for kerneld which was supported in the 2.0.x kernels.

A Linux kernel module must have two entry points for loading and unloading: *init_module()* for loading and *cleanup_module()* for unloading. *init_module()* is invoked when the module is loaded into kernel memory, and registers the module including its functions and exported symbols with the kernel. *cleanup_module()* is invoked just before the module is unloaded, and removes the module functions from the kernel.

Building a Kernel Module

Here we will describe the compiler and linker steps that must be taken and the flags included to create a generic Linux kernel module. We also discuss the procedure necessary to make kernel module global symbols visible to the rest of the kernel.

Those who are familiar with gcc and standard Linux code development should be relatively comfortable with this procedure since the only fundamental difference between kernel module and standard development is in the selection of specific compiler flags. However, for those who are less familiar with this

type of development, we recommend Alessandro Rubini's "Linux Device Drivers" book (O'Reilly & Associates, 1st Edition February 1998, 1-56592-292-1, Order Number: 2921, 442 pages, $29.95), which covers the topic of kernel modules and device drivers very thoroughly. A sample driver called *skull* from this book is a good example that illustrates the techniques for loading and unloading kernel modules, and the issues that must be addressed. The *Makefile* used to build the *skull* driver is also a good template for building kernel modules.

The steps for building a kernel module are:

1. Edit source code (filename.c)
2. Compile with the following (minimal) flags:

   gcc -g -D__KERNEL__ -DMODULE –O2 -Wall –I<Include file paths>

   –c filename.c ***-o filename.o***

   Usually, you will use 'make' to control module building. Make is controlled using a 'makefile' in which you place all the rules necessary for building your kernel modules. The use of makefiles allows you to use macros, which are defined once and then used throughout the makefile in different rules where the macro gets expanded to it'd original definition.

   An example of makefile macros for the above:

   INCLUDEDIRS= /usr/src/linux/include /usr/src/rtai/include

   CFLAGS = -g -D__KERNEL__ -DMODULE –O2 -Wall -I$(INCLUDEDIRS)

   Then the relevant rule (in the makefile) becomes:

   gcc $(CFLAGS) -c filename. c -o filename. o

   Make is a very large and esoteric subject in itself, we would direct the reader to read any of the good books on the subject including GNU Make by Richard M. Stallman and Roland McGrath ISBN 1-882114-79-5.

   Notes:

   - The –g option adds in the debug symbols for use with GDB
   - The -D__*KERNEL*__ flag, (note the double underscores!) is used by the preprocessor to select certain parts of kernel headers.

- The –D*MODULE* symbol must be defined for a kernel loadable module, and should be defined before including <*linux/module.h*>
- the *–O2* flag must be specified, as many functions are declared as inline in the header files, and gcc doesn't expand inlines, unless optimization is enabled.
- -*Wall* is recommended, as the elimination of all compiler warnings will help prevent unexpected errors later on.
- -I<Include file paths> specifies the directories in which the included header files can be found
- The option -c *filename.c* tells gcc to stop after generating the object file (doesn't go on to the link phase).
- -o *filename.o* tells the compiler to create an object file of name *filename.*

3. If and only if your module is composed of more than one object file, you need to perform partial linking of the component object files. Note that only one object file (conventionally the one with init_module and cleanup module) should define the module version with -DMODULE.

   ld -r -o file_klm.o file1.o file2. o

You now have a kernel loadable module called file_klm.o, which can be loaded and unloaded using *insmod* and *rmmod* as described above.

Programming tips:

1. 1. In cases where more than one source file is used, if a variable is referenced in one file yet it is defined in another, you must add the *extern* keyword in the source code (filename.c), for example; if you read a variable in one module and write it in another. Note you'll need to mark external storage (shared memory etc.) as *extern* if a reference is not complete.
   e.g.: *extern int var1;*
2. When you've finished debugging the task, either recompile with debugging disabled (i.e. no –g flag) or use the *strip* utility to discard all non-global symbols without the need to recompile.
   e.g.: *strip –g filename.o* ()
3. Kernel version dependency is a subject that needs to be considered when writing kernel loadable modules. The file *version.h* is included by *module.h* which defines the variable *kernel_version* unless

__NO_VERSION__ is defined. The *__NO_VERSION__* symbol can be used in cases where <*linux/module.h*> will be included in several source files that will be linked to form a single module. This prevents automatic declaration of the variable *kernel_version* in source files where this is not wanted. *ld -r* would complain about multiple definition of *kernel_versio*n.

You can try to load a module against a different kernel version (which does not meet the kernel dependency) by specifying the –f (force) switch to *insmod*, it is not reliable and often causes more problems than the small inconvenience of recompiling the real-time task or of running it as a user space process using the facilities of LXRT.

Exporting Module Symbols

The Linux kernel module interface defines a number of macros to control global visibility of module symbols by the rest of the kernel. The more important macros are described below. These relate to the 2.2.x Linux kernels.

EXPORT_SYMTAB;

When a module requires to export one or more symbols, it must define this macro before including the header file: <*linux/module.h*>

EXPORT_NO_SYMBOLS;

This macro prevents the export of any symbols from the module. It can appear anywhere in the module, as it is an assembler directive. In instances where compatibility is needed with Linux 2.0.x kernels, it should be defined within the *init_module()* function.

EXPORT_SYMBOL(symbol_name);

This macro causes the symbol *symbol_name* to be exported. It must be used outside any function.

EXPORT_SYMBOL_NOVERS(name);

This macro can be used instead of the macro *EXPORT_SYMBOL()* in instances where version information is not required, which can be useful to avoid unnecessary compilation. An example of how to declare and export symbols in a kernel module is shown below:

```
#define EXPORT_SYMTAB;
```

```
#include <linux/module.h>
...
...
// Symbols exported to other kernel modules.
EXPORT_SYMBOL(var1);
EXPORT_SYMBOL(var2);
EXPORT_SYMBOL(func1);
...
...
```

Passing Parameters to Kernel Modules

Because Linux allows data to be passed (using the *insmod* command) to the module when it is loaded, all kernel modules, including real-time tasks can be easily configured upon loading. This is achieved by declaring such variables as global within the module, and then using the MODULE_PARM(variable, type) macro.

When the insmod command is executed, insmod can assign a value of the correct type according to the declaration in MODULE_PARM.

e. g.:

```
int module_number   = 0;
char *report_string = NULL;
MODULE_PARM(module_number, "i")        /* accept an integer *
MODULE_PARM(report_string, "s")          /* string value */

int init_module( void)
{

    printk("Module number %d reports %s\ n", module_number, report_string);
.
.
    return 0;

}
```

then,

```
insmod my_kernel_module module_number=10 report_string="Hi Mum!"
```

will produce:

```
Module number 10 reports Hi Mum!
```

Note in addition you may specify the format parameter as 'h' for 16 bit values. Also you may give a range value such as "1-2i", which say accept 1 to 2 integers (useful for array initialisation).

Integrated Development Environments (IDEs)

RTAI is completely transparent to Linux, and it is implemented using the C programming language. However, many other programming languages are also suitable for developing real-time kernel modules so long as they support an interface to programs written in C. For example: Ada, C++, Fortran, etc. As a consequence any IDE that supports the Gnu compilers and/or allows the user to select a compiler/linker tool-chain could be helpful for development of real-time Linux tasks (i.e. kernel modules).

Currently there are many IDEs available for Linux including both open source and commercial offerings:

- Metrowerks Code Warrior — http://www.metrowerks.com
- K-Develop — http://www.kdevelop.org
- Code Crusader — http://www.newplanetsoftware.com/jcc/
- Cygnus Code Fusion — http://www.cygnus.com/codefusion/
- Code Forge — http://www.codeforge.com

chapter 3

RT Task Programming Basics

RTAI Real-time Task Programming Summary

Except for the requirement to use either the RTAI native API or the POSIX 1003.1c API, the only significant difference between development of real-time tasks and development of standard user space applications (ignoring general issues associated with real-time application engineering) is in their compilation and deployment as kernel modules.

Thus, a real-time task running as a kernel module under RTAI consists of three fundamental code sections.

1. *init_module ( )* function
2. Real-time task specific code (consisting of either the RTAI or POSIX APIs)
3. *cleanup_module ( )* function

init_module ( )

A kernel module must always contain an init_module function. This function is invoked by insmod whenever the module is loaded. The purpose of this function is to prepare for later invocation of the module's functions. This is a useful place to allocate required system resources, declare and start tasks etc.

cleanup_module ( )

The second required module entry point is the cleanup_module function. This is invoked as the module is removed via rmmod. It's job is to inform the kernel that the module has been removed so that none of it's functions are called anymore. This is a convenient place to release all of the system resources allocated during the lifetime of the module, stop and delete tasks etc.

Real-time Task Specific Code

The task-specific code implements the runtime part of the application (at the very least). It uses the RTAI API, data structures, and services to perform the task(s), and associated communications with Linux. This part is optional as it's possible (though rather restrictive) to create applications using just init_module and cleanup-module.

Here is a simple example:

```
#define MODULE
#include <linux/module.h>
#include <linux/cons.h>

static int output=1;

int init_module(void) {
       printk("Output= %d\n",output);
       return 0;
}
void cleanup_module(void){
       printk("Hasta lluego, baby! \n");
}
```

Let's now take a look at an example of a real-time program using RTAI. This is the RTAI 'preempt' example. It is a simple test to verify that a fast high priority task preempts a longer lasting, lower priority one. They communicate with a User Space application via a FIFO and it is this User Space program, 'check' that allows us to visualize the interaction between the tasks.

Here's the code:

```
/*
FILE: rt_process.c

COPYRIGHT (C) 1999  Paolo Mantegazza (mantegazza@aero.polimi.it)

This library is free software; you can redistribute it and/or modify it
  under the terms of the GNU Lesser General Public License as published
  by the Free Software Foundation; either version 2 of the License, or
  (at your option) any later version.
```

```
This library is distributed in the hope that it will be useful, but
  WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
  Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License
  along with this library; if not, write to the Free Software
  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307
  USA.
*/

#include <linux/module.h>

#include <asm/io.h>

#include <rtai.h>
#include <rtai_sched.h>
#include <rtai_fifos.h>

#define TIMERTICKS 500000

#define CMDF0 0

#define ONE_SHOT

static RT_TASK Slow_Task;
static RT_TASK Fast_Task;

static int cpu_used[NR_RT_CPUS];

static void Slow_Thread(int t)
{
        static struct {     char task, susres;
                       unsigned long flags;
                       RTIME time;} msg = {'S',};
        while (1) {
              cpu_used[hard_cpu_id()]++;
            msg.time = rt_get_cpu_time_ns();
            msg.susres = 'r';
              rt_global_save_flags(&msg.flags);
            rtf_put(CMDF0, &msg, sizeof(msg));

            rt_busy_sleep(11*TIMERTICKS);
```

```
            msg.time = rt_get_cpu_time_ns();
            msg.susres = 's';
              rt_global_save_flags(&msg.flags);
            rtf_put(CMDF0, &msg, sizeof(msg));

           rt_task_wait_period();
        }
}

static void Fast_Thread(int t)
{
        static struct {     char task, susres;
                       unsigned long flags;
                       RTIME time;} msg = {'F',};

        while (1) {
              cpu_used[hard_cpu_id()]++;
            msg.time = rt_get_time_ns();
            msg.susres = 'r';
              rt_global_sti();
              rt_global_save_flags(&msg.flags);
            rtf_put(CMDF0, &msg, sizeof(msg));

            rt_busy_sleep(2*TIMERTICKS);

            msg.time = rt_get_time_ns();
            msg.susres = 's';
              rt_global_save_flags(&msg.flags);
            rtf_put(CMDF0, &msg, sizeof(msg));

               rt_task_wait_period();
        }
}

int init_module(void)
{
      RTIME tick_period;
      RTIME now;
         rtf_create_using_bh(CMDF0, 20000, 0);
         rt_task_init(&Fast_Task, Fast_Thread, 0, 2000, 0, 0, 0);
         rt_task_init(&Slow_Task, Slow_Thread, 0, 2000, 1, 0, 0);

#ifdef ONE_SHOT
         rt_set_oneshot_mode();
```

```
#endif
        tick_period = 4*start_rt_timer(nano2count(TIMERTICKS));
        now = rt_get_time();
        rt_task_make_periodic(&Fast_Task, now + tick_period,
                                          tick_period);
        rt_task_make_periodic(&Slow_Task, now + tick_period,
                                         6*tick_period);

        return 0;
}

void cleanup_module(void)
{
        int cpuid;
        stop_rt_timer();
        rt_busy_sleep(10000000);
        rtf_destroy(CMDF0);
        rt_task_delete(&Slow_Task);
        rt_task_delete(&Fast_Task);
        printk("\n\nCPU USE SUMMARY\n");
      for (cpuid = 0; cpuid < NR_RT_CPUS; cpuid++) {
                printk("# %d -> %d\n", cpuid, cpu_used[cpuid]);
}
        printk("END OF CPU USE SUMMARY\n\n");
}

/*
FILE: check.c

COPYRIGHT (C) 1999  Paolo Mantegazza (mantegazza@aero.polimi.it)

This library is free software; you can redistribute it and/or modify it
  under the terms of the GNU Lesser General Public License as published
  by the Free Software Foundation; either version 2 of the License, or
  (at your option) any later version.

This library is distributed in the hope that it will be useful, but
  WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
  Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License
  along with this library; if not, write to the Free Software
```

```
  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307
  USA.
*/

#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <sched.h>
#include <signal.h>

static int end;

static void endme(int dummy) { end = 1; }

int main(void)
{
        int cmd0, count = 0, nextcount = 0;
        struct sched_param mysched;
        char wakeup;
        struct {     char task, susres;
                  int flags;
                       long long time;} msg = {'S',};
      signal (SIGINT, endme);

      mysched.sched_priority = 99;

       if( sched_setscheduler( 0, SCHED_FIFO, &mysched ) == -1 ) {
         puts(" ERROR IN SETTING THE SCHEDULER UP");
         perror( "errno" );
         exit( 0 );
 }

      if ((cmd0 = open("/dev/rtf0", O_RDONLY)) < 0) {
                fprintf(stderr, "Error opening /dev/rtf0\n");
           exit(1);
}
     while(!end) {
           read(cmd0, &msg, sizeof(msg));
            printf("> %c %c %x %lld\n", msg.task, msg.susres, msg.flags
                  & 0x201, msg.time/1000000);
     }
}
```

This application may seem very complicated but it can be broken down into its basic elements, dealing with the Real-Time side first:

The following are the header files that we need for the Linux side of the program. For example, the module.h file is needed so that we can create a kernel module.

```
#include <linux/module.h>
#include <asm/io.h>
```

The following are the header files required in order to make use of RTAI. They give us access to the functions and data structures within RTAI.

```
#include <rtai.h>
#include <rtai_sched.h>
#include <rtai_fifos.h>
```

Now we get to some global definitions and data declarations. TIMERTICKS defines the timer's tick rate in nanoseconds, in this case equating to 0.5 ms, which means that the timer will tick twice per millisecond. CMDFO defines the FIFO number to be used for communications. ONE_SHOT is defined so that some code in *init_module* will set the timer into one-shot mode. One-shot mode allows variable timing where each task can be timed arbitrarily. The default mode for the RTAI timer is periodic mode where tasks are timed relative to a fixed frequency.

```
#define TIMERTICKS 500000
#define CMDF0 0
#define ONE_SHOT
```

Here, task data structures are declared for each of the two real-time tasks to be created.

```
static RT_TASK Slow_Task;
static RT_TASK Fast_Task;
```

This next array is declared to handle running on an SMP machine. Each time either task runs, they increment the value associated with the particular CPU on which the task is running. The *cleanup_module* uses this data in its final report, reporting how many times a task was run on each CPU in the system. Note that on a uni-processor platform this is of rather limited use.

```
static int cpu_used[NR_RT_CPUS];
```

Now, before dealing with the tasks themselves, let's take a look at the module control functions: *init_module* and *cleanup_module*.

Firstly, *init_module*, is where the module starts. It declares two ‘time’ variables: *now* is used as a placeholder for the current time and *tick_period* is used to hold the base timer tick period, both being used when the tasks are enabled.

```
int init_module(void)
{
      RTIME tick_period;
      RTIME now;
```

Next, a Real-Time FIFO is created, using the number declared above (CMDFO) and size 20000 bytes.

```
rtf_create(CMDF0, 20000);
```

Now we create the two application tasks. Each one has a pointer to a pre-declared task structure e.g. Fast_Task), a pointer to the function to be used as the runtime portion of the task, e.g. Fast_Thread, an integer data value to be passed to the task as it starts, a task stack size, a task priority, a flag to say whether or not the tasks uses floating point calculations (does it need the FPU or not?) and a pointer to a signal handler function.

```
rt_task_init(&Fast_Task, Fast_Thread, 0, 2000, 0, 0,
  0);
```

Here’s the declaration of the Slow Task, broken down:

```
rt_task_init(  &Slow_Task, // the task structure
               Slow_Thread, // the task function
               0, // initial data value
                2000, // stack size
               1, // priority
               0, // task does not use the FPU
               0// task has no signal handler
 );
```

This is the point where the timer default is changed from periodic mode to one-shot mode as determined by the #define at the top.

```
#ifdef ONE_SHOT
      rt_set_oneshot_mode();
#endif
```

Now we start the timer. In one-shot mode (the mode we're currently setting-up) the parameter passed to the call *start_rt_timer* is ignored. However, it's left here in case this example is ever re-complied for periodic mode. In periodic mode the TIMERTICKS values declared above is converted from nanoseconds to a period in internal count units. The return value, *tick_period*, is the internal period value that the timer is actually set to, in either mode.

```
tick_period =
  4*start_rt_timer(nano2count(TIMERTICKS));
```

Finally, we start the two application tasks running, note that starting a task is a two-stage process. The current time is read, and both tasks started at the same time but with different periods, the slow task runs six times slower than the fast task. This factor will become apparent when we look at the output produced by running this example.

```
now = rt_get_time();
rt_task_make_periodic(&Fast_Task, now + tick_period,
                              tick_period);
rt_task_make_periodic(&Slow_Task, now + tick_period,
                            6*tick_period);


return 0;
}
```

The cleanup_module function is responsible for tidying-up when the module is removed. Essentially, all system resources allocated by the module need to be de-allocated, all tasks and FIFOs destroyed, the timer stopped etc, etc.

```
void cleanup_module(void)
{
      int cpuid;
```

Stop the timer and wait, without yielding the processor, for a number of nanoseconds, in this case 10000000, which equates to 10ms. Then destroy the FIFO created in init_module and used by the application tasks.

```
stop_rt_timer();
rt_busy_sleep(10000000);
rtf_destroy(CMDF0);
```

Finally, delete the two application tasks and print out a summary of the number of times a task (any task) ran on each processor in the system.

```
    rt_task_delete(&Slow_Task);
    rt_task_delete(&Fast_Task);
    printk("\n\nCPU USE SUMMARY\n");
    for (cpuid = 0; cpuid < NR_RT_CPUS; cpuid++) {
       printk("# %d -> %d\n", cpuid, cpu_used[cpuid]);
    }
    printk("END OF CPU USE SUMMARY\n\n");
}
```

Now, let us return to the two application tasks, Fast_Thread and Slow_Thread. Note that each declares an input parameter but that in this case it's not used.

```
static void Slow_Thread(int t)
{
```

Firstly, a data structure is declared and initialized, to hold reporting data for the task. This data structure declares the format of the messages to be on the FIFO. The User-Space program , *check.c*, will receive and decode these messages for reporting on the console. The task is identified by a single character, 'S' or 'F'. The requirement for the rest of these parameters will become apparent as we dissect the rest of this function.

```
static struct {char task, susres;
unsigned long flags;
RTIME time;} msg = {'S',};
```

Loop forever (until the task is destroyed), and on each pass of the loop increment the CPU usage flag for the processor on which this task runs.

```
        while (1) {
      cpu_used[hard_cpu_id()]++;
```

Now, get the number of nanoseconds since the timer was started and store it in the message structure. Also, set the message identifier to 'r' and save the current state of the CPU interrupt flag (IF) and the global lock flag in the message structure.

```
msg.time = rt_get_cpu_time_ns();
msg.susres = 'r';
rt_global_save_flags(&msg.flags);
```

Having composed a message, write it to the FIFO and wait, without yielding the processor for 5.5 ms

```
rtf_put(CMDF0, &msg, sizeof(msg));
rt_busy_sleep(11*TIMERTICKS);
```

Now, get the elapsed time again, change the message identifier to ’s’, save the flags again and put the new message onto the FIFO.

```
msg.time = rt_get_cpu_time_ns();
msg.susres = 's';
rt_global_save_flags(&msg.flags);
rtf_put(CMDF0, &msg, sizeof(msg));
```

Finally, suspend the task, yielding the processor this time, until it is next scheduled to run.

```
       rt_task_wait_period();
        }
}
```

The Fast_Thread, does the same as the Slow_Thread with a couple of exceptions. Remember that it was started with a different priority (higher) and a different execution period (six times faster than the slow thread). Note also that the busy sleep between writes to the FIFO is much shorter.

```
static void Fast_Thread(int t)
{
        static struct {   char task, susres;
                    unsigned long flags;
                         RTIME time;} msg = {'F',};
      while (1) {
             cpu_used[hard_cpu_id()]++;
          msg.time = rt_get_time_ns();
         msg.susres = 'r';
             rt_global_sti();
               rt_global_save_flags(&msg.flags);
            rtf_put(CMDF0, &msg, sizeof(msg));

         rt_busy_sleep(2*TIMERTICKS);
```

```
            msg.time = rt_get_time_ns();
            msg.susres = 's';
                  rt_global_save_flags(&msg.flags);
               rtf_put(CMDF0, &msg, sizeof(msg));

               rt_task_wait_period();
            }
}
```

Now it's time to look at the User Space program that reads from the FIFO and writes reports to the console, allowing you to visualize what's going on.

First, it must include the relevant Linux headers, declare a static variable 'end' (automatically initialized to zero) to be used for loop control, and a function 'endme' to be used to set end to '1' and thus force loop exit.

```
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <sched.h>
#include <signal.h>

static int end;

static void endme(int dummy) { end = 1; }
```

Here is the 'main' User-Space program. Note that it declares a number of local variables and a message structure *identical* to the one declared by the real-time application tasks, Slow_Thread and Fast_Thread, in the file *rt_process.c.*

```
int main(void)
{
   int cmd0, count = 0, nextcount = 0;
   struct sched_param mysched;
   char wakeup;
   struct {  char task, susres;
           int flags;
```

```
long long time;} msg = {'S',};
```

This line, attaches the function ‘endme’ to the signal SIGINT (which will be generated by Ctrl-C). When the example is run, the user is prompted to hit Ctrl-C in order to terminate the test.

```
signal (SIGINT, endme);
```

Next, the process resets its own scheduling policy and priority in a call to the Linux scheduler. The Linux scheduler defaults to SCHED_OTHER , which is a time-sharing policy most commonly used by linux processes. SCHED_FIFO used to gain a greater level of control over the way in which the process is scheduled. SCHED_FIFO is used with static priorities greater than 0 (in this case the maximum allowed, 99). Using this policy allows the process to preempt any other process scheduled using SCHED_OTHER and any process with a lower priority. For more information, consult the man page for ‘sched_setscheduler’. If this call fails, the process exits after writing a warning message to the console.

```
mysched.sched_priority = 99;
if( sched_setscheduler( 0, SCHED_FIFO, &mysched ) == -1 ) {
     puts(" ERROR IN SETTING THE SCHEDULER UP");
     perror( "errno" );
     exit( 0 );
 }
```

Here, the process opens the real-time FIFO ‘/dev/rtf0’. Note that the minor number used ‘0’ is the same number as the #define CMDFO in the real-time module. This is essential if the real-time applications are to communicate with this program.

```
if ((cmd0 = open("/dev/rtf0", O_RDONLY)) < 0) {
    fprintf(stderr, "Error opening /dev/rtf0\n");
    exit(1);
}
```

Finally, we reach the main loop of the program. Until ‘end’ is set to ‘1’ (or anything other than zero) this loop will continue. The user is prompted by the ‘run’ script accompanying this example in the RTAI distribution, to hit ‘Ctrl-C’ to terminate the program and as we’ve already seen, the signal that generates (SIGINT) has been linked to the ‘endme’ function that will set ‘end’ to ‘1’. The program reads the message at the head of the FIFO and prints out a formatted message onto the console screen, containing all the information passed to it by the real-time application tasks.

```
   while(!end) {
      read(cmd0, &msg, sizeof(msg));
      printf("> %c %c %x %lld\n", msg.task,
  msg.susres, msg.flags
& 0x201, msg.time/1000000);
   }
}
```

So, what do you see when the test runs? Something like this:

> F r 201 3929933

> F s 201 3929934

> F r 201 3929935

> F s 201 3929936

> F r 201 3929937

> F s 201 3929938

> S r 201 3929938

> F r 201 3929939

> F s 201 3929940

> F r 201 3929941

> F s 201 3929942

> F r 201 3929943

> F s 201 3929944

> S s 201 3929944

What this shows is that the Fast Task runs six time faster than the slow one. Remember the factor applied to their period values when the two tasks were started? What this also shows, is that the Fast Task becomes eligible to run whilst the Slow Task is ‘busy sleeping’ between writes to the FIFO and preempts it because it (the Fast Task) is now the highest-priority task eligible to run in the system.

Hopefully, all of this has given you a foundation for writing real-time programs using RTAI. Here's a quick summary of the basics behind real-time task development:

RTAI Real-Time Task Programming Summary

1. Create the *init_module* function

   Note: Usually, *init_module*, starts the timer, creates and starts the tasks and creates any other system resources you might need.

2. Create a real-time task

3. Create a function for the real-time task to run

4. Create the *cleanup_module* function.

   Note: Usually, the *cleanup_module* function deletes the tasks, deletes the corresponding system resources, stops the timer etc.

Note that you don't have to do all of your initialization in *init_module*, nor all your system cleanup in *cleanup_module*, but you have to have them in a kernel module and they provide useful placeholders for these operations.

Inter-Process Communications (IPCs)

The term Inter-Process Communication (IPC) describes different ways of message passing between active processes or tasks, and encompasses numerous forms of data transfer synchronization.

Linux provides standard System V IPC in the form of shared memory, FIFOs, semaphores, mutexes, conditional variables, and pipes that can be used by standard user processes to transfer and share data.

Although these Linux IPC mechanisms are not available to real-time tasks, RTAI provides an additional set of IPC mechanisms that can be used to transfer, and/or share, data between tasks and processes in both the real-time and Linux user space domains.

These mechanisms include:

Native to RTAI:

Real-time fifos: A FIFO (First In First Out) is a read/write buffer used to asynchronously transfer data between real-time Linux tasks and Linux processes, where it is opened for write by one process and for read by another.

Shared memory: Provides a means to transfer data between real-time and user space tasks, in which a portion of physical memory is set aside for sharing between them.

Mailboxes: Provide the ability to transfer data of user-defined sizes between Linux and RTAI. It is assumed that a message format protocol will be imposed by the application level software.

Semaphores: Are used to achieve synchronization between tasks either with regard to access to shared resources or as a simple binary, message-passing system.

RPCs (Remote Procedure Calls): Are similar in operation to QNX-style messages available to real time tasks. These RPCs transfer either an unsigned integer or a pointer to the destination task(s).

IPCs Provided by the POSIX modules:

Mutexes: A mutex variable acts as a mutually exclusive lock, allowing threads to control access to data. The threads agree that only one thread at a time can hold the lock and access the data it protects.

Conditional variables: A condition variable provides a way of naming an event in which threads have a general interest. An event can be something as simple as a counter reaching a particular value or a flag being set or cleared; it may be something more complex, involving a specific coincidence of multiple events. Threads are interested in these events because such events signify that some condition has been met that allows them to proceed with some particular phase of their execution. The pthreads library provides ways for threads both to express their interest in a condition and to signal that an awaited condition has been met.

Message queues: POSIX message queues provide a general and abstract form of communication between real-time tasks. The queues allow messages of different sizes to be placed on them and handle a range of message priorities.

All of the above mechanisms are described in more detail in the appropriate other parts of this document.

FIFOS

Overview

A FIFO (First In First Out) is a uni-directional read/ write buffer used to asynchronously transfer data between running tasks or processes. The size of a FIFO generally refers to the number of fixed-width data items that can be stored in a full FIFO.

In other words, when data is written to the FIFO it is placed in the buffer in multiples of that width. For example, if a FIFO is 8-bits wide, a single 32-bit data word will be placed in the buffer as four, separate bytes. When the data is read from the FIFO, these four bytes will be taken from the buffer in the order in which they were written but the FIFO itself has no knowledge that they represent a single 32-bit data word, that protocol rendering is the responsibility of the user application. The size of a FIFO generally refers to the number of fixed-width data items that can be stored in a full FIFO.

RT-FIFOs allow communications between real-time Linux tasks and Linux processes but they can also be used for communication between RT Tasks and for communication between Linux processes if so desired. An RT-FIFO may be opened on either side of the User space–Kernel Space boundary. One side opens the FIFO for writing and the other for reading. The RT-FIFO allows the *write* process to add data to the buffer without having to wait for the *read* process to be ready until the FIFO becomes full.

RTAI supports two RT_FIFO implementations, named ‘oldfifos’ and ‘newfifos’. Oldfifos are based on the original NMT-RTL FIFOs. Newfifos are based on completely new code but maintain full compatibility with the basic services provided by its original NMT-RTL counterpart while adding some additional features. The remainder of this document describes the Newfifos RTAI implementation that will become the de-facto RTAI standard in future releases.

Although the newfifo API appears to be similar to the earlier NMT-RTL FIFOs, the new implementation is based on the RTAI mailboxes concept and is symmetrically usable from both kernel modules and Linux processes. Apart from the file-style API functions to be used in Linux processes, the only notable difference is that FIFOs implemented on the module side always have only non-blocking read/write access.

Although fifos are strictly no longer needed in RTAI, (due to the services of LXRT) they are kept for both compatibility reasons and, since they do not require any scheduler to be installed, they are very useful tools for communicating with interrupt handlers. In this sense this new implementation

of fifos acts as a kind of universal form of device driver, because once an interrupt handler is installed, one can use fifo services to do all the rest.

RT FIFOs exhibit the following characteristics:

- RT-FIFOS may be created from either user or kernel space
- RT-FIFOs may be ‘named’ and referenced by that name
- RT-FIFOs may be re-sized and reset after their creation.
- FIFOs queue data, so no protocol is required to prevent data overwrites. Applications must, however, deal with FIFO full/empty conditions
- Boundaries between successive data writes are not maintained. Applications must detect boundaries between data, particularly if data is of varying size.
- RT FIFOs appear as devices */dev/rtf0..63* in the file-system. There is no limit to the number of RT FIFOs an application can use, or to the size of data that can be written to an RT FIFO, other than practical memory limits
- RT-FIFOs support the /proc file-system interface.
- RT-FIFOs support semaphores for synchronization.
- RT-FIFOs support multiple readers and writers, and timed reads and writes.
- RT-FIFOs support asynchronous signals for ‘event’ notification (such as the arrival of data on a previously empty FIFO)

Implementation

All of the FIFO and semaphore functionality is implemented by a single kernel module.

From Linux user space, device nodes must be created for the fifos before you can make use of the RT-FIFO driver. This needs only to be done once. When installing RTAI, the first 5 FIFO nodes can be made by typing:

```
make fifo_devs
```

as root from within the RTAI directory. If you require more FIFO device nodes (R2D2 uses nodes 63 and 63) you can run the following command as root in the /dev directory:

```
perl -e 'foreach $i (0..63) { `mknod rtf$i c 150 $i`
  }'
```

This will create device nodes 0 thru 63 for major character device 150 (which has been registerred for RT-FIFOS).

From a user space application, the FIFO is opened before it can be used as with any regular character device, for example (omitting error checking):

```
Char mesg[ ] = "Data to be put on the FIFO";
fifo_id = open("/dev/rtf1", O_WRONLY);
written = write(fifo_id, &mesg, sizeof(mesg));
```

fifo_id is the file descriptor returned by the open system call, this is used in all other accesses to it.

```
eg: rtf_create(1, 1000);
```

which creates an RT-FIFO of initial size 1000 bytes for device minor number 1. At this point, the kernel may read or write (but not both) to the FIFO using the rtf_get/rtf_put real-time API. Also at this point, the FIFO is created and available for opening by user space applications as previously described.

Insert the Module

Insmod *rtai_fifos*

API

FIFO creation

RT FIFOS **can be easily created from both user and kernel space using the same functions.** If a FIFO that that has not been previously created is opened, it is automatically created with a default 1K buffer size. Any subsequent kernel space recreation of the FIFO resizes it without any loss of data.

To create an RT FIFO with a desired buffer size, use:

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

To resize an RT-FIFO from user space, use:

- `rtf_resize(int fd, int new_size);`

From Linux user space, RT-FIFOs are created with:

```
mknod /dev/rtfX c Y X
```

where:

X is the minor device number, from 0 to 63
C implies that the FIFOs are character devices
Y is the device major number

For example:

```
mknod /dev/rtf1 c 63 1
```

which creates an RT_FIFO device with minor number 1.

Then, in a user-space process, the FIFO must be opened before it can be used:

```
Char mesg[] = “Data to be put on the FIFO”;

   fifo_id = Open(“/dev/rtf1”, O_READONLY);
   written = write(fifo_id, &mesg, sizeof(mesg) );
```

fifo_id is the file descriptor associated with the ‘opened’ FIFO which is used in all other accesses to it.

In a kernel module, RT-FIFOs are created with:

```
   rtf_create(fifo_number, size);

eg:rtf_create(1, 1000);
```

which creates an RT-FIFO of initial size 1000 bytes and assigns it the identifier 1, thus opening /dev/rtf1 from the real-time domain.

Handler Functions

RTAI FIFO's can be associated with command handlers. These behave like callback routines that get called when a user space routine either reads or writes to a RT-FIFO. These callback routines are usually implemented in conjunction with rtf_put or rtf_get operations in the handler routine.

To define the handler, you simply write a function in your RT module that performs the steps you require. The function prototype for this handler function is shown below. Note that when called, the handler function is passed the number of the fifo that has been read/written.

Note, the newfifos module passes an addition option character to the handler (either 'r' or 'w'), this can be used to determined if the user space operation was a read or write.

- int my_handler(unsigned int fifo);

To install the handler you need to make the following call, usually this will be done as part of the init_module() module initialisation function:

- rtf_create_handler(fifo_number, my_handler);

Example (Handler)

A handler code example is shown below:

```
int x_handler(unsigned int fifo, int rw);
        if (rw == 'r') {
// handle a read call and return appropriate value.
        } else {
// handle a write call and return appropriate value.
        }
}
```

Signal interface

Asynchronous signals can be used to announce that data is available on a FIFO.

In order to enable asynchronous signals, use:

- `rtf_set_async_sig(int fd, int signum)`

Note that the default signum is SIGIO.

Select/poll and Blocking

Since RTAI's RT-FIFOs allow multiple readers and writers, the select/poll mechanism which is used for synchronization can lead to unexpected blocks.

For example: An application polls and finds that data is available, meanwhile another application can get access to the FIFO and read (or steal) the data.

Then, when the original application comes back to read the data, it finds that the FIFO is empty and blocks.

To avoid these type of problems you can use the following functions:

- `rtf_read_all_at_once(fd, buf, count);`
  -- Blocks until all ‘count’ bytes are available
- `rtf_read_timed(fd, buf, count, ms_delay);`
- `rtf_write_timed(fd, buf, count, ms_delay);`
  -- Blocks just for the specified delay (in milli seconds) but are queued in real-time, Linux process, priority order. If ms_delay is zero then they return immediately with all the data they could get, even if you did not set O_NONBLOCK when the FIFO was created. So, by mixing normal reads and writes with these functions above, you can easily implement blocking, non-blocking and timed I/O. These calls are not standard, thus they are not portable, but they are far easier to use then the select/poll mechanism. The standard llseek is also available but it is equivalent to calling *rtf_reset*, no matter which place in the FIFO you point at in the call.

For another method of waiting, you have available also:

- `rtf_suspend_timed(fd, ms_delay).`

Semaphore interface

RTAI's semaphores define the maximum number of simultaneous processes that may access a critical code section and they can be set to *any* value. As such, they are very flexible and, among other uses, they can be used to synchronize shared memory access without any scheduler installed, or in place of blocking FIFO read/writes with dummy data. A mutex can be simply created by giving the semaphore a value of one.

RTAI's semaphores interact with each process in one of four ways:

1. The process can be queued up, by the semaphore, in priority order
2. The semaphore can wait for an event to occur and return immediately
3. The semaphore can wait for an event to occur or wait until a specific time, whichever comes first
4. The semaphore can wait for an event to occur or wait for a time delay, whichever comes first.

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 with a FIFO and you must get a file descriptor by opening the corresponding FIFO.

Note that these functions are symmetrically available in kernel space with the provision that the FIFO is non-blocking.

Named FIFOs

To make it easier to keep track of which FIFO to use and in order to avoid FIFO number clashes between separate real time tasks, RTAI allows the creation of named FIFOs.

Additionally, 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);`

The above functions are symmetrically available in kernel and user space, both returning the allocated FIFO number. In user space, note that these calls will not automatically open the FIFO device for you. Instead you must append the returned FIFO number onto the end of '/dev/rtf' and then open it in the normal way.

The maximum length of a FIFO's name is defined as RTF_NAMELEN, which is currently set to 15 characters.

When using rtf_ create_ named() from user space you may notice that the first FIFO created is assigned a FIFO number of 1 rather than 0. This is a side effect of the implementation mechanism, and is harmless.

If you want to monitor the FIFO name to number mapping you have two choices. 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.

Printk

RTAI includes the *rt_printk* function which allows you to safely use printk like messages in RTAI modules. It is complemented by *rt_print_to_screen* which is useful if you do not need to log messages.

- `rt_printk( const char *fmt, ...),`
- `rt_print_to_screen(const char *fmt, …)`

RT FIFOs API Summary

Kernel Space calls:

int rtf_init(void);

int rtf_create_handler(unsigned int fifo, int (*handler)(unsigned int fifo));

int rtf_create(unsigned int fifo, int size);

int rtf_create_named(const char *name);

int rtf_getfifobyname(const char *name);

int rtf_reset(unsigned int fifo);

int rtf_destroy(unsigned int fifo);

int rtf_resize(unsigned int minor, int size);

int rtf_put(unsigned int fifo, void * buf, int count);

int rtf_get(unsigned int fifo, void * buf, int count);

int rtf_sem_init(unsigned int fifo, int value);

int rtf_sem_post(unsigned int fifo);

int rtf_sem_trywait(unsigned int fifo);

int rtf_sem_destroy(unsigned int fifo);

```
int rt_printk(const char *fmt, ...);

int rt_print_to_screen(const char *fmt, ...);


/* For compatibility with earlier rtai_fifos releases. No more bh and user
buffers. Fifos are now awakened immediately and buffers > 128K are
vmalloced */

#define rtf_create_using_bh(fifo, size, bh_list) rtf_create(fifo, size)

#define rtf_create_using_bh_and_usr_buf(fifo, buf, size, bh_list)

 rtf_create(fifo, size)

#define rtf_destroy_using_usr_buf(fifo) rtf_destroy(fifo)
```

User Space:

```
int rtf_reset(int fd);

int rtf_resize(int fd, int size);

void rtf_suspend_timed(int fd, int ms_delay);


int open( char *device, mode_t mode);

int read(int fd, void *buf, size_t count);

int write (int fd, void *buf, size_t count);

void close(int fd);


int rtf_open_sized(const char *dev, int perm, int size);

int rtf_read_all_at_once(int fd, void *buf, int count);

int rtf_read_timed(int fd, void *buf, int count, int ms_delay);

int rtf_write_timed(int fd, void *buf, int count, int ms_delay);

void rtf_sem_init(int fd, int value);

int rtf_sem_wait(int fd);
```

int rtf_sem_trywait(int fd);

int rtf_sem_timed_wait(int fd, int ms_delay);

void rtf_sem_post(int fd);

void rtf_sem_destroy(int fd);

void rtf_set_async_sig(int fd, int signum);

int rtf_getfifobyname(const char *name);

int rtf_create_named(const char *name);

Example:

Many of the examples within the RTAI distribution illustrate the basic RT-FIFOs API. The ‘tasktimer’ example shown here, demonstrates many of the RT-FIFO API calls within a single, working example:

```
/*
FILE: Rt_Process.c

COPYRIGHT (C) 1999  Paolo Mantegazza (mantegazza@aero.polimi.it)

This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307
  USA.
*/

#define TICK 1000000 //ns (!!!!! CAREFULL NEVER GREATER THAN 1E7 !!!!!)
```

```
#define LOOPS 80 // dot products for each cpu
#define DIM   300 // size of the dot product vectors
#define MUL   3.141592 // a number to do something
#define RESULT (LOOPS*MUL*MUL*DIM*(DIM + 1)/2.0)
#define SECS_STEP 3E9 //ns, macro to control the the period of the print
  task

/* simple module to exemplify the use of RTAI. However it can be used
  */  /* as it is for a periodic time control, just change the dummy
  */ /* calculation and I/O */

#include <linux/module.h>
#include <asm/io.h>
#include <rtai.h>
#include <rtai_sched.h>
#include <rtai_fifos.h>

#define DTF   0
#define CMDF  1
#define ECHOF 2

#define FLOAT float

static unsigned long out_secs, out_avrj, out_maxj, out_dot,
  out_timdot[2];
static int cpu_used[NR_RT_CPUS];
static volatile int go;

static void print_times(int arg)
{
       while(1) {
                rtf_put(ECHOF, &out_secs, sizeof(out_secs));
                rtf_put(ECHOF, &out_avrj, sizeof(out_avrj));
               rtf_put(ECHOF, &out_maxj, sizeof(out_maxj));
             rtf_put(ECHOF, &out_dot, sizeof(out_dot));
            rtf_put(ECHOF, &out_timdot[0], sizeof(out_timdot[0]));
            rtf_put(ECHOF, &out_timdot[1], sizeof(out_timdot[1]));
             rtf_put(ECHOF, &cpu_used[0], sizeof(cpu_used[0]));
             rtf_put(ECHOF, &cpu_used[1], sizeof(cpu_used[1]));
           rt_task_wait_period();
      }
}

static FLOAT  a[NR_RT_CPUS*LOOPS][DIM], b[DIM], c[NR_RT_CPUS*LOOPS];
```

```
// gcc compiler is smart in getting a very tight optimised loop for this
static FLOAT dot(FLOAT *a, FLOAT *b, int n)
{
       int i = n - 1;
       FLOAT s = 0.0;
       for(; i >= 0; i--) {
              s = s + a[i]*b[i];
       }
         return s;
}

// this is an example of a periodic controller that does a lot of fp
// calculations and keeps the Linux timer handling alive at due time
// it also controls the jitter and toggle a bit on the parallel port and
// strain fifos with a lot of (dummy) data
// the computer load it entails is controlled by TICK, LOOPS and DIM
// macros

#define NREC 1000
#define RECSIZE 5000
static struct { int rec; char buf[RECSIZE - sizeof(int)];} record;
static volatile int sync;
static volatile RTIME tg, tick_period;
static int timer_freq;

static void task1(int arg)
{
       static volatile RTIME t0, t;
       static volatile int secs, bit, first = 0;
       static volatile int avrjitter, maxjitter;
       volatile double s;
       volatile int i, jitter, cpuid;
         t0 = rdtsc();
       while(1) {
              t = tg = rdtsc();
              outb(bit = !bit, 0x378);
              cpuid = hard_cpu_id();
              cpu_used[cpuid]++;
              sync = smp_num_cpus;
                 jitter = imuldiv(tick_period, cpu_freq, timer_freq) -
                                              (int)(t - t0);
              t0 = t;
              if (jitter < 0) jitter = -jitter;
```

```
            avrjitter = (avrjitter + jitter)>>1;
               if (jitter > maxjitter && first > 100)
                                 maxjitter = jitter;
            first++;
            s = 0.0;
               for(i = cpuid*LOOPS; i < (cpuid + 1)*LOOPS; i++) {
                  s += (c[i] = dot(a[i], b, DIM));
            }
               t = rdtsc() - tg;
                if (atomic_dec_and_test((atomic_t *)&sync)) {
                   if (go) {
                         record.rec += RECSIZE;
                              rtf_put(DTF, &record, sizeof(record));
                   }
                   t *= 10;
            }
            secs += TICK/1000;
            out_secs = secs/1000000;
                out_avrj = imuldiv(avrjitter, 1E6, cpu_freq);
                out_maxj = imuldiv(maxjitter, 1E6, cpu_freq);
            out_dot = (int)s;
               out_timdot[cpuid] = -imuldiv((int)t, 1E6, cpu_freq);
            rt_task_wait_period();
         }
}

static void task2(int arg)
{
      static volatile RTIME t;
      volatile int i, cpuid;
      static int bit = 0;

      while(1) {
             cpuid = hard_cpu_id();
             cpu_used[cpuid]++;
            outb(bit = !bit, 0x378);
               for(i = cpuid*LOOPS; i < (cpuid + 1)*LOOPS; i++) {
                   c[i] = dot(a[i], b, DIM);
            }
              t = rdtsc() - tg;
            if (atomic_dec_and_test((atomic_t *)&sync)) {
                   if (go) {
                         record.rec += RECSIZE;
                             rtf_put(DTF, &record, sizeof(record));
```

```
                }
                t *= 10;
            }
               out_timdot[cpuid] = imuldiv((int)t, 1E6, cpu_freq);
             rt_task_wait_period();
        }
}

static char buf[NREC*RECSIZE];

static int start_stop(unsigned int fifo)
{
        rtf_get(CMDF, &fifo, 1);
        rtf_reset(DTF);
        record.rec = 0;
        go = !go;
        return 1;
}

static RT_TASK Task1;
static RT_TASK Task2;
static RT_TASK Task3;

int init_module(void)
{
        RTIME now;
          int linux_cr0, i, k, linux_fpu_reg[27], timer_cpu;

          save_cr0_and_clts(linux_cr0);
          save_fpenv(linux_fpu_reg);
        for (i = 0; i < DIM; i++) {
               b[i] = MUL;
        }
        for (i = 0; i < NR_RT_CPUS*LOOPS; i++) {
               for (k = 0; k < DIM; k++) {
                        a[i][k] = (k + 1)*MUL;
                   }
         }
         restore_fpenv(linux_fpu_reg);
         restore_cr0(linux_cr0);
            printk("<>>> FP RESULT CHECK %d <<<>\n", (int)RESULT);

           rtf_create_using_bh_and_usr_buf(DTF, buf, NREC*RECSIZE, 0);
         rtf_create(CMDF, 100);
```

```
        rtf_create_handler(CMDF, start_stop);
        rtf_create(ECHOF, 1000);
        rt_task_init(&Task1, task1, 0, 4000, 0, 1, 0);
        rt_task_init(&Task2, task2, 0, 4000, 0, 1, 0);
           rt_task_init(&Task3, print_times, 0, 2000, 1, 0, 0);
        tick_period = start_rt_timer(nano2count(TICK));
        if ((timer_cpu = rt_get_timer_cpu()) > 0) {
             rt_set_runnable_on_cpus(&Task1, timer_cpu);
                rt_set_runnable_on_cpus(&Task2, timer_cpu == 1 ? 2 : 1);
             timer_freq = FREQ_APIC;

        } else {
             rt_assign_irq_to_cpu(TIMER_8254_IRQ, 0);
             rt_set_runnable_on_cpus(&Task1, 1);
             rt_set_runnable_on_cpus(&Task2, 2);
             timer_freq = FREQ_8254;

        }
        now = rt_get_time() + 5*tick_period;
        rt_task_make_periodic(&Task1, now, tick_period);
        rt_task_make_periodic(&Task2, now, tick_period);
          rt_task_make_periodic(&Task3, now + nano2count(SECS_STEP),
  nano2count(SECS_STEP));
        return 0;
}

void cleanup_module(void)
{
       int cpuid;
       rt_reset_irq_to_sym_mode(TIMER_8254_IRQ);
       stop_rt_timer();
       rt_busy_sleep(1E7);
       rt_task_delete(&Task3);
       rt_task_delete(&Task2);
       rt_task_delete(&Task1);
      rtf_destroy_using_usr_buf(DTF);
      rtf_destroy(CMDF);
      rtf_destroy(ECHOF);
      printk("\n\nCPU USE SUMMARY\n");
      for (cpuid = 0; cpuid < NR_RT_CPUS; cpuid++) {
                printk("# %d -> %d\n", cpuid, cpu_used[cpuid]);
      }
        printk("END OF CPU USE SUMMARY\n\n");
        return;
```

```
}

/*
FILE: Check.c

COPYRIGHT (C) 1999  Paolo Mantegazza (mantegazza@aero.polimi.it)

This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307
  USA.
*/
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <signal.h>

static int end;

static void endme (int dummy) { end = 1; }

int main(void)
{
    int rtf;
     unsigned long out_secs, out_avrj, out_maxj, out_dot, out_timdot[2];
      int cpu_used[2];
      if ((rtf = open("/dev/rtf2", O_RDONLY)) < 0) {
             fprintf(stderr, "Error opening /dev/rtf2\n");
             exit(1);
      }

        signal (SIGINT, endme);

       while(!end) {
```

```
            read(rtf, &out_secs, sizeof(out_secs));
            read(rtf, &out_avrj, sizeof(out_avrj));
            read(rtf, &out_maxj, sizeof(out_maxj));
            read(rtf, &out_dot, sizeof(out_dot));
                read(rtf, &out_timdot[0], sizeof(out_timdot[0]));
                read(rtf, &out_timdot[1], sizeof(out_timdot[1]));
                read(rtf, &cpu_used[0], sizeof(cpu_used[0]));
                read(rtf, &cpu_used[1], sizeof(cpu_used[1]));
               printf("<>RT_HAL time: %ld s, AvrJ: %ld, MaxJ: %ld us
  (%ld,%ld,%ld)<> %d %d \n", out_secs, out_avrj, out_maxj, out_dot,
  out_timdot[0], out_timdot[1], cpu_used[0], cpu_used[1]);
        }
}

/*
FILE: todisk.c

COPYRIGHT (C) 1999  Paolo Mantegazza (mantegazza@aero.polimi.it)

This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307
  USA.
*/
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <sched.h>

#define NREC 100
#define RECSIZE 5000
#define FILSIZE 300000000
struct { int rec; char buf[RECSIZE - sizeof(int)];} record[NREC];
```

```
int main(void)
{
     int rtf, cmd, i, k;
     int size, fd, count, lost;
      struct sched_param mysched;

     mysched.sched_priority = 99;

      if( sched_setscheduler( 0, SCHED_FIFO, &mysched ) == -1 ) {
           puts(" ERRORE SETTAGGIO SCHEDULER ");
           perror( "errno" );
           exit( 0 );
     }


      if ((rtf = open("/dev/rtf0", O_RDONLY)) < 0) {
            fprintf(stderr, "Error opening /dev/rtf0\n");
            exit(1);
     }
     if ((cmd = open("/dev/rtf1", O_WRONLY)) < 0) {
            fprintf(stderr, "Error opening /dev/rtf1\n");
            exit(1);
     }
     printf("TRUNCATING\n");
      fd = open("dumpfile", O_WRONLY | O_CREAT | O_TRUNC, 0666);
     write(cmd, &cmd, 1);
     printf("GO\n");
      size = 0;
     lost = 0;
     do {
             k = read(rtf, &record, sizeof(record));
             size += k;
             if (size != record[k/RECSIZE-1].rec) {
                 lost += record[k/RECSIZE-1].rec - size;
                   printf("%d %d %d\n", size, record[k/RECSIZE-1].rec,
                                                      lost);
                   size = record[k/RECSIZE-1].rec;
              }
              write(fd, &record, k);
       } while(size < FILSIZE);
       write(cmd, &cmd, 1);
       close(fd);
         printf("END %d %d %d\n", size, record[k/RECSIZE-1].rec, lost);
}
```

FIFOs readme (from RTAI Distribution)

Here you'll find a new fifo implementation for RTAI. It maintains full compatibility with the basic services provided by its original NMT-RTL counterpart while adding some more.

Among the new added services there is the porting of David Schleef (ds@stm.lbl.gov) rt_printk(const char *fmt, ...), that allows you to safely use printk like messages in RTAI modules. It is complemented by rt_print_to_screen(const char *fmt, ...), to be used if you do not need to log messages. See rtai_fifos.h for the calling prototypes.

It is important to remark that even if the RTAI fifo API appears as before the implementation behind it is based on the mailboxes 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 what it has been called for. It is usefull 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') {

// do stuff for a call from read and return appropriate value.

} else {

// do stuff for a call from write and return appropriate value.

```
}

}
```

Even if fifos are strictly no more required in RTAI, because of the availability of LXRT, fifos are kept for both compatibility reasons and because they are very useful tools to be used to 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 you have your interrupt handler installed you can use fifo services to do all the rest.

However the new implementation made it easy to add some new services. One of these is the possibility of using asyncronous signals to notify data availability by catching a user set signal. 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.

It is important to remark that now fifos allows multiple readers/writers so the select/poll mechanism to synchronize with in/out data can lead to unexpected blocks for such cases. For example: you poll and get that there are data available, then read/write them sure not to be blocked, meanwhile another user gets into and stoles all of your data, when you ask for them you get blocked.

To avoid such problems you have available the functions:

- rtf_read_all_at_once(fd, buf, count);

that blocks till 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 millisecs but are queued in real time Linux process priority order. If ms_delay is zero they return immediatly 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 so not portable, but far easy to use then 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 exist already it is opened, it is created with a 1K buffer. any following creation on modules 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 our mailboxes we have made available also binary semaphores. They can be used for many things, e.g. to sinchronize shared memory access without any scheduler installed and in place of using blocking fifos read/writes with dummy data, just to synchronize. 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 but, except for init and create, only for the nonblocking services, i.e: trywait and post.

A final, important, warning. All the new services have been tested in relation to their basic working, while the standard RTL calls worked well on all the i examples they worked before. Thus you will not miss anything with respect to either RTL fifos or the previous adaptation of RTAI to them. We hope in some help in thorougly verifying all the remaining new stuff. To stay on the safe side we default the installation to newfifos but keep old fifos available. See README in this and oldfifos directories.

Note that this directory contains an examples that shows the use of select, timed reads and semaphores.

To use it:

insmod task // for a real time task, a copy of the latency calibration task;

./check // to see the interaction;

check ends by itself. See the macros on top of check.c to change the execution

parameters.

Shared Memory

Overview

There are currently three implementations of Shared Memory within the RTAI distribution: ‘mbuff’, ‘shmem’ and ‘portable_shm’. This description covers only the ‘mbuff’ and ‘shmem’ implementations, as the ‘portable_shm’ was an RTAI extension of ‘mbuff ‘, which is now obsolete and will be removed in a later release.

In a similar way to FIFOs, shared memory provides a way to easily transfer data between real-time and user space tasks in which a portion of physical memory is set aside for sharing between Linux processes and real-time Linux tasks. But unlike FIFOs, shared memory is able to easily pass large amounts of data in one step. The decision to use FIFOs versus shared memory should be based on the natural communication model of the application.

Shared memory has the following characteristics:

- Shared memory does not queue data written to it. Applications requiring handshaking must define a protocol to assure data is not overwritten.
- As data is not queued, individual items in data structures of varying size may be updated without the need for sequential access.
- Shared memory has no point-to-point restriction. Shared memory can be written or read by any number of Linux processes or real-time Linux tasks.

- The number of independent shared memory channels is only limited by the size of physical memory.
- Blocking for synchronization is not directly supported. To determine if data is new, the data must contain a count that can be compared against previous reads or another handshaking mechanism used.
- Mutual exclusion of Linux and Real-Time Linux processes is not guaranteed.
- Interrupted reads and writes cannot be detected. If a requirement, they must be protected by mutexes or other similar mechanisms.

Mbuff vs Shmem

Mbuff is a shared memory implementation developed by Tomasz Motylewski that allows User Space processes to create and share areas of memory with Real time tasks, without requiring RTAI.

Shmem is the RTAI version, developed by Paolo Mantagazza, which operates in much the same way but is dependant upon RTAI.

Implementation

Implementation involves the following steps:

1. Create the shared memory device node. Because this newly created device node remains on your system, you don't have to re-create it every time you run the application. It is implemented as a real device driver hence, it needs to have a "device node" created.

   If you want to request a particular mapping address use the following function:

```
adr = rtai_malloc_adr( start_ address, name, size);
```

   Note that success of mapping on this address is not guaranteed and depends upon whether your user space application has this address range free at the time of the call.

2. Build and load the appropriate kernel module.
3. Load the application modules.

The rtai_shm system that provides the shared memory capability is implemented as a real device driver hence, it needs to have a "device" created. This device resides in the /dev directory and is called ‘rtai_shm’.

The top-level RTAI makefile has a target to build the necessary device inodes and so it is a good idea to use this in order to ensure that RTAI has been fully and properly installed.

```
cd <rtai>

make cleandev

make dev
```

This uses mknod to make the following device:

```
mknod  /dev/rtai_shm c 10 254
```

Build and insert the Shared Memory Module

```
cd <rtai>/shmem

make

insmod rtai_shm
```

At this stage it is a good idea to run the tests that accompany this module to ensure that RTAI has been properly installed and the Shared Memory Module built correctly.

cd <rtai>/shmem/test

cat README- describes the test, how to run it and what it does

make- make the test executables

./start- start the test

./ctest- alter data in shared memory

./stop- stop the test

Using Shared Memory in User Space

In order to access shared memory from tasks running in user space all you need is:

```
adr = rtai_malloc(name, size);
```

Where *name* is a simple long integer identifier and *size* the number of bytes to be reserved for this block of shared memory. Note that memory is reserved in ‘chunks’ of 4096 bytes. The return parameter, adr, is the base address of the allocated area of shared memory.

If you want to ‘force’ the allocation to a specific memory address, you can use the following function:

```
adr = rtai_malloc_adr(start_address, name, size);
```

To de-allocate the area of shared memory:

```
rtai_free(name, adr);
```

Using Shared Memory in Kernel Space

In order to access shared memory from real-time tasks running in kernel space all you need is:

```
adr = rtai_kmalloc(name, size);
```

Where *name* is a simple long integer identifier and *size* the number of bytes to be reserved for this block of shared memory. Note that memory is reserved in ‘chunks’ of 4096 bytes. The return parameter, *adr*, is a pointer to the base address of the allocated area of shared memory. This pointer can then be used to access the data in the shared memory area.

The first allocation does a real allocation. Subsequent calls to allocate with the same name in User Space just map the area in User Space or return the related pointer to the already allocated area in Kernel Space.

To de-allocate the area of shared memory:

```
rtai_kfree(name);
```

Example

The following example is distributed with RTAI in the shmem/test directory. It comprises a kernel module (kalloc) and two user space programs: itest and ctest.

Initially, rtai, rtai_shm and kalloc are loaded, then itest and ctest are run to manipulate data in the shared memory blocks created. Note that this example makes use of test routines not shown here and features unbalanced allocation and de-allocation.

```
insmod rtai
insmod rtai_shm
insmod kalloc
./itest &
./ctest     (you can run this any number of times)
...
rmmod kalloc
```

```
// FILE: kalloc.c
//
#include <linux/version.h>
#include <linux/module.h>
#include <linux/config.h>
#include <linux/types.h>
#include <linux/errno.h>
#include <linux/fcntl.h>
#include <linux/mm.h>
#include <linux/miscdevice.h>
#include <linux/malloc.h>
#include <linux/wrapper.h>

#include <asm/uaccess.h>
#include <asm/pgtable.h>
#include <asm/io.h>

#include "rtai_shm.h"

#define SIZE 5000

int init_module (void)
{
       void *adr;
```

```
         adr = rtai_kmalloc(0xaaaa, SIZE);
         memset(adr, 255, SIZE);
         return 0 ;
}

void cleanup_module (void)
{
       rtai_kfree(0xaaaa);
       return;
}

// FILE: itest.c
//
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>

#include "rtai_shm.h"

#define MEMSIZE 3000

main()
{
      unsigned int *adr, *adr1, i;
        printf("\nALLOCATING %x AND %x IN INITIAL PROCESS\n",
                          0xabcd, 0xaaaa);
      adr = rtai_malloc(0xabcd, 4*MEMSIZE);
      adr1 = rtai_malloc(0xaaaa, 1);
      rtai_malloc(0xabcd, 4*MEMSIZE);
       rtai_malloc(0xaaaa, 1);
      rtai_malloc(0xffff, 1);
       printf("THE FIRST VALUES OF %x AND %x ARE %d %d\n",
                         0xabcd, 0xaaaa, adr[0], adr1[0]);
      adr[0] = adr1[0] = 999999;
      printf("WE CHANGE THEM TO %d\n", adr[0]);
      rtai_check(0xabcd);
        printf("THE MODULE CHANGED THEM TO %d %d\n", adr[0], adr1[0]);

      while (!(i = rtai_is_closable())) sleep(1);
      rtai_not_closable();
```

```
      printf("\nFREEING %x AND %x IN INITIAL PROCESS\n",
                                0xabcd, 0xaaaa);
       rtai_free(0xabcd, adr);
      rtai_free(0xaaaa, adr1);
      rtai_free(0xffaa, adr1);
      rtai_free(0xffaf, adr1);
}

// FILE: ctest.c
//
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>

#include "rtai_shm.h"

#define MEMSIZE 3000

main()
{
        unsigned int *adr, *adr1;
           printf("\nALLOCATING %x AND %x IN CURRENT PROCESS\n",
                                   0xabcd, 0xaaaa);
        adr = rtai_malloc(0xabcd, 4*MEMSIZE);
        adr1 = rtai_malloc(0xaaaa, 1);
          rtai_malloc(0xaaaa, 1);
        rtai_malloc(0xaaaa, 1);
        rtai_malloc(0xaabc, 1);
           printf("THE FIRST VALUES OF %x AND %x ARE %d %d\n",
                                0xabcd, 0xaaaa, adr[0], adr1[0]);
          adr[0] = adr1[0] = 999999;
        printf("WE CHANGE THEM TO %d\n", adr[0]);
        rtai_check(0xabcd);
          printf("THE MODULE CHANGED THEM TO %d %d\n", adr[0], adr1[0]);
          printf("\nFREEING %x AND %x IN CURRENT PROCESS\n",
                                0xabcd, 0xaaaa);
          rtai_free(0xabcd, adr);
        rtai_free(0xaaaa, adr1);
        rtai_free(0xabcd, adr);
        rtai_free(0xffff, adr);
}
```

MBUFF

The ‘mbuff’ system that provides the shared memory capability is implemented as a real device driver hence, it needs to have a "device node" created. This device resides in the /dev directory and is called ‘mbuff’.

Note that the top-level mbuff Makefile contains a target (mbuff) to create the necessary device:

```
mknod  /dev/mbuff c 10 254
```

Build and insert the Shared Memory Module

```
cd <mbuff>

make

insmod mbuff.o
```

Using Mbuff in User Space

In order to access shared memory from tasks running in user space all you need is:

```
mbuf = mbuff_alloc(name, size);
```

Where *name* is a pointer to a character string (maximum length or 32 characters) and *size* the number of bytes to be reserved for this block of shared memory. Note that memory is reserved in ‘chunks’ of 4096 bytes. The return parameter, mbuf, is the base address of the allocated area of shared memory.

The first allocation does a real allocation. Subsequent calls to allocate with the same name in User Space just map the area in User Space or return the related pointer to the already allocated area in Kernel Space. Note that the addresses to reference the same block of shared memory will always be different for user space and kernel space.

Use *mbuff_alloc_at* to map the area at a specific address:

```
mbuf = mbuff_alloc_at(name, size, adr);
```

To de-allocate the area of shared memory:

```
mbuff_free(name, mbuf);
```

Note that each *mbuff_alloc* call should have a corresponding *mbuff_free* call otherwise some buffers will not be de-allocated and you'll get a memory leak.

There are also calls available to allocate and free shared memory without changing the usage counters:

```
mbuf = mbuff_attach( name, size);

mbuf = mbuff_attach_at( name, size, mbuf);

mbuff_detach( name, mbuf);
```

Note that these calls are normally only made from user space, however mbuff_attach/mbuff_detach are available from the kernel via macros that will actually call mbuff_alloc/mbuff_free.

In order to access shared memory from tasks running in kernel space all you need is:

```
mbuf = mbuff_alloc(name, size);
```

And to de-allocate it:

```
mbuff_free(name, mbuff);
```

The calls to *mbuff_attach* and *mbuff_detach* are supported but are implemented as calls to *mbuff_alloc* and *mbuff_free* respectively.

Example

The example shown below is the 'demo.c' example that is distributed as part of the 'mbuff' project:

```
// FILE: demo.c
#include <stdio.h>
#include "mbuff.h"

/* the contents of shared memory may change at any time, thus volatile */
volatile char * shm1, *shm2;

main (int argc,char *argv[]){

          shm1 = (volatile char*) mbuff_alloc("demo1",1024*1024);
          shm2 = (volatile char*) mbuff_alloc("demo1",1024*1024);
        if( shm1 == NULL || shm2 == NULL ) {
```

```
                printf("mbuff_alloc failed\n");
                exit(2);
        }
        sprintf((char*)shm1,"example data\n");
         sleep(5); /* you may change it from the kernel or other program
 here */
          printf("shm1=%p shm2=%p shm2->%s", shm1, shm2, shm2);
        mbuff_free("demo1",(void*)shm1);
        sleep(3);
  /* you may still access shm2 here, it is still the same memory area */
        mbuff_free("demo1",(void*)shm2);
        return(0);
}
```

/PROC

Note that ‘mbuff’ has a /proc interface (/proc/mbuff) that presents the following information:

version: the installed version of mbuff

regions/: for each mbuff ‘region’ (block)

count: the user space allocation

kcount: the kernel space allocation

open_cnt:open count (ie: no users)

open_mode: how it was opened

size: its size

Note finally that mbuff differs from RTAI shmem in 2 main ways:

1. In mbuff, the segment name is reference by a string
2. mbuff does not require any realtime extensions

readme.shmem (RTAI readme)

This directory contains an RTAI specific module that allows sharing memory inter-intra real time tasks and Linux processes. In fact it can be an alternative to SYSTEM V shared memory, the services are symmetricall, i.e. the

same calls can be used both in real time tasks, i.e. within the kernel, and Linux processes. The function calls for Linux processes are inlined in the file "rtai_shm.h". This approach has been preferred to a library since: is simpler, more effective, the calls are short, simple and just a few per process. They are:

```
#include <rtai_shm.h>
unsigned long name;
void *adr;
int size;

call to allocate memory:
adr = rtai_malloc_adr(adr, name, size);
                        // in   userspace
adr = rtai_malloc(name, size);    // in user space
adr = rtai_kmalloc(name, size);
            // in kernel (module) space

call to free memory:
rtai_free(name, adr);  //in user space
rtai_kfree(name);      //in kernel (module) space
```

The first allocation does a real allocation, any subsequent call to allocate with the same name from Linux processes just maps the area to the user space or return the related pointer to the already allocated space in kernel space. Analogously the freeing calls have just the effect of unmapping till the last is done, as that is the one the really frees allocated memory. Clearly cooperating users have to use the same "name".

The all stuff is based on an implementation of basic services made available by Tomasz (Tomek) Motylewski (motyl@stan.chemie.unibas.ch), i.e kvmem.h as obtained from motylewski.h.orig in this distribution. Such an implementation makes it easier for users to code the related services, as calls similar to Unix OS services can be used.

Many thanks again to Tomek for his help and patience in answering my questions. This shared memory implementation has been very fast, in my standard, thank to his code and help. A couple of technicalities:

1. I followed Tomek's idea to use the char misc_device, Linux seems to install by default on major = 10, with minor = 254. You can change it to your preferred value by changing the macro RTAI_SHM_MISC_MINOR in rtai_shm.c. It is registered at insmod so you should be warned if the value is inappropriate for your environment.

2. I used a fixed array list of allocated areas, instead of a list of pointers, as the related operations are not critical. You can make it to suite the size of your needs by changing the macro MAX_SLOTS in rtai_shm.c.

As explained above the allocated area is identified by an unsigned long. To use alphanumeric mnemonic terms a couple of very simple functions are available to translate a SIX CHARACTERs string into and unsigned long, both in kernel and user space. They are:

```
unsigned long nam2num(char *name);
void num2nam(unsigned long num, char *name);
```

So if you like to use them you can do:

```
adr = rtai_malloc(nam2num("myNAME"), size);
```

or

```
rtai_free(num2nam("myNAME"), adr);
```

Allowed characters are:

- English letters (no difference between upper and lower case);
- 10 digits;
- underscore (_) and another character of your choice. The latter will be always converted back to a $ by num2nam.

Paolo Mantegazza (mantegazza@aero.polimi.it).

Dynamic Memory Allocation

Overview

In early versions of RTAI it was necessary for real-time applications to allocate all of their memory before entering real time. To work around this, many

attempted to use the standard Linux memory allocation call of *kmalloc*, but they soon found that the use of this call could block the kernel, meaning that it could not safely be used from a RT task. This unfortunate aspect of kmalloc placed some significant restrictions on many applications whose implementation could benefit from dynamic behavior.

Now however, in versions of RTAI since v1.3, the Dynamic Memory Allocation module allows memory allocation and de-allocation calls to be used from real-time tasks.

Implementation

The rt_mem_mgr package can be installed as a stand-alone module or as part of the standard RTAI distribution. It is included in the RTAI distribution by default the memory manager code is linked directly into the schedulers so there is no need to install a separate module.

To use the package as a stand-alone module, you need to edit the Makefile in the <rtai>/rt_mem_mgr directory as directed in the README file, to turn on the KFLAGS macro, which adds the –DMODULE directive to the compiler. You can then insert the module by typing:

```
insmod <rtai>/rt_mem_mgr/rt_mem_mgr.o
```

API

To allocate memory:

```
addr = rt_malloc(size);
```

And to de-allocate it:

```
rt_free(addr);
```

Example:

The following lists the rt_mem_test.c file, which is a test example distributed with RTAI.

```
////////////////////////////////////////////////////////////////////
//
//
// Copyright (©) 2000 Pierre Cloutier (Poseidon Controls Inc.),
//                    Steve Papacharalambous (Zentropic Computing Inc.),
```

```
//                    All rights reserved
//
// Authors:           Pierre Cloutier (pcloutier@poseidoncontrols.com)
//                    Steve Papacharalambous (stevep@zentropix.com)
//
// Original date:     Wed 23 Feb 2000
//
//
// This library is free software; you can redistribute it and/or
// modify it under the terms of the GNU Lesser General Public
// License as published by the Free Software Foundation; either
// version 2 of the License, or (at your option) any later version.
//
// This library is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
// Lesser General Public License for more details.
//
// You should have received a copy of the GNU Lesser General Public
// License along with this library; if not, write to the Free Software
// Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307
USA.
//
// Dynamic Memory Management simple test program for Real Time Linux.
//
////////////////////////////////////////////////////////////////////////
//
static char id_rt_mem_test_c[] __attribute__ ((unused)) = "@(#)$Id:
rt_mem_test.c,v 1.1 2000/03/10 15:09:24 stevepapa Exp $";

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/version.h>
#include <linux/vmalloc.h>
#include <linux/errno.h>

#include <rtai.h>
#include <rtai_sched.h>
#include <rtai_fifos.h>
```

```
#include "rt_mem_mgr.h"


// ------------------------------< definitions
>------------------------
#ifndef NULL
#define NULL ((void *) 0)
#endif

#define TICK_PERIOD 50000000

#define STACK_SIZE 2000


//
--------------------------------------------------------------------------
//      Local Definitions.
//
--------------------------------------------------------------------------


//
--------------------------------------------------------------------------
//      Package Global Data.
//
--------------------------------------------------------------------------
RT_TASK mem_thread;


//
--------------------------------------------------------------------------
void mem_alloc(int t)
{

  unsigned int mem_size = 0x4000;
  void *mem_ptr1 = NULL;
  void *mem_ptr2 = NULL;
  void *mem_ptr3 = NULL;
```

```
  if((mem_ptr1 = rt_malloc(mem_size)) == NULL) {
    rt_printk("mem_alloc - Error Allocating %d bytes.\n", mem_size);
  } else {
    DBG("mem_alloc - Allocated %d bytes, address: %p.\n", mem_size,
                                                          mem_ptr1);
  }

  if((mem_ptr2 = rt_malloc(mem_size - 0x2000)) == NULL) {
    rt_printk("mem_alloc - Error Allocating %d bytes.\n", mem_size -
5);
  } else {
    DBG("mem_alloc - Allocated %d bytes, address: %p.\n", mem_size - 5,
                                                          mem_ptr2);
  }

  if((mem_ptr3 = rt_malloc(mem_size + 0x3000)) == NULL) {
    rt_printk("mem_alloc - Error Allocating %d bytes.\n", mem_size +
5);
  } else {
    DBG("mem_alloc - Allocated %d bytes, address: %p.\n", mem_size + 5,
                                                          mem_ptr3);
  }

//  display_chunk(mem_ptr1);
  rt_free(mem_ptr3);
  rt_free(mem_ptr1);
  rt_free(mem_ptr2);
//  display_chunk(mem_ptr1);
  rt_task_suspend(rt_whoami());

}  // End function - mem_alloc


//
--------------------------------------------------------------------

////////////////////////////////////////////////////////////////////
//
//
```

```
// Module Initialisation/Finalisation
//
////////////////////////////////////////////////////////////////////
//
int init_module(void)
{

  int r_c = 0;
  RTIME tick_period;

  if((r_c = rt_task_init(&mem_thread, mem_alloc, 0,
                         STACK_SIZE, 2, 1, 0)) < 0) {

    printk("rt_mem_test - Error creating mem_thread: %d\n", r_c);
  }

#ifdef ZDEBUG
  printk("rt_mem_test - Created memory allocation thread.\n");
#endif

  tick_period = start_rt_timer(nano2count(TICK_PERIOD));
  if((r_c = rt_task_make_periodic(&mem_thread, rt_get_time()
                              + (2 * tick_period), tick_period)) < 0) {

    printk("rt_mem_test - Error making mem_thread periodic: %d\n",
r_c);
  }

  return(r_c);

} // End function - init_module


//
--------------------------------------------------------------------
void cleanup_module(void)
{

  stop_rt_timer();
```

```
  rt_busy_sleep(1E7);
  rt_task_delete(&mem_thread);


} // End function - cleanup_module


// ---------------------------------< eof
>----------------------------
```

rt_mem_manager (RTAI readme)

```
Dynamic Memory Management for RTAI.
===================================

Copyright (©) 2000 Pierre Cloutier (Poseidon Controls Inc.),
                   Steve Papacharalambous (Zentropic Computing Inc.),
                   All rights reserved

Authors:           Pierre Cloutier (pcloutier@poseidoncontrols.com)
                   Steve Papacharalambous (stevep@zentropix.com)

Original date:       Sun 05 Mar 2000

This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA.

--------------------------------------------------------------------------------

This package contains an implementation of dynamic memory management for RTAI.
This allows real time tasks to allocate and free memory safely whilst
executing in the real time domain.

--------------------------------------------------------------------------------

Configuration Parameters.
-------------------------

Size of the memory chunks used for dynamic memory allocation can be changed to
suit application requirements.  To change the default size, set to 64 KBytes,
modify the global variable "granularity" and re-compile the package.
```

```
The default number of free chunks, set to 2, can be changed.  To change this
mofify the global variable "low_chk_ref" and re-compile the package.

The default low data mark, set to 512 bytes, which triggers the allocation of
another free chunk can by changed by modification of the global variable
"low_data_mark" and then re-compiling the package.


------------------------------------------------------------------------------

Limitations.
------------



------------------------------------------------------------------------------

Installation.
-------------

This package can be installed as a stand alone kernel module, or as part
of the RTAI distribution.  When part of the RTAI distribution it will
be configured and installed as part of the RTAI installation.

To use the package as a stand alone kernel module follow these instructions:

- Untar the archive:

tar zxvf rt_mem_mgr-<x.xx>.tar.gz

where x.xx is the package revision.


- Change to the memory manager directory:

cd rt_mem_alloc


- Edit the Makefile and uncomment the line:
```

```
# KFLAGS += -DMODULE


- Build the package:

make clean
make


- Install the kernel module:

insmod rt_mem_mgr.o


NB: To install the kernel module you must be super user.

------------------------------------------------------------------------------

Memory Manager API.
-------------------

The API calls for the memory menager are listed below:

void *rt_malloc(unsigned int size);
void rt_free(void *addr);

NB: rt_malloc returns NULL if an error occured.

------------------------------------------------------------------------------

TODO.
-----


------------------------------------------------------------------------------

Acknowledgements.
-----------------

- Paolo Mantegazza (mantegazza@aero.polimi.it) for the RTAI package, and for
his
```

```
  assistance and advice with this module.


- Victor Yodaiken (yodaiken@fsmlabs.com) and Micheal Baranbanov (baraban@fsm-
labs.com)
  for the RTLinux project.

-------------------------------------------------------------------------------
```

Mailboxes

Overview

The mailbox service allows messages between processes to be automatically stored and retrieved as needed in a priority queue.

The mailbox service is very flexible:

- It can be explicitly set up to accept messages of custom sizes.
- Multiple receivers and senders can be connected to the same mailbox where the order in which messages are taken depends on the priority of the receivers.
- When large messages need to be sent, the service provides functions to allow the process to send only the portion of the message that can be stored, returning the number of unsent bytes, or to continue to send the message until all of it has been accepted.

Naturally all the functions described below can also be used symmetrically from Linux processes, through the "lxrt" module. Thus mailboxes can be used instead of FIFOs. However be warned that the number of memcopy operations is doubled, so they can be slightly less efficient, although it is unlikely to be noticeable for relatively short messages. We believe that the advantage of symmetry is so high that it is worth such a very minor penalty.

Implementation

Mailboxes services are provided by the RTAI Scheduler and so applications must #include ‘rtai_sched.h’ to gain access to the Mailbox API and insmod ‘rtai_sched’ in order to make them available to kernel modules. A mailbox is created and initialized by calling the initialization function and passing to it a pointer to a pre-allocated Mailbox data structure, MBX and a parameter defining the required size, in bytes:

```
result = rt_mbx_init(MBX *mbx, size)
```

Where:

mbx refers to a pre-allocated, static mailbox data structure

size is the number of bytes reserved for the mailbox in total and is usually chosen as a multiple of the average size of the messages to be stored.

result will be zero if the mailbox was created correctly. A negative number signifies an error of some kind, (e.g.: too little memory available)

The mailbox is deleted with a call to `rt_mbx_delete(mbx).`

```
result = rt_mbx_delete(MBX *mbx)
```

Where:

mbx refers to a pre-allocated, static mailbox data structure

result will be zero if the mailbox was created correctly. A negative number signifies an error of some kind, (e.g.: invalid MBX)

Usage

Both send and receive can be operated:

unconditionally,

only for the bytes that can be sent or received immediately,

only if the whole message can be sent or received immediately,

timed absolutely or relatively.

Unconditional and timed mode can be used as synchronization tools, while conditional send/receive calls are useful if a non-blocking operation is desired.

Note that all send and receive functions generally return zero if successful, or a negative number if an error occurred (e.g.: an invalid MBX pointer specified). The conditional send functions, may return a positive number that indicates the number of bytes that could not be sent. The receive functions generally return the number of received bytes, which should usually be compared against the number of bytes requested.

MAILBOX API

This API has been re-produced from the README.MBX file:

```
// Initialize the mailbox pointed by mbx with a buffer of size bytes.
// Return == 0 is OK, != 0 error.
int rt_mbx_init(MBX *mbx, int size)

// Delete the mailbox pointed by mbx.
// Return == 0 is OK, != 0 error.
int rt_mbx_delete(MBX *mbx)

// Send unconditionally, i.e. return when the whole message has been
// sent or an error has occurred.
// Send the message pointed to by msg, of size msg_size, to the mailbox
// pointed at by mbx. Returns the number of unsent bytes.
int rt_mbx_send(MBX *mbx, void *msg, int msg_size)

// Send as much of a message as possible.
// Send the message pointed to by msg, of size msg_size, to the mailbox
// pointed at by mbx. Returns the number of unsent bytes.
int rt_mbx_send_wp(MBX *mbx, void *msg, int msg_size)

// Send a message only if it can all be sent immediately.
// Send the message pointed to by msg, of size msg_size, to the mailbox
// pointed at by mbx. Returns the number of unsent bytes.
```

int rt_mbx_send_if(MBX *mbx, void *msg, int msg_size)

// Send until: an absolute time has been reached or an error occurs.

// This call will try to send the whole message but if it cannot it will // return once the current time exceeds ‘time’.

// Send the message pointed to by msg, of size msg_size, to the mailbox

// pointed at by mbx. Returns the number of unsent bytes.

int rt_mbx_send_until(MBX *mbx, void *msg, int msg_size, RTIME time)

// Send until: a delay has expired or an error occurs.

// This call will try to send the whole message but if it cannot it will // return once the ‘delay’ wait period has expired.

// Send the message pointed to by msg, of size msg_size, to the mailbox

// pointed at by mbx. Returns the number of unsent bytes.

int rt_mbx_send_timed(MBX *mbx, void *msg, int msg_size, RTIME delay)

// Receive unconditionally, i.e. return when the all message has been

// received or an error occurred.

// Receive a message from the mailbox pointed by mbx. Store the received // message in the buffer pointed to by msg, whose size is msg_size

// bytes. Returns the number of received bytes.

int rt_mbx_receive(MBX *mbx, void *msg, int msg_size)

// Receive as much of a message as possible and return immediately.

// Receive a message from the mailbox pointed by mbx. Store the received // message in the buffer pointed to by msg, whose size is msg_size

// bytes. Returns the number of received bytes.

int rt_mbx_receive_wp(MBX *mbx, void *msg, int msg_size)

// Receive a whole message. Get the message only if there are msg_size

// bytes available, otherwise return immediately.

// Receive a message from the mailbox pointed by mbx. Store the received // message in the buffer pointed to by msg, whose size is msg_size

// bytes. Returns the number of received bytes.

int rt_mbx_receive_if(MBX *mbx, void *msg, int msg_size)

// Receive until: an absolute time has been reached or an error occurs.

// This call will try to receive the whole message but if it cannot it

// will return once the current time exceeds ‘time’.

// Receive a message from the mailbox pointed by mbx. Store the received // message in the buffer pointed to by msg, whose size is msg_size

// bytes. Returns the number of received bytes.

int rt_mbx_receive_until(MBX *mbx, void *msg, int msg_size, RTIME time)

// Receive until: a delay has expired or an error occurs.

// This call will try to receive the whole message but if it cannot it

// will return once the ‘delay’ wait period has expired.

// Receive a message from the mailbox pointed by mbx. Store the received // message in the buffer pointed to by msg, whose size is msg_size

// bytes. Returns the number of received bytes.

int rt_mbx_receive_timed(MBX *mbx, void *msg, int msg_size, RTIME delay)

Maxibox Example:

Examples of RTAI mailbox usage can be found in the following subdirectories of the examples directory:

- jepplin: a translation to RTAI of the same example found in the NMT-RTL distribution;
- mbx: a test with two senders and a receiver communicating with mailboxes of size less than the actual messages;
- lxrt/master_buddy: a test using a master process and a buddy process to demonstrate RTAI usage from Linux processes using lxrt and mailboxes.

The MBX example is re-produced here:

```
/*
FILE: rt_process.c

COPYRIGHT (C) 1999  Paolo Mantegazza (mantegazza@aero.polimi.it)

This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307
USA.
*/


#include <linux/module.h>
#include <asm/io.h>
```

```
#include <rtai.h>
#include <rtai_sched.h>

#ifdef  VERIFY_FLAGS
#define CHECK_FLAGS \
      { \
            unsigned long flags; \
            rt_global_save_flags(&flags); \
            if (flags != ((1 << IFLAG) | 1)) { \
                  printk("<<<<<<<<<< FLAGS: %lx >>>>>>>>>>\n", flags);
\
            } \
      }
#else
#define CHECK_FLAGS
#endif

#define TICK_PERIOD 1E5

#define DELAY       5E5

#define SLEEP_DELAY 1E5

#define STACK_SIZE 2000

static int cpu_used[NR_RT_CPUS];

static RT_TASK mtask[2], btask, wdog;

static MBX smbx, rmbx[2];

static unsigned long long name[2] = {
            0xaaaaaaaaaaaaaaaaLL, 0xbbbbbbbbbbbbbbbbLL };

static int stop, alarm;

void wfun(int t)
{
```

```
	while(stop) {
		if (alarm && !stop) {
			stop = 1;
			printk("LOCKED\n");
		}
		alarm = 1;
		rt_task_wait_period();
	}
}

void mfun(int t)
{
	unsigned long long msg;
	while(stop) {
		alarm = 0;
		CHECK_FLAGS;
		cpu_used[hard_cpu_id()]++;
		rt_mbx_send_timed(&smbx, &name[t], sizeof(long long),
			nano2count(DELAY));
		msg = 0;
		rt_mbx_receive_timed(&rmbx[t], &msg, sizeof(msg),
			nano2count(DELAY));
		if (msg != 0xccccccccccccccccLL) {
			printk(">EM %d %d<\n", t, stop);
		}
		rt_sleep(nano2count(SLEEP_DELAY));
	}
}

void bfun(int t)
{
	unsigned long long msg;
	unsigned long long name = 0xccccccccccccccccLL;
	while(stop) {
		alarm = 0;
		CHECK_FLAGS;
		cpu_used[hard_cpu_id()]++;
		rt_mbx_receive(&smbx, &msg, sizeof(msg));
		if (msg == 0xaaaaaaaaaaaaaaaaLL) {
```

```
                t = 0;
        } else {
                if (msg == 0xbbbbbbbbbbbbbbbbLL) {
                        t = 1;
                } else {
                        printk(">EB %x %x<\n", ((int *)&msg)[0],
                                                ((int *)&msg)[1]);
                        t = 0;
                }
        }
        rt_mbx_send(&rmbx[t], &name, sizeof(name));
    }
}

int init_module(void)
{
    int period;
    rt_mbx_init(&smbx, 5);
    rt_mbx_init(&rmbx[0], 1);
    rt_mbx_init(&rmbx[1], 3);
    rt_task_init(&wdog, wfun, 0, STACK_SIZE, 0, 0, 0);
    rt_task_init(&mtask[0], mfun, 0, STACK_SIZE, 0, 0, 0);
    rt_task_init(&mtask[1], mfun, 1, STACK_SIZE, 0, 0, 0);
    rt_task_init(&btask, bfun, 0, STACK_SIZE, 0, 0, 0);
    alarm = 0;
    stop = 1;
    rt_set_oneshot_mode();
    period = start_rt_timer(nano2count(TICK_PERIOD));
    rt_task_make_periodic(&wdog, rt_get_time() + period, period);
    rt_task_resume(&btask);
    rt_task_resume(&mtask[0]);
    rt_task_resume(&mtask[1]);
    return 0;
}

void cleanup_module(void)
{
    int cpuid;
```

```
        stop = 0;
        rt_busy_sleep(nano2count(1E7));
        stop_rt_timer();
        rt_task_delete(&mtask[0]);
        rt_task_delete(&mtask[1]);
        rt_task_delete(&btask);
        rt_task_delete(&wdog);
        printk("\n\nCPU USE SUMMARY\n");
        for (cpuid = 0; cpuid < NR_RT_CPUS; cpuid++) {
              printk("# %d -> %d\n", cpuid, cpu_used[cpuid]);
        }
        printk("END OF CPU USE SUMMARY\n\n");
}
```

RTAI Messages and Remote Procedure Calls (RPC)

Overview

RTAI provides a simple inter-task messaging facility whereby single, 32-bit values may be passed between real-time tasks. Remote Procedure Calls (RPCs) do the same thing but the tasks are coupled awaiting a reply from the receiver. RPCs operate like complementary, send and receive message pairs.

Implementation

Messaging and RPC services are provided by the RTAI Scheduler and so applications must #include ‘rtai_sched.h’ to gain access to the RPC API and insmod ‘rtai_sched’ in order to make them available to kernel modules.

Usage

```
insmod rtai
insmod rtai_sched
insmod your_kernel_module
```

Example:

The following example illustrates the use of messages and RPCs. It is taken from the ‘msgsw’ example within the RTAI distribution:

```
/*
FILE: rt_process.c

COPYRIGHT (C) 1999  Paolo Mantegazza (mantegazza@aero.polimi.it)

This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307
USA.
*/


#include <linux/module.h>

#include <asm/io.h>

#include <rtai.h>
#include <rtai_sched.h>

#define ONE_SHOT

/* the address of the parallel port -- you probably should change this
*/
#define LPT 0x378
```

```
#define TIMER_TO_CPU 3 // < 0 || > 1 to maintain a symmetric processed
timer.
//#define RUNNABLE_ON_CPUS (i%2 + 1) // forced statically half and half
#define RUNNABLE_ON_CPUS 3  // 1: on cpu 0 only, 2: on cpu 1 only, 3:
on any.
#define RUN_ON_CPUS (smp_num_cpus > 1 ? RUNNABLE_ON_CPUS : 1)

#define TICK_PERIOD 50000
#define STACK_SIZE 2000
#define LOOPS 10000

#define END 0xFFFF

#define NTASKS 8

RT_TASK thread[NTASKS];

static int cpu_used[NR_RT_CPUS];

RTIME tick_period;

void driver(int t)
{
      RT_TASK *thread[NTASKS];
      int i, l;
      unsigned int msg = 0;
      RTIME now;

      for (i = 1; i < NTASKS; i++) {
            thread[0] = rt_receive(0, &msg);
            thread[msg] = thread[0];
      }
      for (i = 1; i < NTASKS; i++) {
            rt_return(thread[i], i);
      }
      now = rt_get_time();
      rt_task_make_periodic(rt_whoami(), now + NTASKS*tick_period,
tick_period);
```

```
	msg = 0;
	l = LOOPS;
	while(l--) {
		for (i = 1; i < NTASKS; i++) {
			cpu_used[hard_cpu_id()]++;
			if (i%2) {
				rt_rpc(thread[i], msg, &msg);
			} else {
				rt_send(thread[i], msg);
				msg = 1 - msg;
			}
			rt_task_wait_period();
		}
	}
	for (i = 1; i < NTASKS; i++) {
		rt_send(thread[i], END);
	}
	rt_task_delete(rt_whoami());
}


void fun(int t)
{
	unsigned int msg;
	rt_rpc(&thread[0], t, &msg);
	while(msg != END) {
		cpu_used[hard_cpu_id()]++;
		rt_receive(&thread[0], &msg);
		outb((msg & 1), LPT);
		if (rt_isrpc(&thread[0])) {
			rt_return(&thread[0], 1 - msg);
		}
	}
	outb(0, LPT);
	rt_task_delete(rt_whoami());
}


int init_module(void)
```

```
{
      int i;

#ifdef ONE_SHOT
      rt_set_oneshot_mode();
#endif
      rt_task_init(&thread[0], driver, 0, STACK_SIZE, 0, 0, 0);
      for (i = 1; i < NTASKS; i++) {
            rt_task_init(&thread[i], fun, i, STACK_SIZE, 0, 0, 0);
      }
      tick_period = start_rt_timer(nano2count(TICK_PERIOD));
      rt_assign_irq_to_cpu(TIMER_8254_IRQ, TIMER_TO_CPU);
      for (i = 0; i < NTASKS; i++) {
            rt_task_resume(&thread[i]);
      }
      for (i = 0; i < NTASKS; i++) {
            rt_set_runnable_on_cpus(&thread[i], RUN_ON_CPUS);
      }
      return 0;
}


void cleanup_module(void)
{
      int i, cpuid;
      rt_reset_irq_to_sym_mode(TIMER_8254_IRQ);
      stop_rt_timer();
      rt_busy_sleep(1E7);
      for (i = 0; i < NTASKS; i++) {
            rt_task_delete(&thread[i]);
      }
      printk("\n\nCPU USE SUMMARY\n");
      for (cpuid = 0; cpuid < NR_RT_CPUS; cpuid++) {
            printk("# %d -> %d\n", cpuid, cpu_used[cpuid]);
      }
      printk("END OF CPU USE SUMMARY\n\n");
}
```

Message handling API

Send a message to another task and block until it is received:
RT_TASK *rt_send (RT_TASK *destination_task,unsigned int message);

Send a message to another task if and only if it is ready to receive:
```
RT_TASK *rt_send_if(RT_TASK
*destination_task,unsigned int message);
```

Send a message to another task waiting only until ‘time’ is reached before returning if it is not ready to receive:
```
RT_TASK *rt_send_until(RT_TASK
*destination_task,unsigned int message, RTIME time);
```

Send a message to another task waiting only until ‘delay’ has expired before returning if it is not ready to receive:
RT_TASK *rt_send_timed(RT_TASK *destination_task,unsigned int message,RTIME delay);

Receive a message from another task, blocking if necessary until one is sent. If the *sending_task* is 0, messages will be accepted from any task:
```
RT_TASK *rt_receive(RT_TASK *sending_task,unsigned
int *message);
```

Receive a message from another task if, and only if the sender is waiting to send. If the *sending_task* is 0 messages will be accepted from any task:
RT_TASK *rt_receive_if(RT_TASK *sending_task,unsigned int *message);

Receive a message from another task waiting only until ‘time’ is reached before returning if it is not ready to send. If the *sending_task* is 0, messages will be accepted from any task:
RT_TASK *rt_receive_until(RT_TASK *sending_task,unsigned int message,
```
RTIME time);
```

Receive a message from another task waiting only until ‘delay’ has expired before returning if it is not ready to send. If *sending_task* is 0, messages will be accepted from any task:
```
RT_TASK *rt_receive_timed(RT_TASK
*destination_task,unsigned int message,RTIME delay);
```

RPC API

Make a remote procedure call and block, waiting for the reply:
RT_TASK *rt_rpc(RT_TASK *destination_task,unsigned int message,unsigned int *reply);

Make a remote procedure call if (and only if) the destination task is waiting to receive one:
RT_TASK *rt_rpc_if(RT_TASK *destination_task,unsigned int message,unsigned int *reply);

Make a remote procedure call, waiting only until the specified ‘time’ for the destination task to become ready to receive it or for it to reply:
RT_TASK *rt_rpc_until(RT_TASK *destination_task,unsigned int message, unsigned int *reply,RTIME time);

Make a remote procedure call, waiting only until the specified ‘delay’ has expired for the destination task to become ready to receive it or for it to reply:
RT_TASK *rt_rpc_timed(RT_TASK *destination_task,unsigned int message, unsigned int *reply,RTIME delay);

Having received a message, find out whether or not the sending task is waiting for a reply:
int rt_isrpc (RT_TASK *destination_task);

Reply to a sending task:
RT_TASK *rt_return(RT_TASK *destination_task,unsigned int result);

POSIX

Overview

Portable Operating System Interface (plus an ‘X’ to make it sound cool!) is an evolving, growing set of standards designed to promote source-code portability of applications (nirvana being a simple re-compile to move from one operating system to another).

Although the POSIX API is not considered the ultimate in elegance, it does provide both cross-platform portability – with other systems that also implement the POSIX API, and the ability to program real-time tasks in an industry-standard API.

Originally, POSIX.4 (the Real-Time Extensions to POSIX) encompassed: process scheduling, access to time, inter-process communications, (Signals; Messages; Shared Memory and Semaphores) and enhanced I/O. POSIX.4a (actually a completely different POSIX standard) added a Threads implementation into the Real-Time extensions. POSIX.4a has now been renumbered 1003.1c. POSIX.4b adds more Real-Time extensions and has been renumbered 1003.1d. The original POSIX.4 has been renumbered as 1003.1b.

Original POSIX number	Description	New POSIX number
POSIX.4	Real-Time Extensions: e.g.: Semaphores, Priority Scheduling, Process Memory Locking, Shared Memory, Real-time Signal extensions, Clocks and Timers, Messages	1003.1b
POSIX.4a	pthreads	1003.1c
POSIX.4b	Further real-time extensions	1003.1d

RTAI implements section 1003. 1c of the POSIX API, the POSIX threadsor pthreads package, which includes condition variables and mutexes with priority inheritance. This implementation provides hard real-time pthreads where each thread is mapped onto an individual RTAI task. Because all of these threads execute in the same address space, they can concurrently access shared data.

RTAI also implements the POSIX message queues (Pqueues) part of section 1003.1b.

The POSIX specifications allow non-portable extensions (suffixed with _np). RTAI tries to avoid using these because they are non-portable and the thrust of

POSIX itself and the POSIX-compliant extensions to RTAI are to make it more portable.

As of the date of this writing, this POSIX capability works only for real-time tasks *running as kernel modules* – i.e. it has not yet been extended to those tasks running under LXRT.

Features of pthreads

Complicated applications are often most efficiently viewed as a complementary collection of subtasks. In order to make the best use of a computing resource, applications can make use of a computer's ability to perform *multi-tasking*. Traditionally, as applications were divided into multiple tasks the only way to deliver them to the processor was as individual processes. The Threads model takes a process and divides it into two parts:

- One contains resources used across the whole program, such as program instructions and global data. This part is still referred to as the *process*.
- The other contains information related to the execution state, such as a program counter and a stack. This part is referred to as a *thread*.

Not all thread models are the same and the POSIX threads model, as the subject of the rest of this section, for instance, specifies a thread's starting point as a procedure name; contrary to many other popular threads models.

pthreads implement *multi-threading* as a way of performing the many tasks of a program with greater efficiency and speed than would be possible in a serial or multi-process design. pthreads can be created and destroyed and have a set of attributes associated with them that can be set and accessed. These attributes mainly record the Pthread's internal state. pthreads provide two primary methods of synchronizing to ensure that they access shared data in an orderly manner: Mutex variables and Condition variables. A mutex variable acts like a lock protecting shared data, allowing pthreads to synchronize by controlling their access to that data. Condition variables, on the other hand, allow pthreads to synchronize on the value of shared data. Cooperating pthreads wait until the data reaches some particular state or until some particular event occurs. Reader/Writer locks, which are a more complex synchronization tool, are built from a combination of mutexes and condition variables.

A good reference book is: *Programming with POSIX Threads*, David R. Butenhof, Addision Wesley, ISBN 0-201-63392-2. Another good reference book for understanding pthreads is: *Pthreads Programming* by Bradford

Nichols, Dick Buttlar and Jacqueline Proulx Farrell, published by O'Reilly & Associates, Inc.

The RTAI Pthreads module has the following limitations:

- All pthreads are executed in the context of a single task hence there is no support for parent-sibling relationships. Consequently, all API calls that use these relationships, such as pthread_join or pthread_detach, are not implemented
- There are no signal handling calls implemented
- There is currently no support for absolute time, therefore pthread_cond_timedwait is not implemented

Features of Pqueues

- Probably the most general and abstract form of communication is message passing. Most applications can be characterized in terms of messages being passed back and forth between tasks, whether or not the implementation of such applications uses actual message passing.
- A message queue is a priority queue of discrete messages. POSIX message queues offer a certain, basic amount of application access to, and control over, message queue geometry. POSIX message queues were designed as a 'fairly' efficient means of communicating message data between multiple processes. The interface attempts to strike a balance between the different ways people can use message queues (flexibility) and the need for efficiency (simplicity).
- Message queues are *created* or *opened* in much the same way as any UNIX file and like files are 'named' but to distinguish them from files they have their own set of access methods (mq_open, mq_send etc). As each message queue is created, it is sized by declaring the maximum number of message a queue can hold and the maximum size of each message. Like files, message queues are created with access permissions, e.g.; S_IRUSR, which gives read-only permissions to the queue creator, or S_IWOTH, which give write permissions to anyone. Message queues, once created, may be opened by any task that has the appropriate permissions. Once a task has finished using a message queue, it should *close* it. This simply closes that particular task's access to that queue. The queue is not deleted until its creator *unlinks* it.
- Messages are *sent* to a message queue with an associated priority. Similarly, messages are *read* from a queue in priority order. Priority values increment from zero to MQ_PRIO_MAX, which must be at least 32. This

means that each queue has at least 32 levels of message priority. Each priority level can be thought of as an individual message FIFO channel as message order is preserved on each priority level.

- Message queues support blocking unless created with the *mode* argument O_NONBLOCK. In other words, if there is space on a queue a message may be sent to it. However, if the queue is full and the queue was created to support blocking then the sending task will be blocked in its call to *mq_send* until another task reads a message from the queue creating space on it. Similarly, if a queue is empty at the point where *mq_receive* is called and the queue supports blocking then the receiving task will be blocked until a message is placed on the queue.
- Every message queue has the ability to notify one (and only one) process whenever the queue's state changes from empty to not empty. This means that a process does not have to constantly check for messages, instead it can arrange to be poked (notified) when a message arrives.
- Message queue attributes may be queried and if, necessary, altered (assuming the altering task has the correct permissions). These features are useful for interrogating a queue's attributes from a task other than the queue's creator, especially when you don't want to block but the queue has been created with blocking enabled.

Implementation

To use the POSIX capabilities of RTAI:-

insmod rtai.o

insmod rtai_sched.o

insmod rtai_fifos.o

insmod rtai_utils.o

insmod rtai_pthread.o

and, if required,

insmod rtai_pqueues.o

API

As the POSIX API used by RTAI does not use non-portable extensions beyond those in the Linux threads package, no definitions or usage for this API shall be given here. These definitions may be obtained from the standard system “man pages.”

POSIX pthread functions

```
int pthread_ cond_ timedwait( pthread_ cond_ t *cond, pthread_ mutex_ t
*mutex, const struct timespec *abstime);
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
void *(* start_ routine) (void *), void *arg);
void pthread_exit(void *retval);
pthread_t pthread_self(void);
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
int pthread_attr_setdetachstate(pthread_attr_t *attr,
int detachstate);
int pthread_attr_getdetachstate(const pthread_attr_t *attr,
int *detachstate);
int pthread_attr_setschedparam(pthread_attr_t *attr,
const struct sched_param *param);
int pthread_attr_getschedparam(const pthread_attr_t *attr,
struct sched_param *param);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_getschedpolicy(const pthread_attr_t *attr,
int *policy);
```

```
int pthread_attr_setinheritsched(pthread_attr_t *attr, int inherit);

int pthread_attr_getinheritsched(const pthread_attr_t *attr,

int *inherit);

int pthread_attr_setscope(pthread_attr_t *attr, int scope);

int pthread_attr_getscope(const pthread_attr_t *attr, int *scope);

int sched_yield(void);
```

POSIX Mutex Functions

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t
*mutex_attr);

int pthread_mutex_destroy(pthread_mutex_t *mutex);

int pthread_mutexattr_init(pthread_mutexattr_t *attr);

int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);

int pthread_mutexattr_setkind_np(pthread_mutexattr_t *attr, int kind);

int pthread_mutexattr_getkind_np(const pthread_mutexattr_t *attr,

int *kind);

int pthread_setschedparam(pthread_t thread, int policy,const struct
sched_param *param);

int pthread_getschedparam(pthread_t thread, int *policy,struct sched_param
*param);

int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

POSIX Condition Variable Functions

int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *cond_attr);

int pthread_cond_destroy(pthread_cond_t *cond);

int pthread_condattr_init(pthread_condattr_t *attr);

int pthread_condattr_destroy(pthread_condattr_t *attr);

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);

int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);

int pthread_cond_signal(pthread_cond_t *cond);

int pthread_cond_broadcast(pthread_cond_t *cond);

POSIX Queues

mqd_t mq_open(char *mq_name, int oflags, mode_t permissions,

struct mq_attr *mq_attr);

size_t mq_receive(mqd_t mq, char *msg_buffer,

size_t buflen, unsigned int *msgprio);

int mq_send(mqd_t mq, const char *msg, size_t msglen, unsigned int msgprio);

int mq_close(mqd_t mq);

int mq_getattr(mqd_t mq, struct mq_attr *attrbuf);

int mq_setattr(mqd_t mq, const struct mq_attr *new_attrs,

struct mq_attr *old_attrs);

int mq_notify(mqd_t mq,const struct sigevent *notification);

int mq_unlink(mqd_t mq);

readme (/posix/readme)

Implementation of the POSIX pThreads and pQueues API for Real Time Linux.

Authors: Steve Papacharalambous (stevep@zentropix.com)
Trevor Woolven (trevw@zentropix.com)

This is release 0.9 of RTAI pthreads, which implements the Posix 1003.1c Application Programming Interface (API) and release 0.4 of RTAI pqueues, which implements the message queues section of the Posix 1003.1d API.

Please note that this release has been tested as much as possible, however these tests were not exhaustive, especially for SMP architectures. Please report all bugs to the authors.

RTAI pthreads provides hard real-time threads where each thread is a RTAI task. All threads execute in the same address space and hence can work concurrently on shared data.

RTAI pqueues provides kernel-safe message queues.

Note also that these modules can be used interactively.

Requirements

RTAI version 1.1 - available from:

http://www.zentropix.com

http://www.realtimelinux.org

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

http://www.rtai.org

Supported POSIX Calls

See src/README.PTHREADS and src/README.PQUEUES

Limitations

See src/README.PTHREADS and src/README.PQUEUES

The test directory contains a number of test/example programs.

Installation

Install a link from <base dir>/rtai to <base dir>/rtai<curent version>, for example if rtai-1.1 is the current version that is being used and it has been installed in: /usr/src

ln -s /usr/src/rtai-1.1 /usr/src/rtai

This package is included in the standard RTAI distribution, and should already be installed. However if this package has been obtained seperately then it should be installed in the base rtai directory, for example if rtai is installed in:

```
/usr/src/rtai
```

then:

```
cd /usr/src/rtai
```

tar zxvf rtai_posix-0.9.tgz

To build the package:

MAKE SURE that RTAI is set to the correct RTAI installation path FIRST, in the Makefiles. One is located at the top level directory of this package and the other is in the examples subdirectory. The defaulT for this is set to: /usr/src/rtai

```
cd /usr/src/rtai/posix
make clean
make all
```

- make all will also build the example/test programs in the test directory
- make realclean will also clear the test directory

To build the tests/examples:

1. **From the top level directory.**

```
make test
```

2. **From the examples directory.**

```
make clean
make
```

To install the package:

insmod ../modules/rtai.o

insmod ../modules/rtai_sched.o

insmod ../modules/rtai_fifos.o

insmod ./rtai_utils.o

insmod ./rtai_pthread.o

insmod ./rtai_pqueues.o

TODO

See src/README.PTHREADS and src/README.PQUEUES

Acknowledgements

Paolo Mantegazza (mantegazza@aero.polimi.it) for the RTAI package, and for his assistance and advice with this module.

Xavier Leroy (Xavier.Leroy@inria.fr) for his Linuxthreads package which has provided a valuable reference.

Victor Yodaiken (yodaiken@fsmlabs.com) and Micheal Baranbanov (baraban@fsmlabs.com)for the RTLinux project.

readme.pthreads

Implementation of the POSIX pthreads API for Real Time Linux

Author: Steve Papacharalambous (stevep@zentropix.com)

This is release 0.9 of RTAI pthreads, which implements the Posix 1003.1c Application Programming Interface (API).

Please note that this release has been tested as much as possible, however these tests were not exhaustive, especially for SMP architectures.

Please report all bugs to the author.

RTAI pthreads provides hard real-time threads where each thread is a RTAI task. All threads execute in the same address space and hence can work concurrently on shared data.

Supported POSIX Calls

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);

void pthread_exit(void *retval);

pthread_t pthread_self(void);
```

```
int pthread_attr_init(pthread_attr_t *attr);

int pthread_attr_destroy(pthread_attr_t *attr);

int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);

int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate);

int pthread_attr_setschedparam(pthread_attr_t *attr, const struct sched_param
*param);

int pthread_attr_getschedparam(const pthread_attr_t *attr, struct sched_param
*param);

int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);

int pthread_attr_getschedpolicy(const pthread_attr_t *attr, int *policy);

int pthread_attr_setinheritsched(pthread_attr_t *attr, int inherit);

int pthread_attr_getinheritsched(const pthread_attr_t *attr, int *inherit);

int pthread_attr_setscope(pthread_attr_t *attr, int scope);

int pthread_attr_getscope(const pthread_attr_t *attr, int *scope);

int pthread_setschedparam(pthread_t thread, int policy, const struct
sched_param *param);

int pthread_getschedparam(pthread_t thread, int *policy, struct sched_param
*param);

int sched_yield(void);

void clock_gettime( int clockid, struct timespec *current_time);

int pthread_mutex_init(pthread_mutex_t *mutex,

const pthread_mutexattr_t *mutex_attr);

int pthread_mutex_destroy(pthread_mutex_t *mutex);

int pthread_mutexattr_init(pthread_mutexattr_t *attr);

int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);

int pthread_mutexattr_setkind_np(pthread_mutexattr_t *attr, int kind);

int pthread_mutexattr_getkind_np(const pthread_mutexattr_t *attr, int *kind);
```

int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);

int pthread_cond_init(pthread_cond_t *cond,

const pthread_condattr_t *cond_attr);

int pthread_cond_destroy(pthread_cond_t *cond);

int pthread_condattr_init(pthread_condattr_t *attr);

int pthread_condattr_destroy(pthread_condattr_t *attr);

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);

int pthread_cond_timedwait(pthread_cond_t *cond,

pthread_mutex_t *mutex,const struct timespec *abstime);

int pthread_cond_signal(pthread_cond_t *cond);

int pthread_cond_broadcast(pthread_cond_t *cond);

Limitations

1. Parent/Child Relationship

   Currently all the pthreads are executed in the context of a single process, hence there is no parent sibling thread relationship implemented, so all threads are separate entities. Consequently all api calls which use these relationships do nothing, for example: pthread_join, and pthread_detach.

2. Signal Handling.

   No signal handling calls are currently implemented.

3. Absolute time.

There is no support for absolute time so pthread_cond_timedwait is not currently implemented.

TODO

- Parent/sibling related functionality
- Signal handling
- POSIX clocks & timers

Acknowledgements

Paolo Mantegazza (mantegazza@aero.polimi.it) for the RTAI package, and for his assistance and advice with this module.

Xavier Leroy (Xavier.Leroy@inria.fr) for his Linuxthreads package which has provided a valuable reference.

Victor Yodaiken (yodaiken@fsmlabs.com) and Micheal Baranbanov (baraban@fsmlabs.com) for the RTLinux project.

readme.pqueues

Implementation of the POSIX Queues API for Real Time Linux

Author: Trevor Woolven (trevw@zentropix.com)

This is release 0.5 of RTAI pqueues, which implements the Posix 1003.1c Application Programming Interface (API).

Please note that this release has been tested as much as possible, however these tests were not exhaustive, especially for SMP architectures.

Please report all bugs to the author.

RTAI pqueues provides POSIX queues available from RTAI tasks and/or RTAI pThreads.

Supported POSIX Calls

```
//Create or open a message queue
extern mqd_t mq_open(char *mq_name, int oflags, mode_t permissions,
                                        struct mq_attr *mq_attr);
//Receive a message from a message queue
extern size_t mq_receive(mqd_t mq, char *msg_buffer,
                        size_t buflen, unsigned int *msgprio);
//Send a message to a queue
extern int mq_send(mqd_t mq, const char *msg, size_t msglen,
                                        unsigned int msgprio);
//Close a message queue (note that the queue remains in existance!)
extern int mq_close(mqd_t mq);

//Get the attributes of a message queue
extern int mq_getattr(mqd_t mq, struct mq_attr *attrbuf);

//Set a subset of a message queue's attributes
extern int mq_setattr(mqd_t mq, const struct mq_attr *new_attrs,
                        struct mq_attr *old_attrs);
//Register a request to be notified whenever a message arrives on an empty
// queue
extern int mq_notify(mqd_t mq, const struct sigevent *notification);

//Destroy a message queue
extern int mq_unlink(mqd_t mq);
```

Limitations

1. Signal Handling.

   Currently, pqueue 'notification' (of a write into an empty pqueue) is not implemented.

2. Closing/Unlinking

   This must be done while the task(s) is alive as it relies on being able to get the task's queue access data.

3. Configuration

   There are a number of configuration parameters, such as the maximum number of allowed message queues, #defined in rtai_utils.h You must change these to suit your application.

4. Raw RTAI Task Interface.

If the native RTAI tasking interface is used (instead of the pthreads interface), then the following rules MUST be followed:

- A call to init_z_apps must be made after every call to rt_task_init.
- A call to free_z_apps must be made after every call to rt_task_delete.

readme.utils

Common files for Real Time Linux Applications

Copyright: Zentropix LLC, 1999
Authors: Steve Papacharalambous (stevep@zentropix.com)
Trevor Woolven (trevw@zentropix.com)

This directory contains files common to some or all, of the RTAI applications controlled by the RTAI 'project'.

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of

MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA.

Requirements

RTAI version 1.1 - available from:

http://www.zentropix.com

http://www.realtimelinux.org

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

The examples directory contains various test programs

Acknowledgements

Paolo Mantegazza (mantegazza@aero.polimi.it) for the RTAI package, and for his assistance and advice with this module.

Xavier Leroy (Xavier.Leroy@inria.fr) for his Linuxthreads package which has provided a valuable reference.

Victor Yodaiken (yodaiken@fsmlabs.com) and Micheal Baranbanov (baraban@fsmlabs.com) for the RTLinux project.

chapter 4

Advanced RTAI Features

LXRT

Overview

LXRT provides a User-Space interface to the facilities and features of RTAI. It provides a symmetric API that may be used by both real-time RTAI tasks and Linux processes. LXRT is unique to RTAI and is one of its most useful features.

LXRT allows the user to develop a real-time task using RTAI's API from user space. The principal advantage with this approach, is that the task can be developed as a 'soft' real-time task under the memory protection umbrella of standard Linux, whereas tasks running within the kernel have access to unprotected memory space and could over-write critical sections of kernel memory. In addition, while in User Space one has the full range of Linux System calls available too but one has to be careful when switching to kernel space, as Linux System calls are not currently supported from 'hard' real-time LXRT.

Once the developer is satisfied with the functionality of the user-space 'soft' real-time task, it can be transitioned to a 'hard' real-time task by compiling as a kernel module, removing the LXRT module (actually this is not *required* but is recommended for the smallest memory footprint), and finally by inserting the task module.

An additional feature of LXRT is that it allows the dynamic switching of tasks between the hard/soft real-time modes from *within your application*. This allows you to do things such as creating one real-time task from another.

LXRT Versions

In the beginning there was LXRT, which provided the RTAI API to user space processes. LXRT-Informed added to LXRT the ability to recover after the crash of a Linux process with a real-time LXRT component. LXRT-Extended added to LXRT, the ability to run “hard” real-time processes from user space. The bulk of this document describes LXRT-Extended.

How It Works

At the top-level, LXRT simply provides the same set of RTAI API calls available for RTAI applications, in User Space. LXRT enhances its 'soft' real-time performance by requiring the programmer to change the Linux scheduler's policy for the LXRT process from scheduling policy from SCHED_OTHER to SCHED_FIFO. In addition, it is a requirement that the memory for a process be locked-in (free from paging) by using the mlockall system call.

SCHED_OTHER is the standard Linux default used by most processes, SCHED_FIFO (and SCHED_RR) are intended for special, time-critical applications that need precise control over the way in which runnable processes are selected for execution.

Processes scheduled with SCHED_OTHER have a static priority of 0. The scheduler selects which process to actually run from the list of runnable processes based on their ‘nice’ level. This is done to achieve ‘fair’ allocation of the CPU to each process. As you can imagine, this is by no means optimal for processes with execution deadlines to meet.

Processes scheduled with SCHED_FIFO are assigned static priorities in the range 1 to 99, which means that when a SCHED_FIFO process becomes runnable it will immediately preempt a running SCHED_OTHER process or a SCHED_FIFO process of lower priority. A FIFO (first in, first out) policy is applied to processes of the same priority. Preempted SCHED_FIFO processes remain at the head of their priority queue and resume execution again once all higher-priority processes become blocked. Generally, when a SCHED_FIFO process becomes runnable it is placed at the end of the list for its particular priority.

SCHED_RR is a simple enhancement to SCHED_FIFO, whereby each process is only allowed to run for a maximum time quantum before being re-scheduled to the back of its priority list. LXRT does not generally use SCHED_RR.

To summarize, LXRT requires the use of SCHED_FIFO scheduling policy with statically assigned process priorities to achieve ‘soft’ real-time performance whilst in User Space. This is a great improvement on normal Linux process performance but is still subject to missing deadlines when either interrupts or real-time activity consume the CPU. However, LXRT provides the facility to switch an application process to real-time, where it becomes scheduled by the RTAI scheduler not the Linux scheduler. The way LXRT does this is by creating a real-time agent task, which executes the real-time services (such as being scheduled to run) on behalf of the LXRT process, communicating back to the LXRT process once the real-time service has completed.

Performance

While the kernel module real-time tasks continue to provide the very best performance, where context switch times are typically under 40µs (depending on the hardware), LXRT tasks take a marginal hit in context switch time with typical context switch times under 100µs.

LXRT Summary

Summarized below are some of the more important features of LXRT:

LXRT allows tasks to execute as real-time tasks from standard user space:

A. These tasks execute under the Linux memory protection scheme. This provides protection against system crashes during the development and debugging phases of a project.

B. Allows a system to be divided into hard real-time and soft real-time components more easily as the LXRT modules will execute at a higher priority than normal Linux processes, and have finer scheduling granularity.

C. Tasks can be debugged using standard Linux user-space debug tools.

D. When a task has completed its preliminary debug, it can be moved into the kernel space.

E. Tasks make use of the standard RTAI API, which makes it much simpler to move tasks between the hard and soft real-time domains.

General benefits of LXRT include:

A. Once a root user has installed the required modules, the LXRT real-time task technique has an additional advantage since it is usable by non-root users (via an API call). Thus it can be adopted for training purposes where you don't want inexperienced users to have super-user privileges on the development/training machine.

B. Since the real-time tasks are no longer implemented as kernel modules, they no longer carry kernel dependencies. Thus, a binary task is easily transportable among different machines each running perhaps different kernel versions. This allows the developer to easily deploy a binary only real-time task, thus eliminating the need to provide the source code to the end user.

Implementation

Under LXRT, the real-time task is implemented as a user-space task but it is actually scheduled by the real-time Linux scheduler when moved to hard real-time. This is simply achieved by, inserting the LXRT module and using the LXRT API within your user space task.

Insert the Module

```
insmod rtai
insmod rtai_sched
insmod lxrt
   ./run_your_process
```

API

In the main, the LXRT API is identical to RTAI's, with the following exceptions:

LXRT Hard Real Time Non-Root functions

```
print_to_screen(const char *format, ...)
```

Safely prints information and diagnostic messages from hard real-time user space modules to the screen.

```
void rt_make_hard_real_time(void)
```

Translates a soft real-time Linux process into a hard real-time LXRT process.

```
void rt_make_soft_real_time(void)
```

Returns a hard real-time LXRT process to soft real-time Linux process.

```
rt_allow_nonroot_hrt(void)
```

Allow a non-root user to transition an LXRT process to hard real-time, lock process memory in ram and perform IO operations from user space.

LXRT Agent Task creation

Create and Initialize an LXRT real-time agent task for the current process.

```
RT_TASK *rt_task_init(  int taskname,
                        int priority,
                        int stack_size,
                        int max_msg_size );
```

The name can be created from up to 6 alphanumeric characters using the name2num macro, e.g.:

```
   int taskname = nam2num(“MTASK1”);
```

To delete the real-time agent task:

```
int rt_task_delete(RT_TASK *task)
```

RTAI features unsupported in LXRT:

The LXRT API is not a complete one-for-one copy of the RTAI API. There are some functions that are incompatible with the architecture of LXRT and some, like FIFOs, which have become obsolete due to considerable functionality improvements.

The following is an attempt to capture the RTAI functions not currently supported under the existing LXRT API:

FIFOs

Shared Memory

*_irq_*

*_srq_*

*_intr_*

rt_task_init_cpuid

send_ipi*

rt_request_timer

rt_mount_rtai

rt_umount_rtai

rt_sched_type

start_rt_timer_ns

rt_whoami

rt_task_make_periodic_relative_ns

next_period

rt_get_base_linux_task

EXAMPLE:

The following two files work together to illustrate LXRT. They form the LXRT-Informed client-server example taken from the lxrt-informed directory.

```
// FILE: Server.c

#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <sched.h>

#include <rtai.h>
#include <rtai_sched.h>
```

```
#include "../rtai_lxrt.h"

#define SLEEP_LOOPS     2
#define DELAY           5E4
#define MSG_DELAY 1E9
#define MSG_LOOPS 12

main()
{
      unsigned long mtsk_name = nam2num("SRV");
        unsigned long btsk_name = nam2num("CLT");
      int msg;
      RT_TASK *btask, *mtsk, *rcvd_from;
      int i, dist[10000], pid, count, zeroi;
      struct sched_param mysched;
        unsigned long sem_name  = nam2num("SEM");
        SEM *sem;

        memset( dist, 0, sizeof(dist));

      mysched.sched_priority = 99;

      if( sched_setscheduler( 0, SCHED_FIFO, &mysched ) == -1 ) {
            puts(" ERROR IN SETTING THE SCHEDULER UP");
            perror( "errno" );
            exit(1);
            }

      mlockall(MCL_CURRENT | MCL_FUTURE);

      if (!(mtsk = rt_task_init(mtsk_name, 0, 0, 0))) {
            printf("CANNOT INIT SRV TASK\n");
            exit(2);
      }
      printf("SRV TASK INIT: name = %lx, address = %p(%p).\n",
mtsk_name, mtsk, this_rt_task);

      printf("SRV pid %d TASK MAKES ITSELF PERIODIC WITH A PERIOD OF 1
sec\n", getpid());
```

```
        start_rt_timer(nano2count(1E5));
      rt_task_make_periodic(mtsk, rt_get_time(), nano2count(1E9));


            if (!(sem = rt_sem_init(sem_name, 0))) {
               printf("CANNOT CREATE SEMAPHORE %lx\n", sem_name);
               exit(1);
        }
        printf("SRV TASK CREATES SEM: name = %lx, address = %p.\n",
sem_name, sem);

      printf("SRV TASK CREATES CLT TASK\n");
      pid = fork();
      if (!pid) {
            execl("./client", "./client", NULL);
            }

      rt_sleep(nano2count(1*1E9));
      zeroi = 0 ;
        count = MSG_LOOPS;
        while(count--) {

            rcvd_from = rt_receive(rt_get_adr(btsk_name), (void
*)&msg);
            printf("SRV RECEIVED MESSAGE %d FROM CLT\n", msg);

            // Now let's blow up while not received blocked on LXRT
            if(count == 6) count /= zeroi;

            rt_return(rcvd_from, ++msg);
            }

//    printf("SRV TASK DELETES ITSELF\n");
      rt_task_delete(mtsk);
        rt_sem_delete(sem);
      stop_rt_timer();
//    printf("END SRV TASK\n");

exit(0);
```

```
}


// FILE: Client.c

#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <sched.h>

#include <rtai.h>
#include <rtai_sched.h>
#include "../rtai_lxrt.h"

#define MSG_DELAY       1E9
#define MSG_LOOPS 12

main()
{
      unsigned long mtsk_name = nam2num("SRV");
        unsigned long btsk_name = nam2num("CLT");
      int count, msg, rep;
      RT_TASK *mtsk, *btsk, *err;
      struct sched_param mysched;

      mysched.sched_priority = 99;

      if( sched_setscheduler( 0, SCHED_FIFO, &mysched ) == -1 ) {
      puts(" ERROR IN SETTING THE SCHEDULER UP");
      perror( "errno" );
      exit( 0 );
      }

printf("CLT pid %d\n", getpid());

      mlockall(MCL_CURRENT | MCL_FUTURE);

      if (!(btsk = rt_task_init(btsk_name, 0, 0, 0))) {
            printf("CANNOT INIT CLIENT TASK\n");
```

```
            exit(1);
      }

        rt_task_make_periodic(btsk, rt_get_time(), nano2count(1E9));

      printf("CLT TASK: name = %lx, address = %p(%p).\n", btsk_name,
btsk, this_rt_task);

      if (!(mtsk = rt_get_adr(nam2num("SRV")))) {
            printf("CANNOT FIND SRV TASK\n");
            exit(1);
      }

      msg = 0;
      count = MSG_LOOPS;
      while( count-- ) {
//          printf("CLT TASK SENDS MESSAGE %d TO SRV TASK\n", msg );
            err = rt_rpc(mtsk, msg, &rep);
            if( err != mtsk ) {
                  printf("CLT: rt_rpc() failed\n" );
                  break;
                  }
//           printf("CLT TASK GETS REPLY    %d FROM SRV TASK\n", rep );
             rt_sleep(nano2count(1E9));
            msg = rep ;
            }

      printf("CLT TASK DELETES ITSELF\n");
      rt_task_delete(btsk);
}
```

The following example is the LXRT, master_buddy example master_proc.c:

```
/*
FILE: Master_proc.c

COPYRIGHT (C) 1999  Paolo Mantegazza (mantegazza@aero.polimi.it)
```

```
This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307
USA.
*/

#include <stdio.h>
#include <sys/mman.h>
#include <sched.h>

#include <rtai.h>
#include <rtai_sched.h>
#include "../rtai_lxrt.h"

#define PERIODIC_LOOPS 100

#define SLEEP_LOOPS 100

#define MBX_LOOPS 3000

#define DELAY 5E4

#define MSG_DELAY 1E9

main()
{
      unsigned long mtsk_name = nam2num("MTSK");
      unsigned long btsk_name = nam2num("BTSK");
      unsigned long sem_name  = nam2num("SEM");
```

```
      unsigned long smbx_name  = nam2num("SMBX");
      unsigned long rmbx_name  = nam2num("RMBX");
      unsigned long msg;

      long long mbx_msg;
      long long llmsg = 0xaaaaaaaaaaaaaaaaLL;

      RT_TASK *mtsk, *btsk, *rcvd_from;
      SEM *sem;
      MBX *smbx, *rmbx;
      RTIME time;
      int pid, count;

      struct sched_param mysched;

      mysched.sched_priority = 99;

      if( sched_setscheduler( 0, SCHED_FIFO, &mysched ) == -1 ) {
      puts(" ERROR IN SETTING THE SCHEDULER UP");
      perror( "errno" );
      exit( 0 );
      }

      mlockall(MCL_CURRENT | MCL_FUTURE);

      if (!(mtsk = rt_task_init(mtsk_name, 0, 0, 0))) {
            printf("CANNOT INIT MASTER TASK\n");
            exit(1);
      }
      printf("MASTER TASK INIT: name = %lx, address = %p.\n",
mtsk_name, mtsk);

      printf("MASTER TASK STARTS THE ONESHOT TIMER\n");
      rt_set_oneshot_mode();
      start_rt_timer(nano2count(1E7));

      printf("MASTER TASK MAKES ITSELF PERIODIC WITH A PERIOD OF 1
ms\n");
      rt_task_make_periodic(mtsk, rt_get_time(), nano2count(1E6));
```

```
	rt_sleep(nano2count(1E9));

	count = PERIODIC_LOOPS;
	printf("MASTER TASK LOOPS ON WAIT_PERIOD FOR %d PERIODS\n",
count);
	while(count--) {
		printf("PERIOD %d\n", count);
		rt_task_wait_period();
	}

	count = SLEEP_LOOPS;
	printf("MASTER TASK LOOPS ON SLEEP 0.1 s FOR %d PERIODS\n",
count);
	while(count--) {
		printf("SLEEPING %d\n", count);
		rt_sleep(nano2count(DELAY));
	}
	printf("MASTER TASK YIELDS ITSELF\n");
	rt_task_yield();

	printf("MASTER TASK CREATES BUDDY TASK\n");
	pid = fork();
	if (!pid) {
		execl("./buddy_proc", "./buddy_proc", NULL);
	}

	printf("MASTER TASK SUSPENDS ITSELF, TO BE RESUMED BY BUDDY
TASK\n");
	rt_task_suspend(mtsk);

	if (!(sem = rt_sem_init(sem_name, 0))) {
		printf("CANNOT CREATE SEMAPHORE %lx\n", sem_name);
		exit(1);
	}
	printf("MASTER TASK CREATES SEM: name = %lx, address = %p.\n",
sem_name, sem);

	printf("MASTER TASK WAIT_IF ON SEM\n");
	rt_sem_wait_if(sem);
```

```
      printf("MASTER STEP BLOCKS WAITING ON SEM\n");
      rt_sem_wait(sem);

      printf("MASTER TASK SIGNALLED BY BUDDY TASK WAKES UP AND BLOCKS
WAIT TIMED 1 s ON SEM\n");
      rt_sem_wait_timed(sem, nano2count(1E9));

      printf("MASTER TASK DELETES SEM\n");
      rt_sem_delete(sem);

      printf("MASTER TASK BLOCKS RECEIVING FROM ANY\n");
      rcvd_from = rt_receive(0, (void *)&msg);
      printf("MASTER TASK RECEIVED MESSAGE %lx FROM BUDDY TASK\n",
msg);

      printf("MASTER TASK RPCS TO BUDDY TASK THE MESSAGE %lx\n",
0xabcdef);
      rcvd_from = rt_rpc(rcvd_from, 0xabcdef, (void *)&msg);
      printf("MASTER TASK RECEIVED THE MESSAGE %lx RETURNED BY BUDDY
TASK\n", msg);
//exit(1);
      if (!(smbx = rt_mbx_init(smbx_name, 1))) {
            printf("CANNOT CREATE MAILBOX\n", smbx_name);
            exit(1);
      }
      if (!(rmbx = rt_mbx_init(rmbx_name, 1))) {
            printf("CANNOT CREATE MAILBOX\n", rmbx_name);
            exit(1);
      }
      printf("MASTER TASK CREATED TWO MAILBOXES %p %p %p %p \n", smbx,
rmbx, &mtsk_name, &msg);

      count = MBX_LOOPS;
      while(count--) {
            rt_mbx_send(smbx, &llmsg, sizeof(llmsg));
            printf("%d MASTER TASK SENDS THE MESSAGE %llx MBX\n",
count, llmsg);
            mbx_msg = 0;
```

```
            rt_mbx_receive_timed(rmbx, &mbx_msg, sizeof(mbx_msg),
nano2count(MSG_DELAY));
            printf("%d MASTER TASK RECEIVED THE MESSAGE %llx FROM
MBX\n", count, mbx_msg);
            rt_sleep(nano2count(DELAY));
      }

      printf("MASTER TASK SENDS THE MESSAGE %lx TO BUDDY TO ALLOW ITS
END\n", 0xeeeeeeee);
      rt_send(rcvd_from, 0xeeeeeeee);

      printf("MASTER TASK WAITS FOR BUDDY TASK END\n");
      while (rt_get_adr(btsk_name)) {
            rt_sleep(nano2count(1E9));
      }
      printf("MASTER TASK STOPS THE PERIODIC TIMER\n");
      stop_rt_timer();

      printf("MASTER TASK DELETES MAILBOX %p\n", smbx);
      rt_mbx_delete(smbx);
      printf("MASTER TASK DELETES MAILBOX %p\n", rmbx);
      rt_mbx_delete(rmbx);

      printf("MASTER TASK DELETES ITSELF\n");
      rt_task_delete(mtsk);

      printf("END MASTER TASK\n");
}
```

RTAI includes a large number of readme files and examples. As described early in this document, it is not our intention to re-write those sections which are described in detail by the readme files. Thus, below you will find copies of the RTAI v1.4 readme files.

README.LXRT

The all stuff here, and shared memory as well, evolved from the laziness of some people, myself included, in reading Linux manuals for SYSTEM V services as well as from the annoyance of either using timers or select artifact to sleep for less than 1 s.

At some point we decided that it takes less to program it in RTAI than to learn it from the manuals. It was not exactly so, but not so bad either. Thus this directory contains an implementation of services to make available any of the RTAI schedulers functions to Linux processes, so that a fully symmetric implementation of real time services is possible.

To state it more clearly, that means that you can share memory, send messages, use semaphores and timings: Linux<->Linux, Linux<->RTAI and, naturally, RTAI<->RTAI.

As already done for the shared memory the function calls for Linux processes are inlined in the file "rtai_lxrt.h". This approach has been preferred to a library since it is simpler, more effective, the calls are short and simple so that, even if it is likely that there can be more than just a few per process, they could never be charged of making codes too bigger.

At this point it is important to remark that a Linux process wanting to access "lxrt" services, i.e. the real time RTAI schedulers, must create its real time buddy, also called proxy, by using rt_task_init as explained below.

Then to exploit it you must just use the function prototypes available in rtai_sched.h, and documented in the doc files.

An exception to the previous rule are the calls to init tasks, semaphores and mailboxes.

They are, formal arguments names self explain themselves:

- LX_TASK *rt_task_init(unsigned int name, int prio, int stack_size, int max_msg_size), which has less arguments and returns the pointer to the task that is to be used in related calls. The stack and max message size can be zero, in which case the default internal values are used. The assignment of a different value is required only if you want to use task signal functions. In such a case note that these signal functions are intended to catch asyncrounous event in kernel space and, as such, must be programmed into a companion module and interface to their parent Linux process through the available services.

Keep an eye on the default stack (512) and message (256) sizes as they seem to be acceptable, but this API has not been used extensively with complex interrupt service routines. Since the latter are served on the task being interrupted, and more than one can pile up on the same stack, it can be possible that a larger stack is required. In such a case either recompile lxrt.c with macros STACK_SIZE and MSG_SIZE set appropriately, or explicitly assign larger values at your tasks inits. Note that while the stack size can be critical the message size will not. In fact the module reassigns it, appropriately sized, whenever it is needed. The cost is a kmalloc with GFP_KERNEL that can

block, but within the Linux environment. Note also that the max message size if for a buffer to be used to copy whatever message, either mailbox or intertask, from user to kernel space, as messages are not necessarely copied immediatly, and has nothing to do directly with what you are doing.

- SEM *rt_sem_init(unsigned long name, int initial_count), which returns a pointer to the semaphore to be used in related calls.

- MBX *rt_mbx_init(unsigned long name, int buf_size), which returns a pointer to the semaphore to be used in related calls.

Note that the returned pointers cannot be used directly, they are for kernel space data, but just passed as arguments when needed.

For interfacing to tasks, semaphores and mailboxes created by modules you must use:

- void *rt_get_adr(unsigned long name), which return the pointer to the object of name "name". Usually you need not type the returned value since it must just be used in the related calls;

- unsigned long rt_get_name(void *adr), to get the name attached to the object at address "adr".

Modules can get to objects created by Linux processes by using the same calls above, while to make their semaphores and tasks accessible to Linux processes they must use:

- int rt_register(unsigned long name, void *adr), to register their name and address. The value returned is positive for a succesfull registration and zero if the registration failed.

It is important that modules deregister any register objects at the end of the job or when they are deleted, by using either:

- int rt_drg_on_adr(void *adr)
or
- int rt_drg_on_name(unsigned long name)

Again a return value > 0 means success while a zero failure.

Linux processes buddies, and semaphores and mailboxes as well, need not to register/deregister as that is already done at their init and delete respectively, so no related function is provided.

The functions rt_get_name and rt_get_adr can be used to verify the existence of any object, atomically, making it possible a sinchronized and safe beginning/ ending of a cooperative work.

It is important that the register list is a static one whose size is determined by the macro MAX_SLOTS in lxrt.c. Its value is now 128; change it according to your needs, but remember that IT MUST BE A POWER OF 2.

For a better use of the above features you are strongly recomended to use the Linux POSIX soft real time options for locking all of your process in memory and for scheduling it in Linux. You should refer to "man mlock" and "man sched_setscheduler" to see how it is done. They require root permission, but, thank to the extended LXRT version, you can can use: rt_allow_nonroot_hrt() to make those Linux POSIX APIs available to any user. See README.EXTENDED_LXRT in this directory.

Another feature worth considering is to use Linux pthreads_create to create threads, in fact Linux processes, from functions within a file. It can ease prting to task modules in kernel space.

We think that what you can do with this stuff can be very usefull as it provides an easy to use unified environment for any real time application. Such an environment can be of help in the initial development phase of real time applications, as it could be carried out in user space, with the advantage that system crashes could be reduced drastically, or be less dangerous, for your hard disk at least. Once more see also README.EXTENDED_LXRT for the extension of these features to hard real time processes in user space.

I'll hope that Steve can do something similar for POSIX soon. It is likely that he can just copy what is needed almost as it is. So you'll end with "yet another PTHREADS in Linux" that will allow to do POSIX compliant soft-firm-hard real time within a unified environment.

So far we have not measured any performance but we expect results similar to those of making system calls. Similar do not mean equals. Take into account that here, except for the simplest functions that allow a direct call, you

have always to switch real time tasks to get to the RTAI scheduler services. It should be remarked that you must install a SIGINT handler if you want to safely terminate you LXRT processes, cleaning up any RTAI resouce they use, after Ctrl-C. We remind that what you find in directory lxrt is the final development version, the related production version is in lxrt-informed. It may happen that in lxrt you can find features not yet ported in lxrt-informed. It will likely be so for a very short time. So take care of abormal terminations or wait for help from lxrt-informed.

There are also two test cases:

- This test runs through most of the available services between two Linux processes (directory master_buddy);

- This test uses a periodic real time task agent that, after having done its work, actually nothing, relay data sent to himi, through shared memory by a sender Linux process, to a receiver Linux processes. The sender is synchronized to the real time agent task by using semaphores, while the receiver uses intertask messages (directory rt_agent).

As shown above all LXRT objects are identified by an unsigned long. To use alphanumeric mnemonic terms a couple of very simple functions are available to translate a SIX CHARACTERs string into and unsigned long, both in kernel and user space. They are:

- unsigned long nam2num(char *name);

- void num2nam(unsigned long num, char *name);

So if you like to use them you can do:

adr = rtai_malloc(nam2num("myNAME"), size);
or
rtai_free(num2nam("myNAME"), adr);

Allowed characters are:
- English letters (no difference between upper and lower case);
- 10 digits;
- underscore (_) and another character of your choice. The latter will be always converted back to a $ by num2nam.

It is important to remark that, even if what is found under this directory can be used for any application, it is strongly raccomended to use lxrt-informed for production work. See the docs file there for an explanation.

See also "lxrt-informed" for other docs.

README.EXTENDED_LXRT

Extended LXRT - A new concept

We try to provide hard real time services in user space, also for normal, i.e. non root, users. We think that it will not be as good a performer as kernel space real time task modules, but a few microsecs more latency can be acceptable for many applications. Many users will be glad with it for itself. At the very least,

it will be usefull in easing development and many other things: training, teaching and so on. It is wholly along the basic LXRT concept so we have seen it as an extension to LXRT.

To get hard real time in user space you need a fully preemptable kernel. The question, within RTAI "philosphy", is how to get full preemption with minimum changes, possibly none, to the kernel source.

The solution calls for a compromise. We propose to accept that a hard real time process does no Linux context kernel operations leading to a task switch. In than sense, it is better to speak of a "user space kernel module", and we will use the two terms interchangeably.

The approach is similar to what one of us did when he was using QNX: he always mated a hard real time tasks with a buddy for any I/O operation that could lead to excessive delays. In fact, even within such a fully preemptable kernel, I/Os could lead to deadlines misses under heavy hard real time I/O load from many hard real time tasks. Many examples in this distribution, i.e.: clocks, latency calibration and sound, show you a clear picture of how easy it is to use kernel services by mating to a buddy server process, without any problem.

So, at least on the base of our modest experience, that is not an unbearable constraint. Since RTAI has many good intertask services, we do not see any problem in using the same approach again, especially in view of with what Pierre has done, is doing and will do, to make it the "informed" way. It is nonetheless possible that such a constraint will be somewhat lifted as development proceed. Moreover the user space approach does not forbid you to do it in kernel space, if it is eventally needed. In fact it is not seen as an alternative to doing it in the kernel, but simply as a way of giving you more opportunities, at least during the development phase.

Taking into account that the present solution is just at the beginning of its development, we see a lot of space for making it better.

How is that possible?

We think that what you'll have here shows that it can work, even if it can be improved. The idea is to keep soft interrupts disabled for hard real time user space modules. This way, kernel module hard real time tasks and hard real time interrupts can preempt user space modules, but user space modules cannot be preempted neither by Linux hard interrupt nor by Linux processes. Linux hardware interrupt are pended as usual for service when RTAI's real time tasks (both in the kernel and user space) are idle.

How does it work?

Hard real time user space modules are just normal Linux processes that mate to a special buddy hard real time kernel task module, as done under LXRT already. They must be POSIX real time Linux processes locked into memory. To distinguish them from usual LXRT firm real time processes the user simply calls rt_make_hard_real_time(), whereas by using rt_make_soft_real_time() he can return to standard Linux task switching. Note that some of the required features, e.g POSIX real time under Linux, require root permission. However by using the function rt_allow_nonroot_hrt() you are allowed to: make a process POSIX real time, lock the memory and do IO operations, as a normal non root user. It is nonetheless necessary that the superuser "insmod"s the required modules (rtai, rtai_sched and lxrt).

The call to rt_make_hard_real_time allows to take a normal process out from the Linux running queue by calling schedule() after having queued the task to a bottom handler. When the bh runs, the task is scheduled as a hard real time module by lxrt_schedule(), and Linux will not know of it, while having it still in its process list.

Lxrt_schedule() is also set as the signal function to be called when returning to the Linux context from a hard real time kernel space schedule, thus ensuring preemption in any case.

Lxrt_schedule() clear the soft interrupt flags and mimics the Linux schedule() function, with scheduling policy SCHED_FIFO, even from within interrupts.

To return to soft real time, rt_make_soft_real_time() does the opposite.

What it currently does:

There are some (not so) simple test processes that runs periodically and on which scheduling latency is measured. No doubt that it does something different as by running the same tasks under the same load with plain LXRT the latency goes as high as Linux 10 ms tick, compared to a few tens of usecs under user space modules (preliminary rough measures). Note that within this new context it is likely that you can use also Linux pthreads both for soft and hard real time. In fact pthreads are normal user processes in disguise, Xavier made a choice, i.e. pthreads as cloned processes, that is good also for LXRT. Other examples show interacting tasks at work, while the sound task gives an idea of IOs from user space.

The experience gathered so far indicates that, despite the availability of more processing power, under SMP the latency for the same background load can be double/tripled with respectg to UP. That is likely due to cache trashing caused by processe switches and seems not to depend on the RTAI MP scheduler you are using. So it makes a larger jitter difference, with respect to working in kernel space, using hard real time processes under SMP than under UP. In fact

under UP the jitter is roughly the same weather you are using user or kernel space modules.

What it currently does not very quickly:

Lxrt_schedule() can schedule in and out plain Linux processes, but to do it safely that must happen within Linux idle tasks. Clearly when one tests under heavy load the starting and ending of hard real time mode can be somewhat sluggish. In any case problems are just in starting and ending, once user space modules are in place they are fine. The matter has been somewhat improved by forcing the scheduling weight of the idle task, just four lines added/modified within the kernel. We know that there can be other ways of doing it, but all what we could conceive is likely to require heavy kernel modifications. Once more we recall that all our "philosophy" is to deplete the kernel with the slightest changes possible to it, better if none.

Note that within lxrt.c we trapped the kernel sys call and interrupt enabling to be sure that they are not called within hard real time user space modules. The same thing is possible, done directly in rtai.c, for all the reserved Linux traps, but no alternative handler has been implemented yet.

The new additions to lxrt:

- changed rtai_lxrt_handler to avoid ret_from_intr if returning from within a hard real time process;

- added macros my_switch_to(prev,next,last), loaddebug and function

__switch_to, all copied from Linux;

- added lxrt_schedule to schedule hard real time user space tasks among themselves and to and from the Linux context, with soft flags disabled (__cli()), a lot of new data needed are found just above it, the name should self explain them;

- added function lxrt_do_steal to be run from the bh timer to schedule a new hard real time process;

- added the pointer rthal_enint to save the trapped trap rtahl.enint in order to diagnose enable from within rt user space modules;

- added lxrt_enint to actually do the above trapping;

- added lxrt_sigfun to lxrt_schedule when getting back to Linux from the rtai schedulers;

- added steal_from_linux to make a Linux process a user space hard real time module;

- added give_back_to_linux to return a user space module to the Linux processes;

- added linux_syscall_handler to save the trapped Linux sys handler;

- added lxrt_linux_syscall_handler to diagnose calls to sys from hard real time processes;

- print_to_screen to allow a safe printing of diagnosting messages from within user space modules working in hard real time mode.

User functions:

- print_to_screen(const char *format, ...): to safely print information and diagnostic messages in hard real time user space modules;

- void rt_make_soft_real_time(void): to return a hard real time user space process to soft Linux POSIX real time;

- void rt_make_hard_real_time(void): to make a soft Linux POSIX real time process a hard real time LXRT process;

- rt_allow_nonroot_hrt(void): to allow a non root user the make a process Linux POSIX real time, lock process memory in ram and carry out IO operations from user space.

Tests:

There is a wealth of examples to show extended lxrt operations, both in soft and hard real time mode. They can be useful also in giving you some clues for i your applications.

Tests list:

- single task (directory one);

- two tasks (directory two);

- many tasks (directory many);

- many tasks (directory forked);

- many pthreads (directory threads);

- latency calibration (directory latency_calibration);

- sound test (directory sound);

- digital clock with semaphores (directory sem_clock);

- digital clock with messages (directory msg_clock).

- task resumed from an interrupt handler (directory resumefromintr).

The possibility of using pthread_create to generate Linux processes is very usefull since it allows a task layout that is close to the structure of modules. That could make it easier the translation to kernel modules for maximum performances. Also to be remarked is the possibility of resuming user space modules directly from interrupt handlers, see example reseumefromint.

If you want to check the jitter while one of the clocks or the sound example are running, you should enter the latency_calibration directory under another screen and type "./rt_process 1 &" followed by "./check". Try it varying Linux load. Be carefull, you must end it before closing the clocks/sound tests, see a more detailed comment within README in latency_calibration directory.

Have a look at the README files in each directory for more informations.

It is important to remark that what is found under this directory can be used for any application but it is intended mainly for devlopment work. It will be soon ported to lxrt-informed for a safer production use. Thus it is remarked that you must install a SIGINT handler if you want to safely terminate you LXRT processes, cleaning up any RTAI resouce they use, after Ctrl-C. Some examples show how it can be done. We remind once more that what you find in directory lxrt is the final development version, the related production version is in lxrt-informed. It may happen that under this directory you can find features not yet ported in lxrt-informed. It will likely be so for a very short time. So take care of abormal terminations yourself or wait for help from lxrt-informed.

Contributed by: Pierre Cloutier, Paolo Mantegazza, Steve Papacharalambous.

LXRT-INFORMED.FAQ

How does LXRT works?

This onepager is an attempt to explain conceptually how LXRT works. It does not try to get into the nitty gritty details of the implementation but it tries to explain how the context of execution switches between Linux and RTAI.

But first, what are we trying to do?

LXRT provides a family of real time scheduler services that can be used by both real time RTAI tasks and Linux tasks. To keep things simple for the programmer the implementation is fully symmetric. In other words, the same function calls are used in both the kernel and user space.

What are those real time scheduler services?

RTAI provides the standard services like resume, yield, suspend, make periodic, wait until etc. You will also find semaphores, mail boxes, remote procedure calls, send and receive primitives integrated into the state machine of the real time scheduler. Typically, the IPC function calls support:

- Blocking until the transaction occurs.
- Returning immediately if the other end is not ready.
- Blocking for the transaction until a timeout occurs.

How do I setup my Linux program for LXRT?

You call rt_task_init(name, ...). The call differs from the real time counterpart (there are a few exceptions to the symmetry rule) in that, among other things, you provide a name for your program. The name must be unique and is registered by LXRT. Thus, other programs, real time or not, can find the task pointer of your program and communicate with it.

LXRT creates a real time task who becomes the "angel" of your program. The angel's job is to execute the real time services for you. For exemple, if you call rt_sleep(...), LXRT will get your angel to execute the real rt_sleep() function in the real time scheduler. Control will return to your program when the angel returns from rt_sleep().

With LXRT, can a Linux task send a message to a real time task?

Yes. You simply use the rt_send(...) primitive that you would normally use in the code of a kernel program. LXRT gets your angel to execute rt_send(...). Control returns to your program when the target task completes the corresponding rt_receive(...) call.

What happens when I send a message to another user space program?

Well, pretty much the same thing except that you now have two angels talking to each other...

Can a real time task also register a name with LXRT?

Yes. The call rt_register(name, ...) does that. Thus, other programs, real time or not, can find the task pointer of your program and communicate with it.

Where do I put the code for the "angels"?

There is not any code required for the real time component of your Linux task. LXRT uses the standard RTAI scheduler functions for that. In the QNX world, the "angel" is called a virtual circuit.

How does it work from the point of view of a user space program?

The inline functions declared in rtai_lxrt.h all do a software interrupt (int 0xFC). Linux system calls use the software int 0x80. Hence the approach is similar to a system call. LXRT sets the interrupt vector to call rtai_lxrt_handler(void), a function that saves everything on the stach, changes ds and es to __KERNEL_DS and then calls lxrt_handler, the function that does the work.

lxrt_handler(...) extracts the first argument from user space and decides what to do from the service request number srq. For real time services, lxrt_resume(...) is called with the scheduler function address pointer fun, the number of remaining arguments, a pointer to the next argument, a service type argument, and the real time task pointer. lxrt_resume(...) will do what is necessary to change the context of execution to RTAI and transfer execution to the specified function address in the real time scheduler.

lxrt_resume(...) first copies the other arguments on the stack of the real time task. Any required data is also extracted from user space and copied into rt_task->msg_buf. At this point, the addresses of three functions are stored above stack_top (LXRT made sure this wizardry would be possible when it first created the real time task):

top-1 lxrt_suspend(...)

top-2 fun(...)

top-3 lxrt_global_sti(...)

The stack is changed to point to top-3, global interrupts are disabled and and the context of execution is switched to RTAI with the call to LXRT_RESUME(rt_task). RTAI executes lxrt_global_sti(...), fun(...), and eventually lxrt_suspend(...). Remember that fun(...) is a RTAI scheduler function like, for exemple, rt_rpc(...). At this point, fun(...) may or may not complete.

The easy way back to user space - fun(...) completes immediately:

RTAI enters function lxrt_suspend(...) that sets the real time task status to 0 and calls rt_schedule(). The context of execution is eventually switched back to Linux and the system call resumes after LXRT_RESUME(rt_task). Data for mail boxes is copied to user space and a jump to ret_from_intr()is made to complete the system call.

The long way back to user space - fun(...) cannot completed immediately:

RTAI schedules Linux to run again and the state od the real time task is non zero, indicating it is held. Therfore, the system call cannot return to user space and must wait. So it sets itself TASK_INTERRUPTIBLE and calls the Linux scheduler.

Eventually fun(...) completes and RTAI enters function lxrt_suspend(...) that notices the system call is held. So RTAI pends a system call request to instruct Linux to execute another system call whose handler is function lxrt_srq_handler(void). When Linux calls lxrt_srq_handler(), the original system call is re-scheduled for execution and returns to user space as explained above.

What happens to the registered resources if the Linux task crashes?

The "informed" version of LXRT has setup a pointer to a callback function in the do_exit() code of the Linux kernel. The callback is used to free the resources that where registered by the real time task. It also deletes the real time task and unblocks any other task that may have been SEND, RPC, RETURN or SEM blocked on the real time task.

What about mail boxes?

The mail box IPC approach is connection less. In other words, it is not possible for a zombie real time task to detect that another task is MBX blocked specifically for a message from him. The solution here is to use the rt_mbx_receive_timed() with a timeout value and verify the return value to detect the error.

What about performance?

Intertask communications with LXRT are about 36% faster than with old FIFO's. Testing Linux<->Linux communications with int size msg and rep's on a P233 I got these numbers:

LXRT — 12,000 cycles RTAI-0.9x :-)

LXRT — 13,000 cycles RTAI-0.8

Fifo — 19,000 cycles RTAI-0.8

Fifo new —22,300 cycles RTAI-0.8 10% more cycles, a lot more utility

SRR —14,200 cycles QNX 4 Send/Receive/Reply implemented with a Linux module without a real time executive.

./PGGC

README.SYNCIPC

RTAI LXRT Synchronous IPC

This release of LXRT-INFORMED adds much functionality to RTAI: proxies and synchronous IPC with a practical API.

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 about 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).

Make no mistake, unlike the NMT scheduler, raw synchronous IPC has always been there in the RTAI scheduler. However, using the new raw proxies functionality, and the existing rt_rpc(...), rt_receive(...) and rt_return(...) functions available since day one in the RTAI scheduler, the following API calls have been implemented in LXRT with __full__ symmetry:

```
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 symmetry means that you can use the same API calls to communicate within the kernel, within user space, or between the kernel and user space. Plus, the bonus is that LXRT synchronous IPC is by far more efficient than FIFO's. Run your own bench marks, Paolo and I want to ear the results :-) And Yes, that's a challenge!

The program srv.c in the examples subdirectory demos all the new features. As for the previous release, you can hit control C and restart the test. LXRT recovers automatically and releases the resources, including proxy agents that may have been created in user space. Notice that clt.c specifies a lower priority (higher number...) to demonstrate that proxies have priority over messages from the client. Example server.c is still there with the dreaded divide by zero error after 6 loops.

Read file syncipc.txt to learn more about the API of these new functions. One word of caution: the pid's returned by the functions have nothing to do with the standard Linux pid's. Think of them more as handles as they are managed internally by the implementation.

For now, LXRT and the synchronous IPC API can support a maximum of 254 tasks. That may seem like a lot, but I will not be surprised when somebody

emails that's a problem. We'll cross that bridge if and when we get there (but it will not be a big problem). I noticed after writing this that the scheduler defines a maximum of 128 tasks. While debugging I got paranoid and increased the default stack size to 2048 bytes.

If you look at the code, you will notice that synchronous IPC over the network is in the works. File vc.c is a first pass implementation of virtual circuits, the agents between the network driver and the real time tasks. It needs a littlemore analysis, mainly because Linux already does a good job with networks.

One important note if you try to run the demos. You need dynamic memory allocation with rt_malloc() and rt_free(). Setup the scheduler Makefile to include the appropriate calls from the posix module contributed by Steve. Virtual circuits are dynamic animals. To try to allocate everything up front with kmalloc() would complicate things much more and LXRT is already complicatedenough. Read the FAQ and file coe-flow.txt to get a better understanding of whathappens inside LXRT and it's complexity. The modifications needed in the kernel

to support this module are now integrated into the standard RTAI build.

All this was made possible by Paolo Mantegazza who contributed the raw proxies and numerous improvements and fixes in the RTAI real time scheduler. This new and very usefull functionality is the result of good team work. Thanks Paolo.

Finally, if anyone is interested in helping me with the network stuff, just send me an email.

Pierre Cloutier

pcloutier@poseidoncontrols.com

January 10, 2000.

Floating Point Support

Floating-point operations within real-time tasks and Interrupt Service Routines (ISRs) are possible, provided that certain steps are taken before hand. For real-time tasks, the scheduler need to knows that a task requires the FPU before performing a context switch on it, in order to correctly preserve the task’s environment.

Within ISRs, one must protect the code in the same way that Linux does: On Intel processors, Linux uses the hard task switching capability, which causes the TS (task switched) flag to be set in the CR0 register. Once this flag is set, any subsequent floating-point instruction will result in the activation of Trap 7 (device unavailable) until such time as the TS flag is cleared.

Linux uses that Trap to decode that a newly switched process wants to use the FPU. It can thus clear the TS flag and set-up the process environment appropriately by saving the FPU environment, if it was in use by any previously running task. Then, when the ISR has finished using the FPU, Linux can restore the FPU environment back to that of the previously running task, that will be re-activated once the ISR exits.

Tasking FPU Implementation:

A. When you create a task using rt_task_init(), set the *uses_fpu* flag accordingly.

B. During run-time, use the rt_task_use_fpu() function call.

C. During run-time, use the rt_linux_use_fpu() function call.

D. When you load the scheduler, supply a LinuxFpu command-line parameter.

At task creation:

```
rt_task_init(     &My_Task,         // the task structure
                  My_Task_Function, // the task function
                  0,                // initial data value
                  2000,             // stack size
                  1,                // priority
                  0,                // task does not use the FPU
                  0                 // task has no signal han-
                  dler
              );
```

Note that by default, each RT_TASK has the uses_fpu flag set to false.

At run-time:

rt_task_use_fpu, informs the scheduler that floating point arithmetic operations will be used by the real-time task

```
rt_task_use_fpu(*my_task, uses_fpu_flag);
```

rt_linux_use_fpu, informs the scheduler that floating point arithmetic operations will be used by the background task, i.e.: by Linux and all of it's processes.

```
rt_linux_use_fpu(use_fpu_flag);
```

Note that when a task or the kernel uses the FPU, task switching becomes slower as the FPU context is also saved.

When loading the scheduler:

When the scheduler is loaded, you can provide a command-line parameter to turn on Linux use of the FPU. This is the command-line equivalent of rt_linux-use_fpu() described above.

To turn on Linux use of the FPU:

```
insmod rtai_sched LinuxFpu=1
```

This facility is off (disabled) by default.

Use of this parameter instructs the scheduler that the background task, Linux, and all of its processes use the FPU.

ISR FPU Implementation:

There are four macros defined within RTAI to make the job of correctly supporting FPU operations within ISRs easier:

```
#define save_cr0_and_clts(x) __asm__ __volatile__ ("movl %%cr0,%0; clts" :"=r" (x))
#define restore_cr0(x)       __asm__ __volatile__ ("movl %0,%%cr0": :"r" (x))
#define save_fpenv(x)        __asm__ __volatile__ ("fnsave %0" : "=m" (x))
#define restore_fpenv(x)     __asm__ __volatile__ ("frstor %0" :  "=m" (x))
```

Typically, these will be used in the following way, to correctly code an RTAI ISR:

```
unsigned long cr0;
unsigned long linux_fpe[27];
unsigned long task_fpe[27];

save_cr0_and_clts(cr0);          // To save the state of CR0 - mandatory
save_fpenv(linux_fpe);           // To save the Linux FPU environment.
                                 // This is needed ONLY if any Linux process
                                 // uses the FPU.

restore_fpenv(task_fpe);         // To restore your FPU environment.
                                 // This is needed ONLY if this ISR can be
                                 // interrupted or if it contains some
                                 // intermediate results

>>> This is where you put all of your ISR floating-point calculations <<<

save_fpenv(task_fpe);            // To save your FPU environment.
                                 // This is needed only if there are some
                                 // intermediate results that will be used at
                                 // the next interrupt.

restore_fpenv(linux_fpe);        // To restore the Linux FPU environment saved
                                 // above.

restore_cr0(cr0);                // To restore the Linux CR0 register – mandatory.
```

Note that the most important part of the above is the **clts**, (clear task switched flag) instruction. Saving and restoring the FPU environment is required only if Linux itself uses the FPU. Register CR0 must be saved and restored otherwise Linux will get confused about which processes require the FPU and which do not.

Common API

Overview

The Common API (as it's name suggests) provides an API Common to both RTAI and NMT RTLinux. This API is similar to the RTLinux V1 and RTAI APIs. The principal benefit of this compatibility API is that real-time developers can write code that can be compiled and run for both RTAI and RTLinux, without the need for exception code. This approach leads to more maintainable and testable code, reducing the cost of maintaining applications under the real-time projects.

Please note that the Common API was first introduced in RTAI-22.2.4.

Implementation

Implementation of the real-time Linux common API is achieved using a series of #defines, macros and inline functions. Using it in an application is a simple matter of including a header file and using the function calls declared there. This allows a developer to write real-time Linux source code that will compile and run under either RTAI or NMT RTL. In addition, the common API will allow existing RTAI or NMT RTL source code to be compiled under both RTAI and NMT RTL at the same time.

The relevant header file is called "rt_compat.h" and is to be found in the <rtai>/ include directory.

Common API Data Structures

```
typedef void *(*VP_FP)():
typedef void *(*VP_FP_V)(void);

            struct task_data {
                TASK_STR task;
                VP_FP func;
                int arg;
                int stack_size;
                int priority;
                int period;
```

```
    int uses_fpu;
    V_FP_V sig_han;
    long long period_ns;
    long long when_ns;
};
```

Note also that TASK_STR is #defined to be the appropriate, underlying task structure, which for RTAI is RT_TASK and for NMT RTL is pthread_t.

This is a new data structure used by the rt_task_create() and rt_task_del() common API calls. These are new calls not implemented under RTAI or NMT RTL.

Common API Calls

rt_task_create

NAME
rt_task_create

SYNOPSIS
static inline int rt_task_create(struct task_data *t);

DESCRIPTION
This creates a new periodic real-time task. t must be pre-allocated initialized with sensible values.

RETURN VALUE
On success, zero is returned.

SEE ALSO
rt_task_del

rt_task_del

NAME
rt_task_del

SYNOPSIS
For RTAI, this is a wrapper around rt_task_suspend() and rt_task_delete().
For NMT RTL, this is a wrapper around pthread_delete_np().

get_time_ns

NAME
get_time_ns

SYNOPSIS
For RTAI, this is a wrapper around rt_get_cpu_time_ns().
For NMT RTL, this is a wrapper around gethrtime().

rt_task_wait_period

NAME
rt_task_wait_period

SYNOPSIS

In NMT RTL, this is a wrapper around pthread_wait_np().

rt_task_suspend

NAME
rt_task_suspend

SYNOPSIS
In NMT RTL, this is a wrapper around pthread_suspend_np().

rt_task_resume

NAME
rt_task_resume

SYNOPSIS
In NMT RTL, this is a wrapper around pthread_wakeup_np().

rtl_task_make_periodic

NAME
rtl_task_make_periodic

SYNOPSIS
In NMT RTL, this is a wrapper around pthread_make_periodic_np().
In RTAI, this is a wrapper around rt_task_make_periodic().

rt_timer_stop

NAME
rt_timer_stop

SYNOPSIS
In NMT RTL, this is NULL.
In RTAI, this is a wrapper around stop_rt_timer() and rt_busy_sleep();

rt_mount

NAME
rt_mount

SYNOPSIS
In NMT RTL, this is NULL.
In RTAI, this is a wrapper around rt_mount_rtai().

rt_unmount

NAME
rt_unmount

SYNOPSIS
In NMT RTL, this is NULL.
In RTAI, this is a wrapper around rt_umount_rtai().

rtf_create

NAME
rtf_create

SYNOPSIS
In RTAI, this is a wrapper around rtf_create_using_bh().

rt_get_time_ns

NAME
rt_get_time_ns

SYNOPSIS
In NMT RTL this is a wrapper around get_time_ns().

rt_set_oneshot_mode

NAME
rt_set_oneshot_mode

SYNOPSIS
In NMT RTL this is a wrapper around rtl_set_oneshot_mode(

rt_linux_use_fpu

NAME
rt_linux_use_fpu

SYNOPSIS
In NMT RTL this is NULL.

Examples

There are a number of examples of the use of the Common API on the Lineo Open Source Software website, (http://opensource.lineo.com). These examples are available in the file *common_api-01.tar.gz.*

The *loading* example is shown here:

```
////////////////////////////////////////////////////////////////////
//
//  Copyright © 2000 Zentropic Computing
//
//  Authors:          Stuart Hughes
//  Contact:          info@zentropix.com
//  Original date:    Feb 12 2000
//  Ident:            @(#)$Id: loading.c,v 1.1 2000/06/01 11:07:39 seh Exp $
//  Description:      This code provides a linux starvation loading test
//                    be used with RTAI and RTL2.
//
//  This program is free software; you can redistribute it and/or modify
//  it under the terms of the GNU General Public License as published by
//  the Free Software Foundation; either version 2 of the License, or
//  (at your option) any later version.
//
//  This program is distributed in the hope that it will be useful,
//  but WITHOUT ANY WARRANTY; without even the implied warranty of
//  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
//  GNU General Public License for more details.
```

```
//
//  You should have received a copy of the GNU General Public License
//  along with this program; if not, write to the Free Software
//  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
//
////////////////////////////////////////////////////////////////////////
static char id_loading_c[] __attribute__ ((unused)) = "@(#)$Id: loading.c,v 1.1 2000/06/
01 11:07:39 seh Exp $";

#define FREQ            10000                   // Basic frequency in Hz
#include "rt/rt_compat.h"

#define NTASKS          2
#define LOAD           10000

int loading = 1;
MODULE_PARM(loading, "i");

// task control data
#define MASTER 0
#define SLAVE  1
void *master();
void *slave();

struct task_data td[NTASKS] = {
           // func    arg   stack prior period uses_fpu sig_han
       {  {0}, master,  0,  3000,    10,      50,     0,     0  },
       {  {0}, slave,   1,  3000,    5,       0,       0,     0  },
};

////////////////////////////////////////////////////////////////
// RT module initialisation
////////////////////////////////////////////////////////////////
int init_module(void)
{
    int i;

    printk("load factor = %d\n", loading);
    for(i = 0; i < NTASKS; i++) {
       rt_task_create( &td[i] );
    }
    return 0;
}

////////////////////////////////////////////////////////////////
// RT threads and helper functions
```

```
//////////////////////////////////////////////////////////////
void parallel(int value, int channel)
{
    static int output = 0x0000;
    if(value) {
        output |= (1 << channel);
    } else {
        output &= ~(1<< channel);
    }
    outb(output, 0x378);             // LPT1
}

void *master(void *arg)
{
    struct task_data *t       = &td[(int)arg];
    rt_task_wait_period();

    while (1) {
        // let linux breath, we need 2 cycles as we may be immediately
        // ready to run once the high prioriy task resumes
        t->when_ns     = rt_get_time_ns();
        t->period_ns   = (long long)t->period * BASE_PER;
        rtl_task_make_periodic( &t->task, t->when_ns, t->period_ns);
        rt_task_wait_period();
        rt_task_wait_period();       // for RTL you need 2 otherwise you
                                            // don't ever get any suspend time

        // turn led on for the period of time that the long running task
        // is on
        parallel(1, 0);

        // wakeup long running rt task, this will block us as we are
        // a lower priority, this locks out linux for a long time
          rt_task_resume(&td[SLAVE].task);
        parallel(0, 0);
    }
}

void *slave(void *arg)
{
    volatile unsigned long counter;

    while (1) {
        rt_task_suspend(&td[SLAVE].task);
        rtl_schedule();
        for(counter = 0; counter < LOAD * loading; counter++) {
```

```
                counter++;
                counter--;
        }
    }
}

////////////////////////////////////////////////////////////////
// RT module cleanup
////////////////////////////////////////////////////////////////
void cleanup_module(void)
{
    int i;

    rt_timer_stop();
    for(i = 0; i < NTASKS; i++) {
        rt_task_del(&td[i].task);
    }
}
////////////////////////////////////////////////////////////////
//      End of File
////////////////////////////////////////////////////////////////
```

This example tests the effect of real-time locking out the regular linux system.

Two real-time tasks are created, the master (low priority), and the slave (higher priority). The master runs cyclically and wakes-up the slave. The slave then runs at the highest priority locking out the rest of the system for a long period of time (including the master). When the slave finishes, the master immediately runs, and suspends for a few millisecond. Depending on the load factor you start the example with, you can make Linux run for a few milliseconds in every couple of seconds.

In the Makefile a loading parameter is passed in the run target. This is set to 1 to give a small load suitable for a 486. Increase this to get more loading lockout (for instance 500 on a PII 266 MHz)

Note, the example sets bit 0 (pin #2) of the parallel port while the lock out task is running, if you put an led on this pin, you can visualise the mark/space ratio (rttask/linux).

PERL

Overview

Often, during introductory course work on real-time programming, it is desirable to quickly and easily understand the dynamics of a real-time programming environment without becoming wrapped up in the precise definition of an RTOS-specific API. The PERL bindings for RTAI allow the use of the intuitive and familiar PERL scripting language to create, destroy and schedule real-time tasks using the RTAI API within user space.

This simplified programming environment provides the ability to learn the basics of real-time programming without the need to become intimate with the API.

Implementation

The PERL bindings are packaged in a PERL module, LXRT.pm. They rely on the services of LXRT and so the LXRT module must be loaded prior to running a PERL application using them. The bindings reside in the <rtai>/lxrt/LXRT-020 directory but are not made by default when RTAI is built. They must therefore be made by hand prior to use, which is achieved as follows:

```
cd <rtai>/lxrt
make perl_lxrt
```

Which does the following:

```
cd <rtai>/lxrt/LXRT-020
perl Makefile.PL
make
```

Then, to test the Perl bindings:

```
make test
```

The test program will print status information on the screen. Once you are satisfied that this is correct and that you understand what's going on, type:

```
make install
```

Note that (rtai) is the directory location of RTAI, (usually /usr/src/rtai).

The LXRT API is available to PERL applications via the LXRT.pm Perl module.

Example

The following is a listing of the test example, test.pl, which demonstrates the use of the PERL bindings module:

```
#!/usr/bin/perl -w

use LXRT;


   $mtsk_name = nam2num(“MTSK”);

   if (!( $mtsk = rt_task_init($mtsk_name, 0, 0, 0))) {

         die(“Cannot set up the scheduler $!\n”);

   }


   printf(“MASTER TASK INIT:name = %x,address = %x\n”,$mtsk_name, $mtsk);


   printf(“MASTER TASK STARTS THE ONESHOT TIMER\n”);

   rt_set_oneshot_mode();

   start_rt_timer(nano2count(1E7));


   printf(“MASTER TASK MAKES ITSELF PERIODIC. PERIOD = 1ms\n”);

   rt_task_make_periodic($mtsk, rt_get_time(), nano2count(1E6));

   rt_sleep(nano2count(1E9));


   $count = 100;

   printf(“MASTER TASK LOOPS ON WAIT PERIOD FOR %d PERIODS\n”, $count):

   while($count--) {

         printf(“PERIOD %d\n”, $count);

         rt_task_wait_period();
```

```
}

printf("MASTER TASK STOPS THE PERIODIC TIMER\n");
stop_rt_timer();

printf("MASTER TASK DELETES ITSELF\n");
rt_task_delete($mtsk);

printf("END MASTER TASK\n");
```

Pitfalls:

1. Make sure you're root or have the necessary permissions for the RTAI installation directory.
2. Make sure you get the right scheduler loaded. If in doubt check in /lib/modules/<uname –r>/misc.

The /Proc Interface

The RTAI /proc interface extends the Linux /proc file-system to show the current status of critical elements of the RTAI real-time system. Under this interface, you can get information on the state of each of the application and RTAI real-time service kernel modules, including the scheduler, FIFOs, interrupts, and memory manager.

To observe this information, list out one of the files in the /proc/rtai directory:

`i.e. cat /proc/rtai/xxxx`

where xxxx is a file name corresponding to:

scheduler
rtai

fifos
memory_manager

As an example, the following data was obtained on a 33MHz, Uni-Processor, 486 computer, with 8MB RAM, with the frank and rt_mem_test examples loaded:

lsmod

Module	Size	Used By
frank_module	1520	0 (unused)
rt_mem_test	1080	0 (unused)
rtai_fifos	36732	5 [frank_module rt_mem_test]
rtai_sched	19752	0 [frank_module rt_mem_test]
rtai	29544	1 [rtai_fifos rtai_sched]

cat /proc/rtai/rtai

RTAI Real Time Kernel, Version: 1.3

RTAI mount count: 1
APIC Frequency: 6264025
APIC Latency: 3500 ns
APIC Setup: 500 ns

Global irqs used by RTAI:

0

Cpu_Own irqs used by RTAI:

RTAI sysreqs in use:

1 2 3

cat /proc/rtai/fifos

RTAI RealTime fifos Status:

fifo No	Open Cnt	Buff Size	malloc type
0	1	4000	kmalloc
1	1	4000	kmalloc
2	1	200	kmalloc
3	1	100	kmalloc
4	1	100	kmalloc

cat /proc/rtai/scheduler

RTAI Uniprocessor RealTime Task Scheduler:

Calibrated CPU Frequency: 1193180 Hz

Calibrated 8254 interrupt to scheduler latency: 7543 ns

Calibrated one shot setup time: 3352 ns

Priority	Period (ns)	FPU	Sig	State	Task
2	50000000	Yes	No	0x3	1
0	0	No	No	0x3	2
1	0	No	No	0x3	3

cat /proc/rtai/memory_manager

RTAI Dynamic Memory Management Status:

Chunk	Size	Address	1st Free Block	Block Size
0	32768	0xc0408000	0xc0408fd4	28704
1	32768	0xc040000	0xc0400010	32740

readme.rtai_procfile

The proc file for rtai.c now prints all the global external irqs, the cpu own irqs and the RTAI internal system requests (sysreq), as used by RTAI itself and by its other buddy modules.

It prints just integer numbers so it is somewhat criptic. For a more intelligible reading note that:

- global irqs are the same as those printed by doing "cat /proc/interrupts";
- cpu own irqs ar as follows:

  0 INVALIDATE_IPI (dispatched Linux IPIrq)

  1 LOCAL_TIMER_IPI (dispatched Linux IPIrq)

  2 RESCHEDULE_IPI (dispatched Linux IPIrq)

  3 CALL_FUNCTION_IPI (dispatched Linux IPIrq)

  4 SPURIOUS_IPI (dispatched Linux IPIrq)

  5 APIC_ERROR_IPI (dispatched Linux IPIrq)

  6 RTAI_1_IPI (used for global synchronization)

  7 RTAI_2_IPI (used for hard real time in user space)

  8 RTAI_3_IPI (used to schedule on IPIs sent by another scheduling cpu)

  9 RTAI_4_IPI (used to schedule on timer);
- RTAI sysreqs are assigned in the order they are asked for. So it depends on which module you are using and in which order you loaded them. Note however that sysreqs 0 and 1 are reserved. In fact at rtai mount syreq 0 is always assigned and used by rtai_open_srq to determine the sysreq, if any, assigned to a sysreq identefier, while syreq 1 is reserved to rt_printk.

chapter 5

Code Development Techniques

What good is an operating system if you can't easily debug its applications?

A good development and debug procedure for use with RTAI is:

1. Insert the appropriate RTAI modules.
2. Develop the task using RTAI's standard API.
3. Configure the task as a soft real-time task running under LXRT.
4. Debug using standard Linux tools

   When happy?
5. Recompile as a kernel module
6. Debug using kgdb+kmod, and LTT, R2D2 run-time debuggers.

   When happy?.
7. Strip out the debug symbols for the target module.
8. Implement and deploy as a kernel module real-time task

Below, we describe the tools available for the debug elements of this development procedure.

Real-Time Task Debug

Like code development under standard Linux, real-time tasks can be debugged in a simple way, by using embedded printk statements which provide snapshots of system variables as the application executes. Also like standard Linux, real-time tasks can be debugged using powerful and debug tools, most of which have been developed with by, or with the support of Lineo Inc.

Debug Using rt_printk and dmesg

Those familiar with standard Linux kernel debug will recall that the simplest way to debug the kernel is by using the printk and dmesg combination. In standard Linux, printk is used instead of the standard it's output can either be viewed directly at the console, or by using dmesg to view the output capture by syslog.

While printk is very useful for standard kernel debug, the use of printk within a real-time task is dangerous because it can lead to blocking of the task and hence the system. Instead, RTAI includes a function called rt_printk which effectively functions identically to printk and is real-time safe. Using rt_printk, dmesg is used in the traditional manner.

An alternative to rt_printk is rt_print_to_screen(const char *fmt, ...) which can be used if you do not need to log messages but only want to display them on the screen instead.

Debug Tools

As discussed in the development fundamentals chapter of this document, applications to be debugged must be compiled using the -g flag on the command line of gcc.

Although the standard Linux debug tools: GDB (GNU Debugger), its common --but not mandatory --graphical front-end DDD (Data Display Debugger), and the kernel debug stub for GDB (kgdb/gdbtubs) provide good capabilities for standard Linux services, features, and even the kernel, none of these traditional tools address had until recently been able to debug Linux kernel modules very well.

In response to this deficiency, Lineo Industrial Solutions Group has enhanced gdb remote debugging (kgdb+kmod) and developed the run-time debugger, each suitable for real-time Linux tasks and kernel modules. Additionally, Lineo has fully funded and provided development support and testing for a RTAI aware version of the Linux Trace Toolkit.

Kgdb+kmod is a modified version of kgdb/gdbstubs but provides improved kernel module debug support including real-time modules.

The Remote Run-time Data Debugger (R2D2) allows run-time symbolicaccess to both user space applications and kernel modules including real-time modules. This provides a good tool for dynamic debug of both real-time Linux tasks and device drivers (which are examples of kernel modules).

The Linux Trace Toolkit provides and graphically displays all of the information required to reconstruct a system's behavior during a specific period of time and is similar to --and in some respects superior to, those task trace tools offered by proprietary RTOS vendors.

It is not the purpose of the text below to educate the reader on every subtlety and usage of these tools, but simply to provide an overview of their capabilities and potential. Detailed information can be found in the tool specific documentation in their distribution packages.

Kgdb+kmod Host/Target Serial Line Debug

This solution builds upon the kgdb package maintained by David Grothe, which has many contributors, including Lineo. This version has been enhanced to improve module debugging support and to allow debugging of real-time modules.

The scheme use for this package is that a debugging Linux kernel and/or kernel modules are build and deployed on a target platform. These debug targets can be stripped, as all the symbolic information is processed in the host only.

Once the debug kernel/modules have been deployed and a serial cable connected between host and target, you are ready to begin the debug session. The first step is to put the target into debug mode, this may be at the command line prompt on the target, or via the znav GUI if you have a network connection available. Once the target has been activated, you may then start ddd/gdb on the host and issue the remote connection command (if using znav, this will be done for you as part of the session set-up).

Once the host/target session is connected, you may load modules on the target for debug using the 'loadmodule' command at the gdb/ddd command line prompt. This will load the module on the target and add the symbol file on the host debugger. You may add more modules, and also the Linux kernel to build up a complete picture of the session you are debugging on the target.

Detailed information regarding the implementation and operation of kgdb+kmod debug can be found in the the zrttb documentation package, with additional information available in the kgdb+kmod source package.

Remote Run-time Data Debugger (R2D2)

Often, it is necessary to debug dynamic applications or control systems that should, in an ideal world, remain running while system tuning is taking place.

For example, if a developer is determining the control loop coefficients for an automated system, it is tedious and difficult to determine the proper values when the application must be stopped, a new value inserted, the system observed, the system is stopped again, and a new value is inserted, etc.

Lineo's Remote Run-time Data Debugger (R2D2) solves this problem by allowing the application to continue running while global variables within the program are monitored and/or changed. Although R2D2 will allow you to look at and change any globally scoped data, it cannot look at auto variables, or variables that are being dynamically created and destroyed (although if something is malloced and persists, this can be monitored).

R2D2 is able to operate with both user space applications and kernel modules. If the target of R2D2 is a real-time task then its scheduling remains uninterrupted. If the target is a user space task then the amount of intrusion and data transfer can be controlled from the application under debug (if is free to process debug request when it has available time).

R2D2 also allows both self-hosted, remote debugging via a normal network connection. To facilitate this, each target (including the localhost if that is the target), must run a small target agent called zbroker. zbroker is responsible for enforcing security policy, and carrying out the requested actions on the target.

When user space applications have been prepared for debugging (see the example t.c in the distribution), on start-up, they will register with the local zbroker. zbroker will then maintain a list of all available userspace targets available for debugging. This list is presented at the click of a button from the znav GUI, greatly simplifying the debug setup process.

For kernel modules, from znav, you may either select a currently running module, or select a new module for debug. zbroker will load the module and setup the run-time debug session for you.

Features of R2D2 include debug symbol page saving and loading, target select, quick jump back to root or parent symbols, single snapshot value, continuous variable monitor, select variable display format, and more.

Linux Trace Toolkit (LTT)

Since the advent of the computer, engineers have always sought a means to adequately quantify their performance. This need to measure performance has resulted in a number of standard analysis tools such as: /proc and hardware counters along with gprof, strace, ps and many others. However, these essential and familiar tools only provide a snapshot of the system for a single moment in time. Although the developer can set a higher polling rate in an

effort to gain a continuous performance tracing, this is often unsuccessful (try "top -q" just for the fun of it).

The Linux Trace Toolkit (LTT) project is maintained by Karim Yaghmour of OperSys Inc. (http://www.opersys.com). Under funding by Lineo, Inc he has modified LTT to be RTAI aware and interoperable, thus providing a comprehensive system trace capability to both standard Linux and now, Real-Time Linux (RTAI) tasks.

LTT provides developers with all of the information necessary to reconstruct a system's behavior over a certain period of time. Using LTT, one can graphically view the exact the dynamics of a system, answering such questions as:

- Why do certain synchronization problems occur?
- What exactly happens to an application when a packet is received for it?
- Overall, where do all the applications that I use pass their time?
- Where are the I/O latencies in a given application? etc.

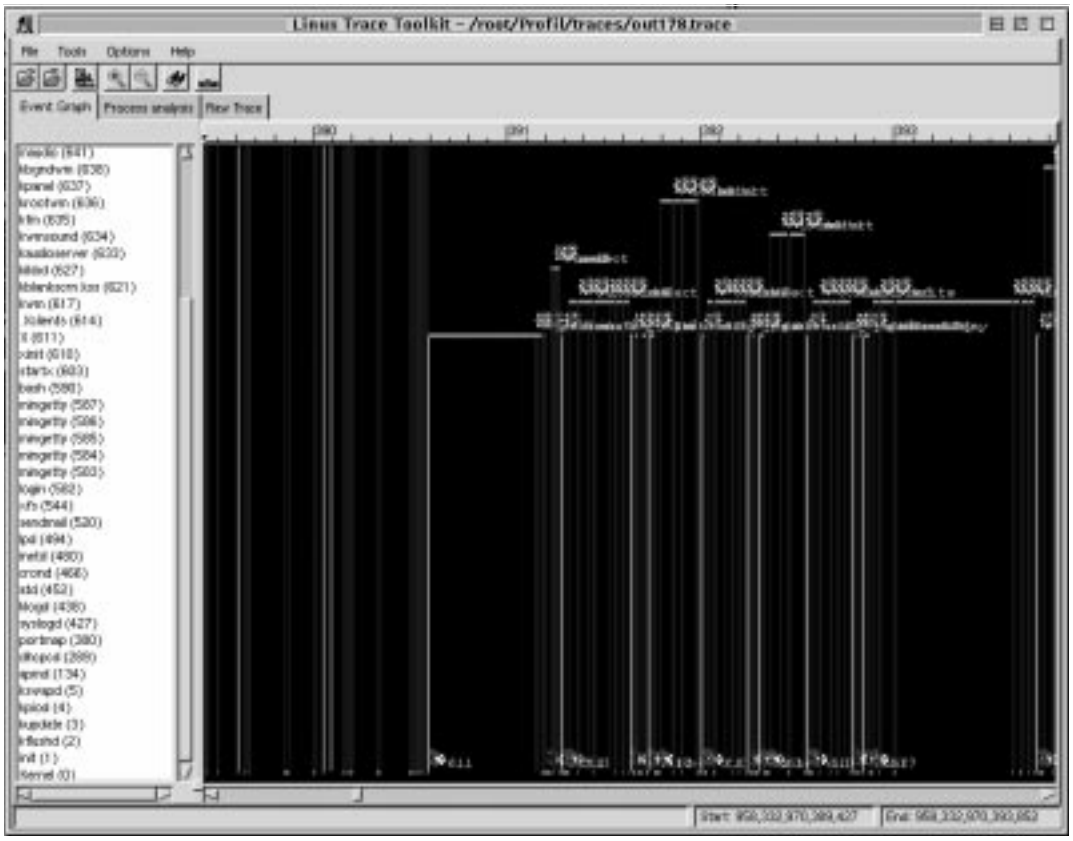
Below, we've provided a short description of the three primary outputs from LTT (event graph, process analysis, and raw list of events) in order for you to become familiar with LTT's capabilities. However, the information contained in the LTT's distribution and at the project's home page (http://www.opersys.com) should be referenced for detailed LTT information.

Event Graph

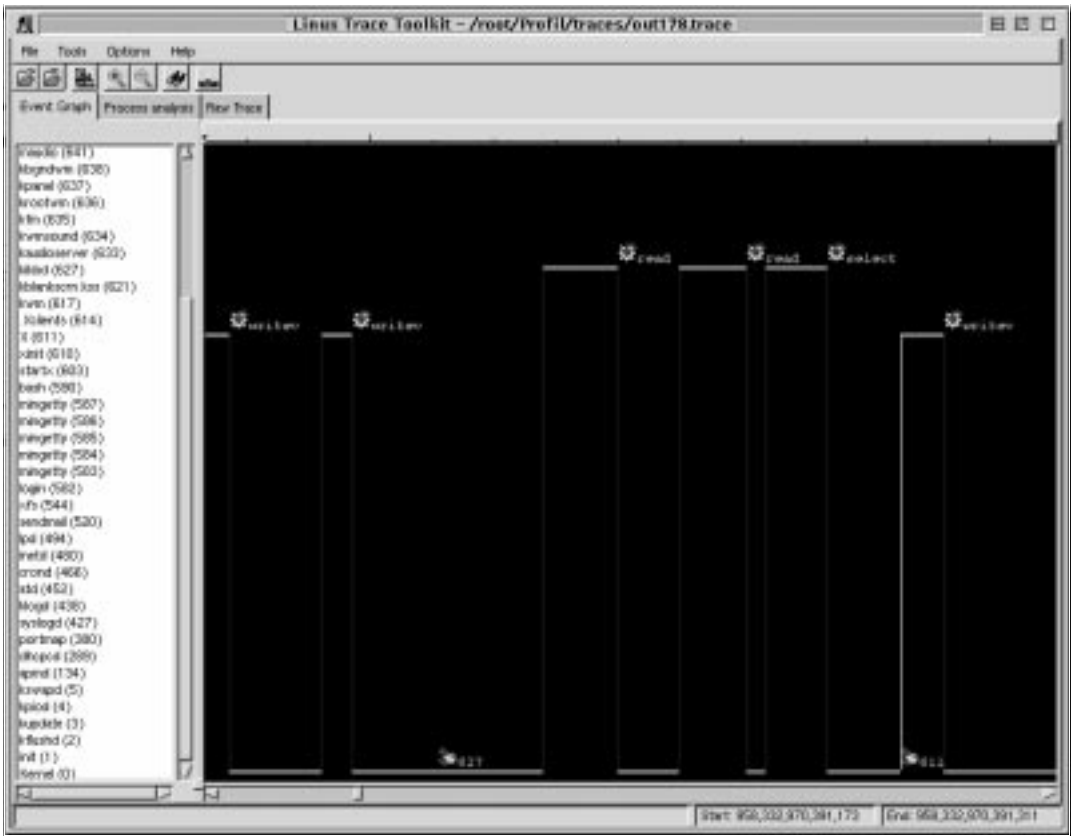
The event graph provides the viewer with a unique perspective on the flow of events in the system. Every control modifying event changes the appearance of the graph.

- A vertical line marks a shift of control from or to the kernel.
- A horizontal line marks a time lap during which a process or the kernel was executing.
- Blue vertical lines are either an entry or an exit to a system call.
- Grey vertical lines mark entry of exit from a trap.
- White line mark entry or exit by way of interrupt.
- Orange horizontal line marks time spent in the kernel.
- Green horizontal time marks time spent in a process.

- Yellow horizon lines can be placed in order to make visual task identification easier.



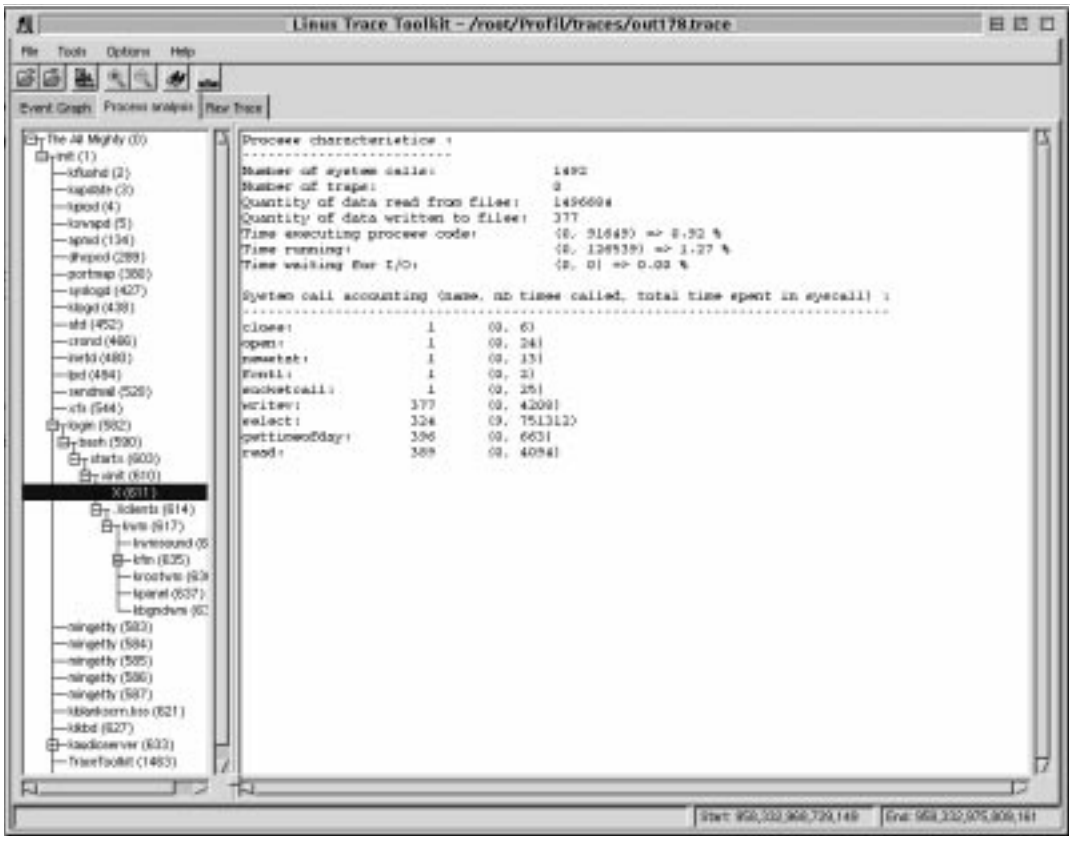
Normal event graph



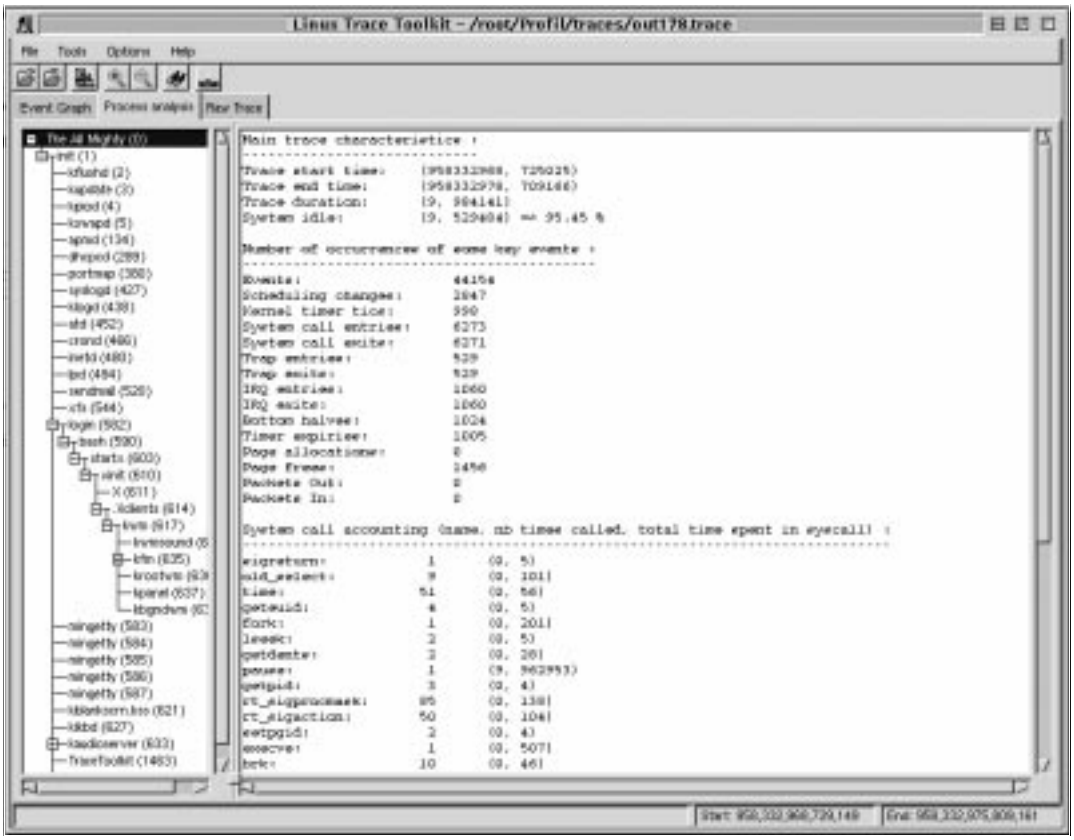
Event graph zoom-in

Process Analysis

The process analysis thumbnail provides the user with an in-depth analysis of every process that existed during the course of the trace. The items displayed are the same for all the processes except process 0 (idle) which is called 'The All Mighty' and is used to display the summurized information about the whole system.



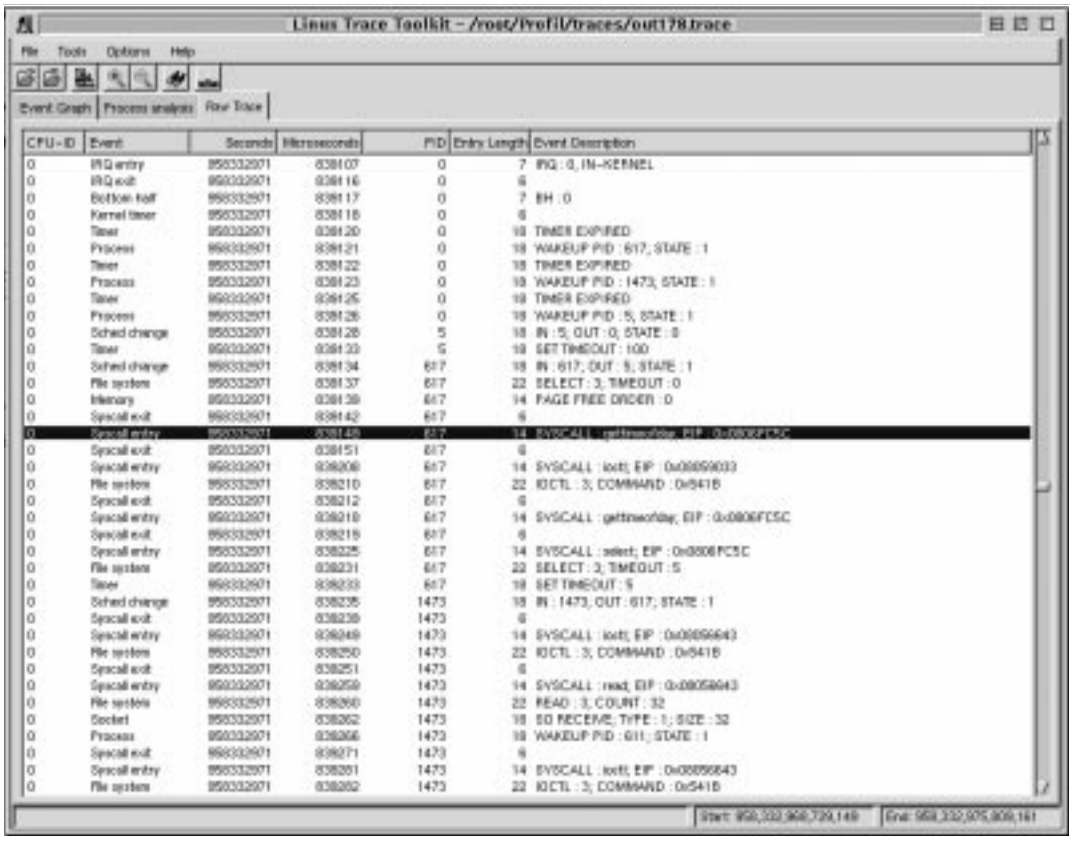
Normal Process Analysis View



Full System analysis View

Raw List of Events

The raw list of events is, as its name says, the raw list of events that occurred during the period of the trace. All the events are listed with the exact time at which they occurred, the PID of the process to which they belonged, the amount of space occupied by the event in the trace module in the kernel and the string accompanying the event, if any.



Raw List of Events

appendix **A** **RTAI API**

Overview of Available Functions

This document explains how to call the functions available in RTAI.

Task functions
Timer functions
Semaphore functions
Mailbox functions
Message handling functions
RPC (Remote Procedure Call) functions
RTAI service functions
FIFO communication functions
Auxiliary functions
POSIX extensions

Functions provided by the RTAI_SCHED module:

Task functions

rt_task_init
rt_task_init_cpuid
rt_task_delete
rt_task_make_periodic
rt_task_make_periodic_relative_ns
rt_task_wait_period
rt_task_yield
rt_task_suspend
rt_task_resume
rt_busy_sleep
rt_sleep
rt_sleep_until
rt_get_task_state

rt_whoami
rt_task_signal_handler
rt_set_runnable_on_cpus
rt_set_runnable_on_cpuid
rt_task_use_fpu
rt_linux_use_fpu
rt_preempt_always
rt_preempt_always_cpuid

Timer functions

rt_set_oneshot_mode	rt_set_periodic_mode
start_rt_timer	stop_rt_timer
count2nano	nano2count
rt_get_time	
rt_get_time_ns	
rt_get_cpu_time_ns	
next_period	

Semaphore functions

rt_sem_init
rt_sem_delete
rt_sem_signal
rt_sem_wait
rt_sem_wait_if
rt_sem_wait_until
rt_sem_wait_timed

Mailbox functions

rt_mbx_init

rt_mbx_delete	
rt_mbx_send	rt_mbx_receive
rt_mbx_send_wp	rt_mbx_receive_wp
rt_mbx_send_if	rt_mbx_receive_if
rt_mbx_send_until	rt_mbx_receive_until
rt_mbx_send_timed	rt_mbx_receive_timed

Message handling functions

rt_send	rt_receive
rt_send_if	rt_receive_if
rt_send_until	rt_receive_until
rt_send_timed	rt_receive_timed

RPC (Remote Procedure Call) functions

rt_rpc
rt_rpc_if
rt_rpc_until
rt_rpc_timed
rt_isrpc
rt_return

Functions provided by the RTAI module

RTAI service functions

rt_global_cli	rt_global_sti

rt_global_save_flags	rt_global_restore_flags
rt_global_save_flags_and_cli	
send_ipi_shorthand	
send_ipi_logical	
rt_assign_irq_to_cpu	rt_reset_irq_to_sym_mode
rt_request_global_irq	rt_free_global_irq
request_RTirq	free_RTirq
rt_request_linux_irq	rt_free_linux_irq
rt_pend_linux_irq	
rt_request_srq	rt_free_srq
rt_pend_linux_srq	
rt_request_timer	rt_free_timer
rt_mount_rtai	rt_umount_rtai
rt_ack_irq	
rt_mask_and_ack_irq	
rt_unmask_irq	
rt_startup_irq	rt_shutdown_irq
rt_enable_irq	rt_disable_irq
enable_RTirq	disable_RTirq

Functions provided by the RTAI_FIFO module

Called from an RT task	Called from a Linux process
rtf_create	rtf_open_sized
	[open]
rtf_destroy	[close]
rtf_reset	rtf_reset
rtf_resize	rtf_resize
rtf_put	[write]
	rtf_write_timed
rtf_get	[read]
	rtf_read_all_at_once
	rtf_read_timed
rtf_create_handler	
	rtf_set_async_sig

In Linux, RT fifos have to be created by mknod /dev/rtf*x* c 150 *x* where *x* is the minor device number, from 0 to 63; thus on the Linux side RT fifos can be used as standard character devices.

RTAI FIFO semaphore functions

Called from an RT task	Called from a Linux process
rtf_sem_init	rtf_sem_init
rtf_sem_post	rtf_sem_post

	rtf_sem_wait
rtf_sem_trywait	rtf_sem_trywait
	rtf_sem_timed_wait
rtf_sem_destroy	rtf_sem_destroy

RTAI FIFO auxiliary functions

rt_printk
rt_print_to_screen

Acknowledgments

Original document written by: E. Bianchi, L. Dozio, P. Mantegazza.
Dipartimento di Ingegneria Aerospaziale
Politecnico di Milano
e-mail: mantegazza@aero.polimi.it
e-mail: dozio@aero.polimi.it
Revised and updated by Gábor Kiss
Computer and Automation Institute of Hungarian Academy of Sciences
Further revised and updated by Trevor Woolven --Trevorw@lineo.com
Lineo, Inc.

TASK FUNCTIONS

rt_task_init, rt_task_init_cpuid

NAME
rt_task_init, **rt_task_init_cpuid** - create a new real time task

SYNOPSIS
#include "rtai_sched.h"
int **rt_task_init** (RT_TASK ***task*,**

```
                        void (*rt_thread)(int),
                        int data,
                        int stack_size,
                        int priority,
                        int uses_fpu,
                        void(*signal)(void)          );

int  rt_task_init_cpuid (  RT_TASK *task,
                        void (*rt_thread)(int),
                        int data,
                         int stack_size,
                        int priority,
                        int uses_fpu,
                        void(*signal)(void),
                        unsigned int cpuid           );
```

DESCRIPTION

rt_task_init and **rt_task_init_cpuid** create a real time task.

task is a pointer to an RT_TASK type structure whose space must be provided by the application. It must be kept during the whole lifetime of the real time task and cannot be an automatic variable.

rt_thread is the entry point of the task function. The parent task can pass a single integer value *data* to the new task.

stack_size is the size of the stack to be used by the new task, and *priority* is the priority to be given the task. The highest priority is 0, while the lowest is RT_LOWEST_PRIORITY.

uses_fpu is a flag. Nonzero value indicates that the task will use the floating point unit.
signal is a function that is called, within the task environment and with interrupts disabled, when the task becomes the current running task after a context switch.

The newly created real time task is initially in a suspend state. It is can be made active either with rt_task_make_periodic, rt_task_make_periodic_relative_ns or rt_task_resume .

On multiprocessor systems **rt_task_init_cpuid** assigns task to a specific CPU *cpuid*. **rt_task_init** automatically selects which CPU will the task run on. This assignment may be changed by calling rt_set_runnable_on_cpus or rt_set_runnable_on_cpuid . If *cpuid* is invalid **rt_task_init_cpuid** falls back to automatic CPU selection.

RETURN VALUE
On success 0 is returned. On failure a negative value is returned as described below.

ERRORS
-EINVAL
Task structure pointed by *task* is already in use. -ENOMEM
stack_size bytes could not be allocated for the stack.

rt_task_delete

NAME
rt_task_delete - delete a real time task

SYNOPSIS
#include "rtai_sched.h"
int **rt_task_delete** (RT_TASK ***task**);

DESCRIPTION **rt_task_delete** deletes a real time task previously created by rt_task_init or rt_task_init_cpuid .
task is the pointer to the task structure. If task *task* was waiting for a semaphore it is removed from the semaphore waiting queue else any other task blocked on message exchange with *task* is unvlocked.

RETURN VALUE
On success 0 is returned. On failure a negative value is returned as described below.

ERRORS
-EINVAL if *task* does not refer to a valid task.

rt_task_make_periodic, rt_task_make_periodic_relative_ns

NAME
rt_task_make_periodic, **rt_task_make_periodic_relative_ns** - make a task run periodically

SYNOPSIS
#include "rtai_sched.h"
int **rt_task_make_periodic** (RT_TASK ***task**,

RTIME *start_time*,
RTIME *period*);

int **rt_task_make_periodic_relative_ns** (RT_TASK ***task**,
RTIME *start_delay*,
RTIME *period*);

DESCRIPTION
rt_task_make_periodic and **rt_task_make_periodic_relative_ns** mark the task *task*, previously created with rt_task_init , as suitable for a periodic execution, with period *period*, when rt_task_wait_period is called. The time of first execution is given by *start_time* or *start_delay*. *start_time* is an absolute value measured in clock ticks. *start_delay* is relative to the current time and measured in nanosecs.

RETURN VALUE
On success 0 is returned. On failure a negative value is returned as described below.

ERRORS
-EINVAL if *task* does not refer to a valid task.

rt_task_wait_period

NAME
rt_task_wait_period - wait till next period

SYNOPSIS
#include "rtai_sched.h"
void **rt_task_wait_period** (void);

DESCRIPTION
rt_task_wait_period suspends the execution of the currently running real time task until the next period is reached. The task must have been previously marked for execution with rt_task_make_periodic or rt_task_make_periodic_relative_ns.

Note that the task is suspended only temporarily, i.e. it simply gives up control until the next time period.

rt_task_yield

NAME
rt_task_yield - yield the current task

SYNOPSIS
#include "rtai_sched.h"
void **rt_task_yield** (void);

DESCRIPTION
rt_task_yield stops the current task and takes it at the end of the list of ready tasks, with the same priority. The scheduler makes the next ready task of the same priority active.

rt_task_suspend

NAME
rt_task_suspend - suspend a task

SYNOPSIS
#include "rtai_sched.h"
int **rt_task_suspend**(RT_TASK ***task*);

DESCRIPTION
rt_task_suspend suspends execution of the task *task*. It will not be executed until a call to rt_task_resume or rt_task_make_periodic is made.
RETURN VALUE
On success 0 is returned. On failure a negative value is returned as described below.

ERRORS
-EINVAL if *task* does not refer to a valid task.

rt_task_resume

NAME
rt_task_resume - resume a task

SYNOPSIS
#include "rtai_sched.h"
int **rt_task_resume** (RT_TASK **task*);

DESCRIPTION
rt_task_resume resumes execution of the task *task* previously suspended by rt_task_suspend or makes a newly created task ready to run.

RETURN VALUE
On success 0 is returned. On failure a negative value is returned as described below.

ERRORS
-EINVAL if *task* does not refer to a valid task.

rt_busy_sleep, rt_sleep, rt_sleep_until

NAME
rt_busy_sleep, **rt_sleep**, **rt_sleep_until** - delay/suspend execution for a while

SYNOPSIS
#include "rtai_sched.h"
void **rt_busy_sleep** (int *nanosecs*);
void **rt_sleep** (RTIME *delay*);
void **rt_sleep_until** (RTIME *time*);

DESCRIPTION
rt_busy_sleep delays the execution of the caller task without giving back the control to the scheduler. This function burns away CPU cycles in a busy wait loop It may be used for very short synchronization delays only. *nanosecs* is the number of nanoseconds to wait.

rt_sleep suspends execution of the caller task for a time of *delay* internal count units. During this time the CPU is used by other tasks.

rt_sleep_until is similar to **rt_sleep** but the parameter *time* is the absolute time till the task have to be suspended. If the given time is already passed this call has no effect.

Note: a higher priority task or interrupt handler can run during wait so the actual time spent in these functions may be longer than the specified. NOTE A higher priority task or interrupt handler can run during wait so the actual time spent in these functions may be longer than the specified.

rt_get_task_state

NAME
rt_get_task_state - query task state

SYNOPSIS
#include "rtai_sched.h"
int **rt_get_task_state** (RT_TASK ***task*);

DESCRIPTION
rt_get_task_state returns the state of a real time task.
task is a pointer to the task structure.

RETURN VALUE
Task state is formed by the bitwise OR of one or more of the following flags:

READY
Task *task* is ready to run (i.e. unblocked).

SUSPENDED
Task *task* is suspended.

DELAYED
Task *task* waits for its next running period or expiration of a timeout.

SEMAPHORE
Task *task* is blocked on a semaphore.

SEND
Task *task* sent a message and waits for the receiver task.

RECEIVE
Task *task* waits for an incoming message.

RPC
Task *task* sent a Remote Procedure Call and the receiver was not get it yet.

RETURN
Task *task* waits for reply to a Remote Procedure Call.

Note: the returned task state is just an approximation. Timer and other hardware interrupts may cause a change in the state of the queried task before the caller could evaluate the returned value. Caller should disable interrupts if it wants reliable info about another task.

NOTE
rt_get_task_state does not perform any check on pointer *task*.

rt_whoami

NAME
rt_whoami - get the task pointer of the current task

SYNOPSIS
#include "rtai_sched.h"
RT_TASK ***rt_whoami** (void);

DESCRIPTION
Calling **rt_whoami** a task can get a pointer to its own task structure.

RETURN VALUE
The pointer to the current task is returned.

rt_task_signal_handler

NAME
rt_task_signal_handler - set the signal handler of a task

SYNOPSIS
#include "rtai_sched.h"
void **rt_task_signal_handler** (RT_TASK ***task*, void (***handler*)(void));

DESCRIPTION
rt_task_signal_handler installs or changes the signal function of a real time task.
task is a pointer to the real time task
handler is the entry point of the signal function.
Signal handler function can be set also when the task is newly created with rt_task_init. Signal handler is

a function called within the task environment and with interrupts disabled, when the task becomes the current running task after a context switch.

RETURN VALUE
On success 0 is returned. On failure a negative value is returned as described below.

ERRORS
-EINVAL if *task* does not refer to a valid task.

rt_set_runnable_on_cpus, rt_set_runnable_on_cpuid

NAME
rt_set_runnable_on_cpus, **rt_set_runnable_on_cpuid** - assign CPUs to a task

SYNOPSIS
#include "rtai_sched.h"
void **rt_set_runnable_on_cpus** (RT_TASK ***task**, unsigned int *cpu_mask*);
void **rt_set_runnable_on_cpuid** (RT_TASK ***task**, unsigned int *cpuid);*

DESCRIPTION
rt_set_runnable_on_cpus, **rt_set_runnable_on_cpuid** select one or more CPUs which are allowed to run task *task*. **rt_set_runnable_on_cpuid** assigns task to a specific CPU however **rt_set_runnable_on_cpus** magically selects one CPU from the given set which task *task* will run on. Bit<n of *cpu_mask* enables CPU<n.
If no CPU selected by *cpu_mask* or *cpuid* is available, both functions choose a possible CPU automagically.
Note: This call has no effect on uniprocessor systems.

rt_task_use_fpu, rt_linux_use_fpu

NAME
rt_task_use_fpu, **rt_linux_use_fpu** - set indication of FPU usage

SYNOPSIS
#include "rtai_sched.h"
int **rt_task_use_fpu** (RT_TASK***** *task*, int *use_fpu_flag***);**

void **rt_linux_use_fpu** (int *use_fpu_flag*);

DESCRIPTION
rt_task_use_fpu informs the scheduler that floating point arithmetic operations will be used by the real time task *task*.

rt_linux_use_fpu informs the scheduler that floating point arithmetic operations will be used the background task (i.e. the Linux kernel itself and *all of its processes*!).
If *use_fpu_flag* has nonzero value, FPU context is also switched when *task* or the kernel became active. This makes task switching slower. Initial value of this flag is set by rt_task_init when the real time task is created. By default Linux "task" has this flag cleared. It can be set with **LinuxFpu** command line parameter of the **rtai_sched** module.

RETURN VALUE
On success 0 is returned. On failure a negative value is returned as described below.

ERRORS
-EINVAL if *task* does not refer to a valid task.

rt_preempt_always, rt_preempt_always_cpuid

NAME
rt_preempt_always, **rt_preempt_always_cpuid** - enable hard preemption

SYNOPSIS
#include "rtai_sched.h"
void **rt_preempt_always** (int *yes_no*);
void **rt_preempt_always_cpuid** (int *yes_no*, unsigned int *cpu_id*);

DESCRIPTION
In the oneshot mode a timed task is made active/current at the expiration of the timer shot. The next timer expiration is programmed by choosing, from among the timed tasks, the one with a priority higher than the current after the current has released the CPU, always assuring the Linux timing. While this policy minimizes the programming of the oneshot mode, enhancing efficiency, it can be unsuitable when a task has to be guarded against looping by watch dog task with high priority value, as in such a case the latter as no chance of running.

Calling these functions with nonzero value assures that a timed high priority; preempting task is always

programmed to be fired while another task is current. The default is no immediate preemption in oneshot mode, firing of the next shot programmed only after the current task releases the CPU. Initial value of this flag can be set with **PreemptAlways** command line parameter of the **rtai_sched** module. Note: currently that both functions are equal, parameter *cpu_id* is ignored.

TIMER FUNCTIONS

rt_set_oneshot_mode, rt_set_periodic_mode

NAME
rt_set_oneshot_mode, **rt_set_periodic_mode** - set timer mode

SYNOPSIS
#include "rtai_sched.h"
void **rt_set_oneshot_mode** (void);
void **rt_set_periodic_mode** (void);

DESCRIPTION
rt_set_oneshot_mode sets the oneshot mode for the timer. It consists in a variable timing based on the cpu clock frequency. This allows task to be timed arbitrarily. It must be called before using any time related function, including conversions.

rt_set_periodic_mode sets the periodic mode for the timer. It consists of a fixed frequency timing of the tasks in multiple of the period set with a call to start_rt_timer . The resolution is that of the 8254 frequency (1193180 hz). Any timing request not an integer multiple of the period is satisfied at the closest period tick. It is the default mode when no call is made to set the oneshot mode.

Oneshot mode can be set initially also with **OneShot** command line parameter of the **rtai_sched** module.

NOTE
Stopping the timer by stop_rt_timer sets the timer back into its defult (periodic) mode. Call **rt_set_oneshot_mode** before each start_rt_timer if it required.

start_rt_timer, stop_rt_timer

NAME
start_rt_timer, **stop_rt_timer** - start/stop timer

SYNOPSIS
#include "rtai_sched.h"
RTIME **start_rt_timer** (int *period*);
void **stop_rt_timer** (void);

DESCRIPTION
start_rt_timer starts the timer with a period *period.* The period is in internal count units and is required only for the periodic mode. In the oneshot the parameter value is ignored. **stop_rt_timer** stops the timer. The timer mode is set to periodic.

RETURN VALUE
The period in internal count units is returned.

count2nano, nano2count

NAME
count2nano, **nano2count** - convert internal count units to nanosecs and back

SYNOPSIS
#include "rtai_sched.h"
RTIME **count2nano** (RTIME *timercounts*);
RTIME **nano2count** (RTIME *nanosecs*);

DESCRIPTION
count2nano converts the time of *timercounts* internal count units into nanoseconds.
nano2count converts the time of *nanosecs* nanoseconds into internal counts units.
Remember that the count units are related to the cpu frequency in oneshot mode and to the 8254 frequency (1193180 Hz) in periodic mode.

RETURN VALUE
The given time in nanoseconds/internal count units is returned.

rt_get_time, rt_get_time_ns, rt_get_cpu_time_ns

NAME
rt_get_time, **rt_get_time_ns**, **rt_get_cpu_time_ns** - get the current time

SYNOPSIS
#include "rtai_sched.h"
RTIME **rt_get_time** (void);
RTIME **rt_get_time_ns (**void);
RTIME **rt_get_cpu_time_ns** (void);

DESCRIPTION
rt_get_time returns the number of real time clock ticks since RT_TIMER bootup (*whatever this means*). This number is multiple of the 8254 period in periodic mode, while is multiple of cpu clock period in one-shot mode. **rt_get_time_ns** is the same as **rt_get_time** but the returned time is converted to nanoseconds.

rt_get_cpu_time_ns always returns the cpu time in nanoseconds, whatever timer is in use.

RETURN VALUE
The current time in internal count units/nanoseconds is returned.

next_period

NAME
next_period - get the time of next run

SYNOPSIS
#include "rtai_sched.h"
RTIME **next_period** (void);

DESCRIPTION
next_period returns the time when the caller task will run next.

RETURN VALUE
The next period time in internal count units is returned.

rt_busy_sleep, rt_sleep, rt_sleep_until

NAME
rt_busy_sleep, **rt_sleep**, **rt_sleep_until** - delay/suspend execution for a while

SYNOPSIS
#include "rtai_sched.h"
void **rt_busy_sleep** (int *nanosecs*);
void **rt_sleep** (RTIME *delay*);
void **rt_sleep_until** (RTIME *time*);

DESCRIPTION
rt_busy_sleep delays the execution of the caller task without giving back the control to the scheduler. This function burns away CPU cycles in a busy wait loop It may be used for very short synchronization delays only. *nanosecs* is the number of nanoseconds to wait.

rt_sleep suspends execution of the caller task for a time of *delay* internal count units. During this time the CPU is used by other tasks.

rt_sleep_until is similar to **rt_sleep** but the parameter *time* is the absolute time till the task have to be suspended. If the given time is already passed this call has no effect.

NOTE A higher priority task or interrupt handler can run during wait so the actual time spent in these functions may be longer than the specified.

SEMAPHORE FUNCTIONS

rt_sem_init

NAME
rt_sem_init - initialize a semaphore

SYNOPSIS
#include "rtai_sched.h"
void **rt_sem_init** (SEM***** *sem*, int *value*);

DESCRIPTION

rt_sem_init initializes a semaphore *sem*. A semaphore can be used for communication and synchronization among real-time tasks.

sem must point to a statically allocated structure. *value* is the initial value of the semaphore (usually 1). Positive value of the semaphore variable shows how many tasks can do a 'P' operation without blocking. (Say how many tasks can enter the critical region.) Negative value of a semaphore shows that how many task is blocked on it. (Unless it is initialized to negative in advance but this would be totally senseless).

RETURN VALUE
None

ERRORS
None

NOTE
Just for curiosity: the explanation of "P" and "V":

The name of the *P operation* comes the Dutch "prolagen", a combination of "proberen" (to try) and "verlagen" (to reduce). It is also derived from the word "passeren" (to pass).

The name of the *V operation* comes from the Dutch "verhogen" (to increase) or "vrygeven" (to release). (Source: Daniel Tabak - Multiprocessors, Prentice Hall, 1990.)

rt_sem_delete

NAME
rt_sem_delete - delete a semaphore

SYNOPSIS
#include "rtai_sched.h"
int **rt_sem_delete** (SEM***** *sem*);

DESCRIPTION
rt_sem_delete deletes a semaphore previously created with rt_sem_init .
sem points to the structure used in the corresponding call to rt_sem_init.
Any task blocked on this semaphore is allowed to run when semaphore is destroyed.

RETURN VALUE

On success, 0 is returned. On failure, a nonzero value is returned, as described below.

ERRORS
0xffff if *sem* does not refer to a valid semaphore. NOTE
-EINVAL would be more a consistent error code.

rt_sem_signal

NAME
rt_sem_signal - signalling a semaphore

SYNOPSIS
#include "rtai_sched.h"
int **rt_sem_signal** (SEM***** *sem*);

DESCRIPTION
rt_sem_signal is the semaphore post (sometimes known as 'give', 'signal', or 'V') operation. It is tipically called when the task leaves a critical region. The semaphore value is incremented and tested. If the value is not positive, the first task in semaphore's waiting queue is allowed to run. **rt_sem_signal** does not block the caller task. *sem* points to the structure used in the call to rt_sem_init.

RETURN VALUE
On success, 0 is returned. On failure, a nonzero value is returned as described below.

ERRORS
0xffff if *sem* does not refer to a valid semaphore.

NOTE
-EINVAL would be more a consistent error code.

rt_sem_wait

NAME
rt_sem_wait - wait a semaphore

SYNOPSIS
#include "rtai_sched.h"

int **rt_sem_wait** (SEM***** *sem*);

DESCRIPTION
rt_sem_wait is the semaphore wait (sometimes known as 'take' or 'P') operation. It is tipically called when a task enters a critical region. The semaphore value is decremented and tested. If it is still non-negative

rt_sem_wait returns immediately. Otherwise the caller task is blocked and queued up. Queueing may happen in priority order or on FIFO base. This is determined by compile time option SEM_PRIORD and, in this case, **rt_sem_wait** returns if The caller task is in the first place of the waiting queue and an other task issues a rt_sem_signal call;

An error occurs (e.g. the semaphore is destroyed); *sem* points to the structure used in the call to rt_sem_init.

RETURN VALUE

On success an undetermined number is returned. (Actually the return value somehow depends on the semaphore value.)

On failure, a special value is returned as described below.

ERRORS
0xffff if *sem* does not refer to a valid semaphore.

NOTE
The normal return value should not depend on the current value of the semaphore. In the current implementation number 0xffff can be returned under normal circumstances too and it is undistinguishable from the error code, so avoid counting up to 0xffff.

rt_sem_wait_if

NAME
rt_sem_wait_if - take a semaphore if possible

SYNOPSIS
#include "rtai_sched.h"
int **rt_sem_wait_if** (SEM***** *sem*);

DESCRIPTION
rt_sem_wait_if is a version of the semaphore wait (sometimes known as 'take' or 'P') operation. It is similar to **rt_sem_wait** but it is never blocks the caller. If the semaphore is not free, **rt_sem_wait_if** returns immediately and the semaphore value remains unchanged.

RETURN VALUE
On failure a special value is returned as described below. Otherwise the return value is undetermined. (Actually it is somehow derived from the current value of the semaphore.)

ERRORS
0xffff if *sem* does not refer to a valid semaphore.

NOTE
The normal return value should not depend on the current value of the semaphore. In the current implementation number 0xffff can be returned under normal circumstances too and it is undistinguishable from the error code, so avoid counting up to 0xffff.

Moreover the caller cannot figure out, whether if taking the semaphore was successful or not.

rt_sem_wait_until , rt_sem_wait_timed

NAME
rt_sem_wait_until , **rt_sem_wait_timed** - wait a semaphore with timeout

SYNOPSIS
#include "rtai_sched.h"
int **rt_sem_wait_until** (SEM***** *sem***,** RTIME *time*);
int **rt_sem_wait_timed** (SEM***** *sem***,** RTIME *delay);*

DESCRIPTION
rt_sem_wait_until and **rt_sem_wait_timed** are version of the semaphore wait (sometimes known as 'take' or 'P') operation. The semaphore value is decremented and tested. If it is still non-negative these functions return immediately. Otherwise the caller task is blocked and queued up. Queueing may happen in priority order or on FIFO base. This is determined by compile time option SEM_PRIORD. In this case these functions return if The caller task is in the first place of the waiting queue and an other task issues a rt_sem_signal call;
Timeout occurs;

An error occurs (e.g. the semaphore is destroyed); In case of timeout the semaphore value is incremented before return.

time is an absolute value, *delay* is relative to the current time.

RETURN VALUE
On failure a special value is returned as described below. Otherwise the return value is undetermined. (Actually it is somehow derived from the current value of the semaphore.)

ERRORS
0xffff if *sem* does not refer to a valid semaphore.

BUGS
The normal return value should not depend on the current value of the semaphore. In the current implementation number 0xffff can be returned under normal circumstances too and it is undistinguishable from the error code., so avoid counting up to 0xffff.

Moreover the caller cannot figure out, whether if taking the semaphore was successful or not.

MAILBOX FUNCTIONS

rt_mbx_init

NAME
rt_mbx_init - initialize mailbox

SYNOPSIS
#include "rtai_sched.h"
int **rt_mbx_init** (MBX***** *mbx*, int *size*);

DESCRIPTION
rt_mbx_init initializes a mailbox of size *size*. *mbx* have to point to a statically allocated structure. Using mailboxes is a flexible method of task-to-task communication. Tasks are allowed to send arbitrary size messages by using any mailbox buffer size. Clearly you should use a buffer sized at least as the largest message you envisage. However if you expect a message larger than the average message size very rarely you can use a smaller buffer without much loss of efficiency. In such a way you can set up your own mailbox usage protocol, e.g. using fix size messages with a buffer that is an integer multiple of such

a size guarantees that each message is sent/received atomically to/from the mailbox. Multiple senders and receivers are allowed and each will get the service it requires in turn, according to its priority.

RETURN VALUE
On success 0 is returned. On failure a negative value is returned as described below.

ERRORS
-EINVAL if space could not be allocated for the mailbox buffer.

rt_mbx_delete

NAME
rt_mbx_delete - delete mailbox

SYNOPSIS
#include "rtai_sched.h"
int **rt_mbx_delete** (MBX***** *mbx*);

DESCRIPTION
rt_mbx_delete removes a mailbox previously created with rt_mbx_init . *mbx* points to the structure used in the corresponding call to rt_mbx_init. RETURN VALUE
On success 0 is returned. On failure a negative value is returned as described below.

ERRORS
-EINVAL if *mbx* points to an invalid mailbox.
-EFAULT if *mbx* found in an invalid state.

rt_mbx_send

NAME
rt_mbx_send - send message unconditionally

SYNOPSIS
#include "rtai_sched.h"
int **rt_mbx_send** (MBX***** *mbx*, void***** *msg*, int *msg_size*);

DESCRIPTION
rt_mbx_send sends a message *msg* of *msg_size* bytes to the mailbox *mbx*. The caller will be blocked until the whole message is enqueued or an error occurs.

RETURN VALUE
On success, the number of unsent bytes is returned. On failure a negative value is returned as described below.

ERRORS
-EINVAL if *mbx* points to an invalid mailbox.

rt_mbx_send_wp

NAME
rt_mbx_send_wp - send bytes as many as possible

SYNOPSIS
#include "rtai_sched.h"
int **rt_mbx_send_wp** (MBX***** *mbx*, void***** *msg*, int *msg_size*);

DESCRIPTION
rt_mbx_send_wp sends as many as possible of bytes of message *msg* to mailbox *mbx* then returns immediately. The message length is *msg_size*.

RETURN VALUE
On success, the number of unsent bytes is returned. On failure a negative value is returned as described below.

ERRORS
-EINVAL if *mbx* points to an invalid mailbox.

rt_mbx_send_if

NAME
rt_mbx_send_if - send a message if possible

SYNOPSIS
#include "rtai_sched.h"
int **rt_mbx_send_if** (MBX***** *mbx*, void***** *msg*, int *msg_size*);

DESCRIPTION
rt_mbx_send_if tries to enqueue a message *msg* of *msg_size* bytes to the mailbox *mbx*. It returns immediately, the caller is never blocked.

RETURN VALUE
On success, the number of unsent bytes (0 or *msg_size*) is returned. On failure a negative value is returned as described below.

ERRORS
-EINVAL if *mbx* points to an invalid mailbox.

rt_mbx_send_until, rt_mbx_send_timed

NAME
rt_mbx_send_until, **rt_mbx_send_timed** - send a message with timeout

SYNOPSIS
#include "rtai_sched.h"
int **rt_mbx_send_until** (MBX***** *mbx*, void***** *msg*, int *msg_size*, RTIME *time*);
int **rt_mbx_send_timed** (MBX***** *mbx*, void***** *msg*, int *msg_size*, RTIME *delay*);

DESCRIPTION
rt_mbx_send_until and **rt_mbx_send_timed** send a message *msg* of *msg_size* bytes to the mailbox *mbx*. The caller will be blocked until all bytes of message is enqueued, timeout expires or an error occurs. *time* is an absolute value. *delay* is relative to the current time.

RETURN VALUE
On success, the number of unsent bytes is returned. On failure a negative value is returned as described below.

ERRORS
-EINVAL if *mbx* points to an invalid mailbox.

rt_mbx_receive

NAME
rt_mbx_receive - receive message unconditionally

SYNOPSIS
#include "rtai_sched.h"
int **rt_mbx_receive** (MBX***** *mbx*, void* *msg*, int *msg_size*);

DESCRIPTION
rt_mbx_receive receives a message of *msg_size* bytes from the mailbox *mbx*. The caller will be blocked until all bytes of the message arrive or an error occurs.

msg points to a buffer provided by the caller.

RETURN VALUE
On success, the number of received bytes is returned. On failure a negative value is returned as described below.

ERRORS
-EINVAL if *mbx* points to an invalid mailbox.

rt_mbx_receive_wp

NAME
rt_mbx_receive_wp - receive bytes as many as possible

SYNOPSIS
#include "rtai_sched.h"
int **rt_mbx_receive_wp** (MBX***** *mbx*, void***** *msg*, int *msg_size*);

DESCRIPTION
rt_mbx_receive_wp receives at most *msg_size* of bytes of message from mailbox *mbx* then returns immediately.

msg points to a buffer provided by the caller.

RETURN VALUE

On success, the number of received bytes is returned. On failure a negative value is returned as described below.

ERRORS
-EINVAL if *mbx* points to an invalid mailbox.

rt_mbx_receive_if

NAME
rt_mbx_receive_if - receive a message if available

SYNOPSIS
#include "rtai_sched.h"
int **rt_mbx_receive_if** (MBX* *mbx*, void* *msg*, int *msg_size*);

DESCRIPTION
rt_mbx_receive_if receives a message from the mailbox *mbx* if the whole message of *msg_size* bytes is available immediately.

msg points to a buffer provided by the caller.

RETURN VALUE
On success, the number of received bytes (0 or *msg_size*) is returned. On failure a negative value is returned as described below.

ERRORS
-EINVAL if *mbx* points to an invalid mailbox.

rt_mbx_receive_until, rt_mbx_receive_timed

NAME
rt_mbx_receive_until, **rt_mbx_receive_timed** - receive a message with timeout

SYNOPSIS
#include "rtai_sched.h"

int **rt_mbx_receive_until** (MBX***** *mbx*, void***** *msg*, int *msg_size*, RTIME *time*);
int **rt_mbx_receive_timed** (MBX***** *mbx*, void***** *msg*, int *msg_size*, RTIME *delay*);

DESCRIPTION
rt_mbx_receive_until and **rt_mbx_receive_timed** receive a message of *msg_size* bytes from the mailbox *mbx*. The caller will be blocked until all bytes of the message arrive, timeout expires or an error occurs.
time is an absolute value. *delay* is relative to the current time.
msg points to a buffer provided by the caller.

RETURN VALUE
On success, the number of received bytes is returned. On failure a negative value is returned as described below.

ERRORS
-EINVAL if *mbx* points to an invalid mailbox.

Message Handling

rt_send

NAME
rt_send - send a message

SYNOPSIS
#include "rtai_sched.h"
RT_TASK***** **rt_send** (RT_TASK***** *task***,** unsigned int *msg*);

DESCRIPTION
rt_send sends the message *msg* to the task *task*. If the receiver task is ready to get the message **rt_send** returns immediately. Otherwise the caller task is blocked and queued up. (Queueing may happen in priority order or on FIFO base. This is determined by compile time option MSG_PRIORD.)

RETURN VALUE
On success, *task* (the pointer to the task that received the message) is returned. If the caller is unblocked but message has not been sent (e.g. the task *task* was killed before receiving the message) 0 is returned.

On other failure, a special value is returned as described below.

ERRORS
0 if the receiver task was killed before receiving the message.
0xffff if *task* does not refer to a valid task.

rt_send_if

NAME
rt_send_if - send a message if possible

SYNOPSIS
#include "rtai_sched.h"
RT_TASK***** **rt_send_if** (RT_TASK***** *task*, unsigned int *msg*);

DESCRIPTION
rt_send_if tries to send the message *msg* to the task *task*. The caller task is never blocked.

RETURN VALUE
On success, *task* (the pointer to the task that received the message) is returned. If message has not been sent, 0 is returned. On other failure, a special value is returned as described below.

ERRORS
0 if the *task* was not ready to receive the message.
0xffff if *task* does not refer to a valid task.

rt_send_until, rt_send_timed

NAME
rt_send_until, **rt_send_timed** - send a message with timeout

SYNOPSIS
#include "rtai_sched.h"
RT_TASK***** **rt_send_until** (RT_TASK***** *task*, unsigned int *msg*, RTIME *time*);
RT_TASK***** **rt_send_timed** (RT_TASK***** *task*, unsigned int *msg*, RTIME*delay*);

DESCRIPTION
rt_send_until and **rt_send_timed** send the message *msg* to the task *task*. If the receiver task is ready to get the message these functions return immediately. Otherwise the caller task is blocked and queued up. (Queueing may happen in priority order or on FIFO base. This is determined by compile time option MSG_PRIORD.) In this case these functions return if The caller task is in the first place of the waiting queue and the receiver gets the message;
Timeout occurs;
An error occurs (e.g. the receiver task is killed); *time is an absolute value, delay is relative to the current time.*

RETURN VALUE
On success, *task* (the pointer to the task that received the message) is returned. If message has not been sent, 0 is returned. On other failure, a special value is returned as described below.

ERRORS
0 if operation timed out, message was not delivered.
0xffff if *task* does not refer to a valid task.

rt_receive

NAME
rt_receive - receive a message

SYNOPSIS
#include "rtai_sched.h"
RT_TASK***** **rt_receive** (RT_TASK***** *task*, unsigned int *******msg*);

DESCRIPTION
rt_receive gets a message from the task specified by *task*. If *task* is equal to 0, the caller accepts message from any task. If there is a pending message, **rt_receive** returns immediately. Otherwise the caller task is blocked and queued up. (Queueing may happen in priority order or on FIFO base. This is determined by compile time option MSG_PRIORD.

msg points to a buffer provided by the caller.

RETURN VALUE
On success, a pointer to the sender task is returned. If the caller is unblocked but no message has been

received (e.g. the task *task* was killed before sending the message) 0 is returned. On other failure, a special value is returned as described below.

ERRORS
0 if the sender task was killed before sending the message.
0xffff if *task* does not refer to a valid task.

rt_receive_if

NAME
rt_receive_if - receive a message if possible

SYNOPSIS
#include "rtai_sched.h"
RT_TASK***** **rt_receive_if** (RT_TASK***** *task*, unsigned int *****msg);

DESCRIPTION
rt_receive_if tries to get a message from the task specified by *task*. If *task* is equal to 0, the caller accepts message from any task. The caller task is never blocked.
msg points to a buffer provided by the caller.

RETURN VALUE
On success, a pointer to the sender task is returned. If no message has been received, 0 is returned. On other failure, a special value is returned as described below.

ERRORS
0 if there was no message to receive.
0xffff if *task* does not refer to a valid task.

rt_receive_until, rt_receive_timed

NAME
rt_receive_until, **rt_receive_timed** - receive a message with timeout

SYNOPSIS
#include "rtai_sched.h"

RT_TASK*** rt_receive_until** (RT_TASK***** *task*, unsigned int *****_msg_, RTIME*time*);
RT_TASK*** rt_receive_timed** (RT_TASK***** *task*, unsigned int *****_msg_, RTIME*delay*);

DESCRIPTION
rt_receive_until and **rt_receive_timed** receive a message from the task specified by *task*. If *task* is equal to 0, the caller accepts message from any task. If there is a pending message, these functions return immediately. Otherwise the caller task is blocked and queued up. (Queueing may happen in priority order or on FIFO base. This is determined by compile time option MSG_PRIORD.) In this case these functions return if The caller task is in the first place of the waiting queue and the sender sends a message;
Timeout occurs;

An error occurs (e.g. the sender task is killed); *msg* points to a buffer provided by the caller.
time is an absolute value. delay is relative to the current time.

RETURN VALUE
On success, a pointer to the sender task is returned. If no message has been received, 0 is returned. On other failure, a special value is returned as described below.

ERRORS
0 if operation timed out, no message was received.
0xffff if *task* does not refer to a valid task.

Remote Procedure Calls

rt_rpc

NAME
rt_rpc - make a remote procedure call

SYNOPSIS
#include "rtai_sched.h"
RT_TASK ***rt_rpc** (RT_TASK *****_task_, unsigned int *msg*, unsigned int *****_reply_);

DESCRIPTION
rt_rpc makes a Remote Procedure Call. RPC is like a send/receive pair. **rt_rpc** sends the message *msg* to the task *task* then waits until a reply is received. The caller task is always blocked and queued up.

(Queueing may happen in priority order or on FIFO base. This is determined by compile time option MSG_PRIORD.)
The receiver task may get the message with any rt_receive_* function. It can send the answer with rt_return . *reply* points to a buffer provided by the caller.

RETURN VALUE
On success, *task* (the pointer to the task that received the message) is returned. If message has not been sent (e.g. the task *task* was killed before receiving the message) 0 is returned. On other failure, a special value is returned as described below.

ERRORS
0 if the receiver task was killed before receiving the message.
0xffff if *task* does not refer to a valid task.

rt_rpc_if

NAME
rt_rpc_if - make a remote procedure call if receiver is ready

SYNOPSIS
#include "rtai_sched.h"
RT_TASK ***rt_rpc_if** (RT_TASK ***task**, unsigned int *msg*, unsigned int ***reply**);

DESCRIPTION
rt_rpc_if tries to make a Remote Procedure Call. If the receiver task is ready to accept a message **rt_rpc_if** sends the message *msg* then waits until a reply is received. In this case the caller task is blocked and queued up. (Queueing may happen in priority order or on FIFO base. This is determined by compile time option MSG_PRIORD.) If the receiver is not ready **rt_rpc_if** returns immediately.
The receiver task may get the message with any rt_receive_* function. It can send the answer with rt_return. *reply* points to a buffer provided by the caller.

RETURN VALUE
On success, *task* (the pointer to the task that received the message) is returned. If message has not been sent, 0 is returned. On other failure, a special value is returned as described below.

ERRORS
0 if the *task* was not ready to receive the message or it was killed before sending the reply.
0xffff if *task* does not refer to a valid task.

rt_rpc_until, rt_rpc_timed

NAME
rt_rpc_until, **rt_rpc_timed** - make a remote procedure call with timeout

SYNOPSIS
#include "rtai_sched.h"
RT_TASK ***rt_rpc_until** (RT_TASK ***task*, unsigned int *msg*, unsigned int ***reply*, RTIME *time*);
RT_TASK ***rt_rpc_timed** (RT_TASK ***task*, unsigned int *msg*, unsigned int ***reply*, RTIME *delay*);

DESCRIPTION
rt_rpc_until and **rt_rpc_timed** make a Remote Procedure Call. They send the message *msg* to the task *task* then wait until a reply is received or a timeout occurs. The caller task is always blocked and queued up. (Queueing may happen in priority order or on FIFO base. This is determined by compile time option MSG_PRIORD.)

The receiver task may get the message with any rt_receive_* function. It can send the answer with rt_return.
reply points to a buffer provided by the caller.
*time i*s an absolute valu*e.*
delay is relative to the current time.

RETURN VALUE
On success, *task* (the pointer to the task that received the message) is returned. If message has not been sent or no answer arrived, 0 is returned. On other failure, a special value is returned as described below.

ERRORS
0 if he message could not be sent or the answer did not arrived in time.
0xffff if *task* does not refer to a valid task.

rt_isrpc

NAME
rt_isrpc - check if sender waits for reply or not

SYNOPSIS
#include "rtai_sched.h"
int **rt_isrpc** (RT_TASK ***task*);

DESCRIPTION
After receiving a message, by calling **rt_isrpc** a task can figure out whether the sender task *task* is waiting for a reply or not. No answer is required if the message sent by a rt_send_* function or the sender called rt_rpc_timed or rt_rpc_until but it is already timed out. RETURN VALUE If the *task* waits for a reply, a nonzero value is returned. Otherwise 0 is returned. BUGS **rt_isrpc** does not perform any check on pointer *task*.

rt_isrpc cannot figure out what RPC result the sender is waiting for.

NOTES
rt_return is intelligent enough to not send an answer to a task which is not waiting for it. Therefore using **rt_isrpc** is not necessary and is discouraged.

rt_return

NAME
rt_return - send back the result of a remote procedure call

SYNOPSIS
#include "rtai_sched.h"
RT_TASK ***rt_return** (RT_TASK ***task**, unsigned int *result*);

DESCRIPTION
rt_result sends the result *result* to the task *task*. If the task calling rt_rpc_* previously is not waiting the answer (i.e. killed or timed out) this return message is silently discarded.

RETURN VALUE
On success, *task* (the pointer to the task that is got the the reply) is returned. If the reply message has not been sent, 0 is returned. On other failure, a special value is returned as described below.

ERRORS
0 if the reply message was not delivered.
0xffff if *task* does not refer to a valid task.

RTAI Service Functions

rt_global_cli, rt_global_sti

NAME
rt_global_cli, **rt_global_sti** - disable/enable interrupts

SYNOPSIS
#include "rtai.h"
void **rt_global_cli** (void);
void **rt_global_sti** (void);

DESCRIPTION
rt_global_cli hard disables interrupts (cli) on the requesting cpu and acquires the global spinlock to the calling cpu so that any other cpu synchronized by this method is blocked.
rt_global_sti hard enables interrups (sti) on the calling cpu and releases the global lock.

rt_global_save_flags, rt_global_save_flags_and_cli, rt_global_restore_flags

NAME
rt_global_save_flags, **rt_global_save_flags_and_cli**, **rt_global_restore_flags** - save/restore CPU flags

SYNOPSIS
#include "rtai.h"
void **rt_global_save_flags** (unsigned long ***flags**);
int **rt_global_save_flags_and_cli** (void);
void **rt_global_restore_flags** (unsigned long *flags*);

DESCRIPTION
rt_global_save_flags saves the cpu interrupt flag (IF) and the global lock flag, in bits 9 and 0 of flags.
rt_global_save_flags_and_cli hard disables interrupts on the requesting CPU and returns old state of cpu interrupt flag (IF) and the global lock flag, in bits 9 and 0.

rt_global_restore_flags restores the cpu hard interrupt flag (IF) and the global lock flag as given by *flags*, freeing or acquiring the global lock according to the state of the global flag bit.

send_ipi_shorthand, send_ipi_logical

NAME
send_ipi_shorthand, **send_ipi_logical** - send interprocessor message

SYNOPSIS
#include "rtai.h"
void **send_ipi_shorthand** (unsigned int *shorthand*, int*irq*);
void **send_ipi_logical** (unsigned long *dest*, int *irq*);

DESCRIPTION
send_ipi_shorthand sends an inter-processor message of *irq* to:

- all CPUs if *shorthand* is equal to APIC_DEST_ALLINC;
- all but itself if *shorthand* is equal to APIC_DEST_ALLBUT;
- itself if *shorthand* is equal to APIC_DEST_SELF.

send_ipi_logical sends an interprocessor message of *irq* to all CPUs defined by *dest.*

dest is given by an unsigned long corresponding to a bitmask of the CPUs to be sent. It is used for local apics programmed in flat logical mode, so the max number of allowed CPUs is 8, a constraint that is valid for all functions and data of RTAI. The flat logical mode is set when RTAI is installed by calling rt_mount_rtai .

rt_assign_irq_to_cpu, rt_reset_irq_to_sym_mode

NAME
rt_assign_irq_to_cpu, **rt_reset_irq_to_sym_mode** - set/reset IRQ-CPU assignment

SYNOPSIS
#include "rtai.h"
int **rt_assign_irq_to_cpu** (int *irq*, int *cpu*);

int **rt_reset_irq_to_sym_mode** (int *irq*);

DESCRIPTION
rt_assign_irq_to_cpu forces the assignment of the external interrupt *irq* to the CPU *cpu*.
rt_reset_irq_to_sym_mode resets the interrupt *irq* to the symmetric interrupts management.
The symmetric mode distributes the IRQs over all the CPUs.
Note: These functions have effect on multiprocessor systems only.

RETURN VALUE
If there is one CPU in the system, 1 returned.
If there is at least 2 CPU, on success 0 is returned.
If *cpu* is refers to a non-existent CPU, the number of CPUs is returned.
On other failures, a negative value is returned as desribed below.

ERRORS
-EINVAL if *irq* is not a valid IRQ number or some internal data incosistency is found.

rt_request_global_irq, request_RTirq, rt_free_global_irq, free_RTirq

NAME
rt_request_global_irq, **request_RTirq**, **rt_free_global_irq**, **free_RTirq** - install/uninstall IT service routine

SYNOPSIS
#include "rtai.h"
int **rt_request_global_irq** (unsigned int *irq*, void (***handler*)(void));
int **rt_free_global_irq** (unsigned int *irq*);
int **request_RTirq** (unsigned int *irq*, void (***handler*)(void));
int **free_RTirq** (unsigned int *irq*);

DESCRIPTION
rt_request_global_irq installs function *handler* as an interrupt service routine for IRQ level *irq*. *handler* is then invoked whenever interrupt number *irq* occurs. It is important to note that **rt_request_global_irq** does not enable the related interrupt irq by default. It is responsability to the user to do that by using **rt_enable_irq** or **rt_startup_irq**. The installed handler must take care of properly activating any Linux handler using the same *irq* number by calling rt_pend_linux_irq .
rt_free_global_irq uninstalls the interrupt service routine.
request_RTirq and free_RTirq are macros defined in rtai.h and is supported only for backwards com-

patibility with our variant of RT_linux for 2.0.35. They are fully equivalent of the other two functions above.

RETURN VALUE
On success 0 is returned. On failure a negative value is returned as described below.

ERRORS
-EINVAL if *irq* is not a valid IRQ number or *handler* is NULL.
-EBUSY if there is already a handler of interrupt *irq*.

rt_request_linux_irq, rt_free_linux_irq

NAME
rt_request_linux_irq, **rt_free_linux_irq** - install/uninstall shared Linux interrupt handler

SYNOPSIS
#include "rtai.h"
int **rt_request_linux_irq** (unsigned int *irq*,
void (***handler***)(int *irq*, void **dev_id*, struct pt_regs ***regs***),
char ***linux_handler_id***,
void ***dev_id**);

int **rt_free_linux_irq** (unsigned int *irq*, void ***dev_id*);

DESCRIPTION
rt_request_linux_irq installs function *handler* as an interrupt service routine for IRQ level *irq*, forcing Linux to share the IRQ with other interrupt handlers. The handler is appended to any already existing Linux handler for the same irq and run as a Linux irq handler. In this way a real time application can monitor Linux interrupts handling at is will. The handler appears in /proc/interrupts.
linux_handler_id is a name for /proc/interrupts. The parameter *dev_id* is to pass to the interrupt handler, in the same way as the standard Linux irq request call.
The interrupt service routine can be uninstalled with **rt_free_linux_irq**.

RETURN VALUE
On success 0 is returned. On failure a negative value is returned as described below.

ERRORS
-EINVAL if *irq* is not a valid IRQ number or *handler* is NULL.

-EBUSY if there is already a handler for interrupt *irq*.

rt_pend_linux_irq

NAME
rt_pend_linux_irq - make Linux service an interrupt

SYNOPSIS
#include "rtai.h"
void **rt_pend_linux_irq** (unsigned int *irq*);

DESCRIPTION
rt_pend_linux_irq appends a Linux interrupt *irq* for processing in Linux IRQ mode, i.e. with interrupts fully enabled.

NOTES
rt_pend_linux_irq does not perform any check on *irq*.

rt_request_srq, rt_free_srq

NAME
rt_request_srq, **rt_free_srq** - install/unistall a system request handler

SYNOPSIS
#include "rtai.h"
int **rt_request_srq** (unsigned int *label*,
void (***rtai_handler*)(void),
long long (***user_handler*)(unsigned int *whatever*));

int **rt_free_srq** (unsigned int *srq*);

DESCRIPTION
rt_request_srq installs a system request *handler* as a system call to request a service from the kernel. System request cannot be used safely from RTAI so you can setup a handler that receives real time requests and safely executes them when Linux is running. The system call can be uninstalled using **rt_free_srq**.

rt_free_srq uninstalls the system call identified by *srq*.

RETURN VALUE
On success the number of the assigned system request is returned. On failure a negative value is returned as described below.

ERRORS
-EINVAL if *rtai_handler* is NULL or *srq* is invalid.
-EBUSY if no free srq slot is available.

rt_pend_linux_srq

NAME
rt_pend_linux_srq - append a Linux SRQ

SYNOPSIS
#include "rtai.h"
void **rt_pend_linux_srq** (unsigned int *srq*);

DESCRIPTION
rt_pend_linux_srq appends a system call request *srq* to be used as a service request to the Linux kernel. *srq* is the value returned by rt_request_srq .

NOTES
rt_pend_linux_srq does not perform any check on s*rq.*

rt_request_timer, rt_free_timer

NAME
rt_request_timer, **rt_free_timer** - install a timer interrupt handler

SYNOPSIS
#include "rtai.h"
void **rt_request_timer** (void (***handler*)(void), int *tick*, int *apic*);
void **rt_free_timer** (void);

DESCRIPTION
rt_request_timer requests a timer of period *tick* ticks, and installs the routine *handler* as a real time interrupt service routine for the timer.
Set *tick* to 0 for oneshot mode. (In oneshot mode is not used). If *apic* has a nonzero value the local APIC timer is used. Otherwise timing is based on the 8254.
rt_free_timer uninstalls the timer previously set by **rt_request_timer.**

rt_mount_rtai, rt_umount_rtai

NAME
rt_mount_rtai, **rt_umount_rtai -** initialize/uninitialize real time application interface

SYNOPSIS
#include "rtai.h"
void **rt_mount_rtai** (void);
void **rt_umount_rtai** (void);

DESCRIPTION
rt_mount_rtai initializes the real time application interface, i.e. grabs anything related to the hardware, data or service, pointed by at by the Real Time Hardware Abstraction Layer (struct rt_hal rthal;).
rt_umount_rtai unmounts the real time application interface resetting Linux to its normal state.

rt_ack_irq, rt_mask_and_ack, rt_unmask_irq, rt_startup_irq, rt_shutdown_irq, rt_enable_irq, rt_disable_irq, enable_RTirq, disable_RTirq

NAME
rt_ack_irq, rt_mask_and_ack, rt_unmask_irq, rt_startup_irq, rt_shutdown_irq, rt_enable_irq, rt_disable_irq, enable_RTirq, disable_RTirq – dispatching interrupt functions

SYNOPSIS
#include "rtai.h"
void **rt_ack_irq**(unsigned int irq);
void **rt_mask_and ack_irq**(unsigned int irq);
void **rt_unmask_irq**(unsigned int irq);
void **rt_startup_irq**(unsigned int irq);

void **rt_shutdown_irq**(unsigned int irq);
void **rt_enable_irq**(unsigned int irq);
void **rt_disable_irq**(unsigned int irq);
void **enable_Rtirq**(unsigned int irq);
void **disable_Rtirq**(unsigned int irq);

DESCRIPTION
Each of these functions dispatches the appropriate interrupt handler function, via an array of pre-allocated functions.
rt_ack_irq acknowledges the specified interrupt, **rt_mask_and_ack_irq** acknowledges the interrupt and masks it out and **rt_unmask_irq** unmasks the interrupt. **rt_startup_irq, rt_shutdown_irq, rt_enable_irq** and **rt_disable_irq** invoke the appropriate Linux IRQ methods.

NOTE
enable_RTirq and **disable_RTirq** are macros wrapping **rt_enable_irq** and **rt_disable_irq** respectively.

RTAI FIFOs

rtf_create

NAME
rtf_create - create a real-time FIFO

SYNOPSIS
#include "rtai_fifos.h"
int **rtf_create** (unsigned int *fifo*, int *size*);

DESCRIPTION
rtf_create creates a real-time fifo (RT-FIFO) of initial size *size* and assigns it the identifier *fifo*.
fifo is a small postive integer what identifies the fifo on further operations. It have to be less than RTF_NO. *fifo* may refer an existing RT-FIFO. In this case the size is adjusted if necessary. The RT-FIFO is a mechanism, implemented as a character device, to communicate between real-time tasks and ordinary Linux processes. The rtf_* functions are used by the real-time tasks; Linux processes use standard character device access functions such as read, write, and select.

RETURN VALUE

On success, *size* is returned. On failure, a negative value is returned.

ERRORS
-ENODEV if *fifo* is greater than or equal to RTF_NO.
-ENOMEM if *size* bytes could not be allocated for the RT-FIFO.

NOTES
If resizing was unsuccessful, no error code is returned.

rtf_destroy

NAME
rtf_destroy - close a real-time FIFO

SYNOPSIS
#include "rtai_fifos.h"
int **rtf_destroy** (unsigned int *fifo*);

DESCRIPTION
rtf_destroy closes a real-time fifo previously created/reopened with rtf_create or **rtf_open_sized**. An internal mechanism counts how many times a fifo was opened. Opens and closes must be in pair. **rtf_destroy** should be called as many times as rtf_create was. After the last close the fifo is really destroyed.

RETURN VALUE
On success, a non-negative number is returned. Actually it is the open counter, that means how many times **rtf_destroy** should be called yet to destroy the fifo. On failure, a negative value is returned.

ERRORS
-ENODEV if *fifo* is greater than or equal to RTF_NO.
-EINVAL if *fifo* does not refer to an open fifo.

rtf_reset

NAME
rtf_reset - reset a real-time FIFO

SYNOPSIS
#include "rtai_fifos.h"
int **rtf_reset** (unsigned int *fifo*);

DESCRIPTION
rtf_reset resets RT-FIFO *fifo* by setting its buffer pointers to zero, so that any existing data is discarded and the fifo started anew like at its creations.

RETURN VALUE
On success, 0 is returned. On failure, a negative value is returned.

ERRORS
-ENODEV if *fifo* is greater than or equal to RTF_NO.
-EINVAL if *fifo* does not refer to an open fifo.
-EFAULT if operation was unsuccessful.
NAME
rtf_resize - resize a real-time FIFO

SYNOPSIS
#include "rtai_fifos.h"
int **rtf_resize** (unsigned int *fifo*, int *size*);

DESCRIPTION
rtf_resize modifies the real-time fifo *fifo*, previously created with rtf_create, to have a new size of *size*. Any data in the fifo is discarded.

RETURN VALUE
On success, *size* is returned. On failure, a negative value is returned.

ERRORS
-ENODEV if *fifo* is greater than or equal to RTF_NO.
-EINVAL if *fifo* does not refer to an open fifo.
-ENOMEM if *size* bytes could not be allocated for the RT-FIFO. Fifo size is unchanged.

rtf_put

NAME

rtf_put - write data to FIFO

SYNOPSIS
#include "rtai_fifos.h"
int **rtf_put** (unsigned int *fifo*, void ***buf**, int *count*);

DESCRIPTION
rtf_put tries to write a block of data to a real-time fifo previously created with rtf_create.
fifo is the ID with which the RT-FIFO was created. *buf* points the block of data to be written.
count is the size of the block in bytes.
This mechanism is available only to real-time tasks; Linux processes use a write to the corresponding /dev/fifo<n device to enqueue data to a fifo. Similarly, Linux processes use read or similar functions to read the data previously written via **rtf_put** by a real-time task.

RETURN VALUE
On success, the number of bytes written is returned. Note that this value may be less than *count* if *count* bytes of free space is not available in the fifo. On failure, a negative value is returned.

ERRORS
-ENODEV if *fifo* is greater than or equal to RTF_NO.
-EINVAL if *fifo* does not refer to an open fifo.

rtf_get

NAME
rtf_get - read data from FIFO

SYNOPSIS
#include "rtai_fifos.h"
int **rtf_get** (unsigned int *fifo*, void ***buf**, int *count*);

DESCRIPTION
rtf_get tries to read a block of data from a real-time fifo previously created with a call to rtf_create.
fifo is the ID with which the RT-FIFO was created.
buf points a buffer of count bytes size provided by the caller. This mechanism is available only to real-time tasks; Linux processes use a read from the corresponding fifo device to dequeue data from a fifo. Similarly, Linux processes use write or similar functions to write the data to be read via rtf_put by a real-time task.

rtf_get is often used in conjunction with rtf_create_handler to process data received asynchronously from a Linux process. A handler is installed via rtf_create_handler; this handler calls **rtf_get** to receive any data present in the RT-FIFO as it becomes available. In this way, polling is not necessary; the handler is called only when data is present in the fifo.

RETURN VALUE
On success, the size of the received data block is returned. Note that this value may be less than *count* if *count* bytes of data is not available in the fifo. On failure, a negative value is returned.

ERRORS
-ENODEV if *fifo* is greater than or equal to RTF_NO.
-EINVAL if *fifo* does not refer to an open fifo.

rtf_create_handler

NAME
rtf_create_handler - install a FIFO handler function

SYNOPSIS
#include "rtai_fifos.h"
int **rtf_create_handler** (unsigned int *fifo*, int (***handler***)(unsigned int *fifo*));

DESCRIPTION
rtf_create_handler installs a handler which is executed when data is written to or read from a real-time fifo.
fifo is an RT-FIFO that must have previously been created with a call to rtf_create. The function pointed by handler is called whenever a Linux process accesses that fifo.
rtf_create_handler is often used in conjunction with rtf_get to process data acquired asynchronously from a Linux process. The installed handler calls rtf_get when data is present. Because the handler is only executed when there is activity on the fifo, polling is not necessary.

RETURN VALUE
On success, 0 is returned. On failure, a negative value is returned.

ERRORS
-EINVAL if *fifo* is greater than or equal to RTF_NO, or *handler* is NULL.

NOTE

rtf_create_handler does not check if FIFO referred by *fifo* is open or not. The next call of rtf_create will uninstall the handler just "installed".

RTAI FIFO Semaphore functions:

rtf_sem_init, rtf_sem_post, rtf_sem_trywait, rtf_sem_destroy

NAME
rtf_sem_init, rtf_sem_post, rtf_sem_trywait, rtf_sem_destroy – create, post, receive and destroy an RTAI FIFOs package semaphore

SYNOPSIS
#include “rtai_fifos.h”
int **rtf_sem_init**(unsigned int *minor,* int *value);*
int **rtf_sem_post(**unsigned int *minor*);
int **rtf_sem_trywait**(unsigned int *minor);*
int **rtf_sem_destroy**(unsigned int *minor);*

DESCRIPTION

RETURN VALUES
rtf_sem_init, rtf_sem_post and **rtf_sem_destroy** all return 0.
rtf_sem_trywait returns a 0 if the Semaphore is in use and a 1 if it’s free.

ERRORS
None.

RTAI FIFO Auxiliary Functions:

rt_printk, rt_print_to_screen

NAME
rt_printk, rt_print_to_screen – safe versions of printk for real-time.

SYNOPSIS
#include “rtai_fifos.h”

int **rt_printk**(const char *_fmt_, …);
int **rt_print_to_screen**(const char *fmt, …);

DESCRIPTION
These functions offer safe *printk* functionality to real-time modules.

RETURN VALUES
Both of these functions return the length of the constant char string written.

ERRORS
None

RTAI POSIX Extensions

Message Queue Functions (provided by module: rtai_pqueue)

mq_open

NAME
mq_open – open (and if necessary, create) a POSIX message queue.

SYNOPSIS
#include “rtai_pqueue.h”
mqd_t **mq_open**(char *_mq_name_,
int _oflags_,
mode_t _permissions_,
struct mq_attr *_mq_attr_);

DESCRIPTION
mq_open is used to create a message queue or open an existing one for use.

mq_name is the name by which the queue will be known or is already known.
oflags controls the way in which the message queu is opened: O_RDONLY, O_WRONLY or O_RDWR are used to specify how the task wishes to access the queue (read-only, write-only or read-write). O_NONBLOCK can be used to disable blocking when attempting to write to a full queue or read from an

empty one. O_CREAT is set to indicate that the calling task wants to create a queue rather than just access an existing one. It is only in this case that *permissions* and *mq_attr* are used. The queue will be created with the permissions (User/Group/Other, read/write/execute) specified in *permissions* and with the geometry specified in *mq_attr.* O_EXCL modifies the behaviour of O_CREAT. If both flags are set and the queue already exists, **mq_open** will return with an error, otherwise it will just attach to the existing queue.
Permissions specifies the User/Group/Other, read/write/execute permissions associated with the queue.
mq_attr specifies the message queue geometry:

mq_maxmsgs - the maximum number of messages the queue can hold
mq_msgsize – the maximum size of an individual message
mq_flags – blocking/non-blocking behaviour specifier (only used by *mq_setattr* and *mq_getattr*)
mq_curmsgs – the number of messages currently on the queue

RETURN VALUE
mq_open returns the descriptor for the created or open message queue, or a negative value as shown below if it fails. Note that the returned descriptor type *mqd_t* is an integral type but is not a file descriptor and should not be confused with one.

ERRORS
If mq_open fails to create or open the specified message queue, it returns with one of the following values:

-EACCES: if the message queue already exists and the permissions specified in *oflags* are denied or,
permission to create the message queue is denied.
-EEXIST: if O_CREAT and O_EXCL are specified in *oflags* and the message queue already exists.
-EINVAL: if an incorrect *mq_name* was passed, or either of the *mq_attr* attributes *mq_maxmsg* or *mq_msgsize* are less than or equal to zero.
-ENOENT: the message queue does not exist and O_CREAT was not specified
-EMFILE: no spare message queue descriptors left.
-ENOMEM: if there's insufficient memory to create the queue
-ENAMETOOLONG: if *mq_name* is too long.

mq_receive

NAME
mq_receive – get a message from a queue.

SYNOPSIS
#include “rtai_pqueue.h”
size_t **mq_receive**(mqd_t *mq*,
char **msg_buffer*,
size_t *buflen*,
unsigned int **msgprio*);

DESCRIPTION
mq_receive is used to retrieve a message from the queue *mq.*
The received message is retrieved from the message queue and stored in the buffer pointed to by *msg_buffer,* whose length is *buflen.*

Messages are received in priority order, using FIFO order within the same priority level.

RETURN VALUE
mq_receive returns 0, if the message was successfully retrieved from the queue, or a negative value as shown below if it fails.

ERRORS
If **mq_receive** fails, it returns with one of the following values:

-EBADF: if the descriptor *mq* is invalid
-EMSGSIZE: if the supplied *buflen* was less than the *mq_msgsize* attribute of the queue.
-EAGAIN: if the queue is non-blocking and empty.
-EINVAL: if the calling task does not have the correct permissions for reading from the queue.

mq_send

NAME
mq_send – put a message onto a queue.

SYNOPSIS
#include “rtai_pqueue.h”
int **mq_send**(mqd_t *mq*,
const char **msg*,
size_t *msglen*,
unsigned int *msgprio*);

DESCRIPTION
mq_send is used to post a message onto the queue *mq*.
The message pointed at by *msg*, of length *msglen* is placed on the queue with priority *msgprio*.

Messages are posted onto the queue in priority order with FIFO ordering used within the same priority level.

RETURN VALUE
mq_send returns 0 if the message was placed on the queue successfully, otherwise a negative number is returned as shown below:

ERRORS
If **mq_send** fails, it returns with one of the following values:

-EBADF:	if the queue descriptor *mq* is invalid.
-EINVAL:	if the message priority *msgprio* is greater than MQ_MAX_PRIO or if the writing task does not have the correct queue access permissions.
-EMSGSIZE:	if the message size *msglen* is greater than the the *mq_msgsize* attribute of the queue.
-EAGAIN:	if the message queue is non-blocking and the queue is full

mq_close

NAME
mq_close – close a message queue.

SYNOPSIS
#include “rtai_pqueue.h”
int **mq_close** (mqd_t *mq*);

DESCRIPTION
mq_close is used to sever the connection between a task and the queue *mq.*

Message queues are persistent, which means that the messages sent to a queue remain on it even if no task has the queue opened. Once a task has closed a queue, it is able to re-open it at any time (before *mq_unlink* destroys it) and retrieve any messages remaining on the queue.

RETURN VALUE

mq_close returns o if the named queue *mq* is successfully closed. Otherwise a negative number is returned as shown below:

ERRORS
If **mq_close** fails, it returns with one of the following values:

-EINVAL: if the queue descriptor *mq* is invalid, or if the task has not previously opened this queue.

mq_getattr

NAME
mq_getattr – get a message queue's attributes.

SYNOPSIS
#include "rtai_pqueue.h"
int **mq_getattr**(mqd_t *mq*,
struct mq_attr **attrbuf*);

DESCRIPTION
mq_getattr is used to retrieve the attributes associated with the message queue *mq.* The retrieved attributes are stored in the structure *attrbuf.* With one exception, these attributes are set when the message queue is created (see *mq_open* with O_CREAT set). The exception is *mq_flags* which can be set dynamically using *mq_setattr* to set the blocking/non-blocking characteristics of the queue.

The structure *mq_attr* contains two field that define the queue's specific geometry:
mq_msgsize: defines the maximum size of a single message on the queue.
mq_maxmsg: defines the maximum number of messages held on queue at any one time.
In addition, the structure *mq_attr* contains the following two fields:
mq_flags: defines the blocking characteristics of the queue: blocking or non-blocking via the flag MQ_NONBLOCK.
mq_curmsgs: defines the number of messages currently on the queue.

RETURN VALUE
mq_getattr returns 0 if it successfully retrieves the queue's attributes, otherwise a negative number is returned as shown below:

ERRORS
If **mq_getattr** fails, it returns with one of the following values:

-EBADF: if the queue descriptor *mq* is invalid.

mq_setattr

NAME
mq_setattr – set a message queue's attributes

SYNOPSIS
#include "rtai_pqueue.h"
int **mq_setattr**(mqd_t *mq*,
const struct mq_attr **new_attrs*,
struct mq_attr **old_attrs*);

DESCRIPTION
mq_setattr is used to set the blocking characteristics of the queue *mq*.

The original queue attributes for queue *mq* are retrieved and stored in the buffer pointed to by *old_attrs* if that pointer is not NULL. A NULL pointer is ignored but (obviously) the old attributes are not retreived. New attributes are set from the values stored in the structure referenced by *new_attrs.* Note however, that only the *mq_flags* attribute is changed, the others are ignored. If the flag MQ_NONBLOCK is set, the queue will be set to non-blocking mode, otherwise blocking mode is assumed.

RETURN VALUE
mq_setattr returns 0 if it was able to set the new queue attributes, otherwise a negative number is returned as shown below:

ERRORS
If **mq_setattr** fails, it returns with one of the following values:

-EBADF: if the queue descriptor *mq* is invalid
-EINVAL: if the calling task has not previously opened the queue *mq* or if it has incorrect access permissions.

mq_notify

NAME
mq_notify – instruct a message queue to notify a task when data is available.

SYNOPSIS
int **mq_notify**(mqd_t *mq*,
const struct sigevent **notification*);

DESCRIPTION
mq_notify is used to request that the calling task be notified when a message arrives on the otherwise empty message queue *mq*. This functionality is useful for asynchronous notification of message arrival to avoid polling or blocking with *mq_receive.*

A message queue can register only one such request from *all* tasks. Once one task has successfully attached a notification request all subsequent attempts by any task will fail.

The structure type *sigevent* is defined in the Linux header file *asm/siginfo.h.*

Passing in a NULL pointer for the *notification* parameter may clear a previously attached notification request.

RETURN VALUE
mq_notify returns 0 if it successfully attaches the notification request, otherwise it returns with a negative number as shown below.

ERRORS
If **mq_notify** fails, it returns with one of the following values:

-1:	if the notification request cannot be attached because another task has already attached one, or if the notification request cannot be cleared because it is owned by another task.
-EBADF:	if the queue descriptor *mq* is invalid.

mq_unlink

NAME
mq_unlink – destroy a message queue.

SYNOPSIS
int **mq_unlink**(mqd_t mq);

DESCRIPTION
mq_unlink is used to destroy the message queue *mq* if, and only if, no other task has it open. Any messages remaining on the queue are lost and the memory for it de-allocated. At this point the queue becomes inaccessible to any other task. If a task calls *mq_unlink* to destroy a queue and another task has that queue open, then the destruction is deferred until the last task closes it's access to the queue. Once *mq_unlink* is called on a queue all other tasks are prevented from opening it – the only tasks able to access the queue being those that had it open before *mq_un link* was called.

RETURN VALUE
mq_unlink returns 0 if the queue was successfully destroyed. If other tasks have the queue open then it returns with the positive count of the number of these task, otherwise it returns with a negative number as shown below:

ERRORS
If **mq_unlink** fails, it returns with one of the following values:

-EBADF: if the message queue descriptor *mq* is invalid.

Pthread functions (provided by module: rtai_pthread)

pthread_creat

NAME
pthread_create – create a Pthread

SYNOPSIS
#include "rtai_pthread.h"
int **pthread_create**(pthread_t *thread,
pthread_attr_t *attr,
void *(* start_ routine) (void *),
void *arg);

DESCRIPTION

This function is used to create a thread, running the *start_routine* with an argument value of *arg.* The allocated thread descriptor is returned in *thread.* The *attr* parameter specifies optional creation attributes.

RETURN VALUES
This function returns 0 if successful, otherwise an error code is returned as shown below.

ERRORS
If this function fails, it returns with one of the following error codes:

EINVALif *attr* is invalid
EAGAIN if there are insufficient resources available to create the thread.

pthread_creat

NAME
pthread_exit – terminate the calling Pthread

SYNOPSIS
#include “rtai_pthread.h”
void **pthread_exit**(void *retval);

DESCRIPTION
This function is used to terminate the calling thread, returning the value *retval* to any joining thread.

RETURN VALUES
None.

ERRORS
None.

pthread_self

NAME
pthread_self - get the identifier of the calling Pthread

SYNOPSIS

pthread_t **pthread_self**(void);

DESCRIPTION
This function is used to get the calling thread's descriptor.

RETURN VALUES
This function returns the descriptor of the calling thread.

ERRORS
None.

pthread_attr_init

NAME
pthread_attr_init – initialize a Pthread attributes object and fill it with default values

SYNOPSIS
int **pthread_attr_init**(pthread_attr_t *attr);

DESCRIPTION
This function is used to initialize a thread attributes object with default values specified in *attr*.

RETURN VALUES
This function returns 0 if successful, otherwise an error code is returned as shown below.

ERRORS
If this function fails, it returns with one of the following error codes:

ENOMEM if there is insufficient memory available for the attributes object.

pthread_attr_destroy

NAME
pthread_attr_destroy – destroy a Pthread attributes object.

SYNOPSIS

int **pthread_attr_destroy**(pthread_attr_t *attr);

DESCRIPTION
This function is used to destroy the thread attributes object *attr.*

RETURN VALUES
This function returns 0 if successful, otherwise an error code is returned as shown below.

ERRORS
If this function fails, it returns with one of the following error codes:

EINVALif *attr* is invalid.

pthread_attr_setdetachstate

NAME
pthread_attr_setdetachstate – set the detach state for the Pthread

SYNOPSIS
int **pthread_attr_setdetachstate**(pthread_attr_t *attr,
int detachstate);

DESCRIPTION
This function is used to specify whether threads created with the attributes object *attr* will run detached or not according to the value specified in *detatchstate.*

RETURN VALUES
This function returns 0 if successful, otherwise an error code is returned as shown below.

ERRORS
If this function fails, it returns with one of the following error codes:

EINVALif *attr* is invalid, or
if detachstate is invalid.

pthread_attr_getdetachstate

NAME
pthread_attr_getdetachstate – get the detach state for the Pthread.

SYNOPSIS
int **pthread_attr_getdetachstate**(const pthread_attr_t *attr,
int *detachstate);

DESCRIPTION
This function is used to interrogate the detach state of the threads created with the attributes object *attr.* Their detach state parameter is returned in *detachstate*.

RETURN VALUES
This function returns 0 if successful, otherwise an error code is returned as shown below.

ERRORS
If this function fails, it returns with one of the following error codes:

EINVALif *attr* is invalid.

pthread_attr_setschedparam

NAME
pthread_attr_setschedparam – set the Pthread scheduling parameters.

SYNOPSIS
int **pthread_attr_setschedparam**(pthread_attr_t *attr,
const struct sched_param *param);

DESCRIPTION
This function is used to specify the scheduling parameters used by threads created with attributes object *attr.* These values are specified in the *param* variable.

RETURN VALUES
This function returns 0 if successful, otherwise an error code is returned as shown below.

ERRORS

If this function fails, it returns with one of the following error codes:

EINVALif either *attr* or *param* is invalid.
ENOSYSif priority scheduling is not supported
ENOTSUP if *param* is set to an unsupported value

pthread_attr_getschedparam

NAME
pthread_attr_getschedparam – get the Pthread scheduling parameters.

SYNOPSIS
int **pthread_attr_getschedparam**(const pthread_attr_t *attr,
struct sched_param *param);

DESCRIPTION
This function is used to determine the scheduling parameters used by threads created with attributes object *attr.* The parameters are stored in the location pointed to by *param.*

RETURN VALUES
This function returns 0 if successful, otherwise an error code is returned as shown below.

ERRORS
If this function fails, it returns with one of the following error codes:

EINVALif *attr* is invalid.
ENOSYSif priority scheduling is not supported

pthread_attr_setschedpolicy

NAME
pthread_attr_setschedpolicy – set the Pthread scheduling policy.

SYNOPSIS
int **pthread_attr_setschedpolicy**(pthread_attr_t *attr,
int policy);

DESCRIPTION
This function is used to specify the scheduling policy used by threads created with attributes object *attr,* according to the value supplied in *policy.*

RETURN VALUES
This function returns 0 if successful, otherwise an error code is returned as shown below.

ERRORS
If this function fails, it returns with one of the following error codes:

EINVALif either *attr* or *policy* is invalid.
ENOSYSif priority scheduling is not supported
ENOTSUP if *policy* is set to an unsupported value

pthread_attr_getschedpolicy

NAME
pthread_attr_getschedpolicy – get the Pthread scheduling policy.

SYNOPSIS
int **pthread_attr_getschedpolicy**(const pthread_attr_t *attr,
int *policy);

DESCRIPTION
This function is used to determine the scheduling policy used by threads created with the attributes object *attr.* The policy they are using is stored in the location pointed to by *policy.*

RETURN VALUES
This function returns 0 if successful, otherwise an error code is returned as shown below.

ERRORS
If this function fails, it returns with one of the following error codes:

EINVALif *attr* is invalid.
ENOSYSif priority scheduling is not supported

pthread_setschedparam

NAME
pthread_setschedparam - set thread scheduling parameters.

SYNOPSIS
#include “rtai_pthread.h”
int **pthread_setschedparam**(pthread_t thread,
int policy,
const struct sched_param *param);

DESCRIPTION
This function is used to specify the scheduling *policy* and parameters *(param)* to be used by the thread *thread.*

RETURN VALUE
This function returns 0 if successful, otherwise an error code is returned as shown below.

ERRORS
If this function fails, it returns with one of the following error codes:

EINVALif either *attr* or *policy* is invalid.
ENOSYSif priority scheduling is not supported.
ESRCH if *thread* does not refer to an existing thread
ENOTSUP if *policy* or *param* is unsupported
EPERM calling thread does not have permission to set *policy* or *param*.

pthread_getschedparam

NAME
pthread_getschedparam – get thread scheduling parameters.

SYNOPSIS
#include “rtai_pthread.h”
int **pthread_getschedparam**(pthread_t thread,
int *policy,
struct sched_param *param);

DESCRIPTION
This function is used to determine the scheduling *policy* and *param*eters used by *thread.* These values are stored in the locations pointed to by the *policy* and *param* variables.

RETURN VALUE
This function returns 0 if successful, otherwise an error code is returned as shown below.

ERRORS
If this function fails, it returns with one of the following error codes:

ENOSYSif priority scheduling is not supported.
ESRCH if *thread* does not refer to an existing thread

pthread_attr_setinheritsched

NAME
pthread_attr_setinheritsched - set the Pthread scheduling inheritance

SYNOPSIS
int **pthread_attr_setinheritsched**(pthread_attr_t *attr,
int inherit);

DESCRIPTION
This function is used to specify whether threads created with attributes object *attr* will run using the scheduling policy and parameters of the creator or those specified in the attributes object *attr,* by specifying the appropriate attribute in the *inherit* parameter.

Whenever the scheduling policy or parameters in a thread attributes object are changed the *inherit* attribute must also be changed from PTHREAD_INHERIT_SCHED to PTHREAD_EXPLICIT_SCHED.

RETURN VALUES
This function returns 0 if successful, otherwise an error code is returned as shown below.

ERRORS
If this function fails, it returns with one of the following error codes:

EINVALif either *attr* or *inherit* are invalid.

ENOSYSif priority scheduling is not supported.

pthread_attr_getinheritsched

NAME
pthread_attr_getinheritsched – get the Pthread scheduling inheritance.

SYNOPSIS
int **pthread_attr_getinheritsched**(const pthread_attr_t *attr,
int *inherit);
DESCRIPTION
This function is used to determine whether threads created with attributes object *attr* are running with the scheduling policy and parameters of their creators or those specified by the attributes object. The actual parameter in use is stored in the location pointed to by *inherit.*

RETURN VALUES
This function returns 0 if successful, otherwise an error code is returned as shown below.

ERRORS
If this function fails, it returns with one of the following error codes:

EINVALif *attr* is invalid.
ENOSYSif priority scheduling is not supported.

pthread_attr_setscope

NAME
pthread_attr_setscope - set the Pthread scheduling scope.

SYNOPSIS
int **pthread_attr_setscope**(pthread_attr_t *attr,
int scope);

DESCRIPTION
This function is used to specify the contention scope to be used by threads created using the attributes object *attr,* by specifying the appropriate value in the *scope* parameter. The possible values are:

PTHREAD_SCOPE_PROCESS, where the thread contends with other threads in the process for processor resources, and PTHREAD_SCOPE_SYSTEM, where the thread contends with threads in all processes for the processor resource.

RETURN VALUES
This function returns 0 if successful, otherwise an error code is returned as shown below.

ERRORS
If this function fails, it returns with one of the following error codes:

EINVALif *attr* is invalid.
ENOSYSif priority scheduling is not supported.
ENOTSUP if *scope* is unsupported.

pthread_attr_getscope

NAME
pthread_attr_getscope – get the Pthread scheduling scope.

SYNOPSIS
int **pthread_attr_getscope**(const pthread_attr_t *attr,
int *scope);

DESCRIPTION
This function is used to determine the contention scope used by threads created with the attributes object *attr.* The actual scope of these threads is stored in the location pointed to by *scope*.

RETURN VALUES
This function returns 0 if successful, otherwise an error code is returned as shown below.

ERRORS
If this function fails, it returns with one of the following error codes:

EINVALif *attr* is invalid.
ENOSYSif priority scheduling is not supported.

sched_yield

NAME
sched_yield - yield the processor.

SYNOPSIS
int **sched_yield**(void);

DESCRIPTION
This function is used to yield the processor to another thread. The calling thread is made *ready to run* but after all other threads of the same priority. This call can be used to ensure cooperating threads of the same priority share the processor resource more equitably, especially on a uniprocessor machine.

RETURN VALUES
This function returns 0 if successful, otherwise an error code is returned as shown below.

ERRORS
If this function fails, it returns –1 and sets ERRNO as described below:

ENOSYSif *sched_yield* is not supported.

Mutex Functions (provided by module: rtai_pthread)

pthread_mutex_init

NAME
pthread_mutex_init – initialize a mutex object

SYNOPSIS
#include “rtai_pthread.h”
int **pthread_mutex_init**(pthread_mutex_t *mutex,
const pthread_mutexattr_t *mutex_attr);

DESCRIPTION
This function is used to initialize the *mutex.* The *mutex_attr* object specifies optional creation attributes.

RETURN VALUE
This function returns 0 if successful, otherwise an error code is returned as shown below.

ERRORS
If this function fails, it returns with one of the following error codes:

EINVALif *mutex_attr* is invalid.
EBUSY if the *mutex* is already initialized.
ENOMEM if there's insufficient memory.
EAGAIN if there are insufficient resources available (other than memory).
EPERM if the calling thread does not have permission to perform this operation.

pthread_mutex_destroy

NAME
pthread_mutex_destroy – destroy a mutex object.

SYNOPSIS
#include "rtai_pthread.h"
int **pthread_mutex_destroy**(pthread_mutex_t *mutex);

DESCRIPTION
This function is used to destroy the *mutex* object.

RETURN VALUE
This function returns 0 if successful, otherwise an error code is returned as shown below.

ERRORS
If this function fails, it returns with one of the following error codes:

EINVALif *mutex* is invalid.
EBUSY if the *mutex* is in use.

pthread_mutexattr_init

NAME
pthread_mutexattr_init – initialize mutex object attributes.

SYNOPSIS

#include “rtai_pthread.h”
int **pthread_mutexattr_init**(pthread_mutexattr_t *attr);

DESCRIPTION
This function is used to initialize a mutex attributes object with default attributes stored in *attr.*

RETURN VALUE
This function returns 0 if successful, otherwise an error code is returned as shown below.

ERRORS
If this function fails, it returns with one of the following error codes:

ENOMEM if there’s insufficient memory for the attributes object *attr.*

pthread_mutexattr_destroy

NAME
pthread_mutexattr_destroy – destroy mutex object attributes.

SYNOPSIS
#include “rtai_pthread.h”
int **pthread_mutexattr_destroy**(pthread_mutexattr_t *attr);

DESCRIPTION
This function is used to destroy the mutex attributes object *attr.*

RETURN VALUE
This function returns 0 if successful, otherwise an error code is returned as shown below.

ERRORS
If this function fails, it returns with one of the following error codes:

EINVALif *attr* is invalid.

pthread_mutexattr_setkind_np

NAME
pthread_mutexattr_setkind_np – set mutex kind attributes.

SYNOPSIS
#include “rtai_pthread.h”
int **pthread_mutexattr_setkind_np**(pthread_mutexattr_t *attr,
int kind);

DESCRIPTION
This function is used to specify the *kind* or *type* of mutex created using *attr.*
The possible types are as follows:

PTHREAD_MUTEX_FAST_NP is the default type; PTHREAD_MUTEX_RECURSIVE_NP allows any thread to lock the mutex ‘recursively’ – it must be unlocked an equal number of times to release the mutex; PTHREAD_MUTEX_ERRORCHECK_NP detects and reports simple usage errors.

RETURN VALUE
This function returns 0 if successful, otherwise an error code is returned as shown below.

ERRORS
If this function fails, it returns with one of the following error codes:

EINVALif either *attr* or *kind* is invalid.

NOTE:
Later versions of the Pthreads specification change *kind* to *type*, remove the _NP (non-portable) suffix, rename FAST to DEFAULT and add another *type*, NORMAL.

pthread_mutexattr_getkind_np

NAME
pthread_mutexattr_getkind_np – get mutex kind attributes.

SYNOPSIS
#include “rtai_pthread.h”
int **pthread_mutexattr_getkind_np**(const pthread_mutexattr_t *attr,

int *kind);

DESCRIPTION
This function is used to determine the *kind* or *type* of mutex created using *attr.*

RETURN VALUE
This function returns 0 if successful, otherwise an error code is returned as shown below.

ERRORS
If this function fails, it returns with one of the following error codes:

EINVAL if *attr* is invalid.

NOTE:
Later versions of the Pthreads specification change *kind* to *type*, remove the _NP (non-portable) suffix, rename FAST to DEFAULT and add another *type*, NORMAL.

pthread_mutex_trylock

NAME
pthread_mutex_trylock – non-blocking mutex lock

SYNOPSIS
#include “rtai_pthread.h”
int **pthread_mutex_trylock**(pthread_mutex_t *mutex);

DESCRIPTION
This function is used to lock the *mutex.* If the mutex is already locked the function returns immediately with EBUSY, otherwise the calling thread becomes the mutex owner until it unlocks it.

RETURN VALUE
This function returns 0 if successful, otherwise an error code is returned as shown below.

ERRORS
If this function fails, it returns with one of the following error codes:

EINVALif either *mutex* or the mutex’s *kind* is invalid or if the thread priority exceeds the mutex priority ceiling

EBUSY if *mutex* is already locked.
EDEADLCK if the calling thread already owns the mutex.

pthread_mutex_lock

NAME
pthread_mutex_lock – blocking mutex lock.

SYNOPSIS
#include “rtai_pthread.h”
int **pthread_mutex_lock**(pthread_mutex_t *mutex);

DESCRIPTION
This function is used to lock the *mutex*. If the mutex is already locked the calling thread is blocked until the mutex is unlocked. On return, the thread owns the mutex until it calls *pthread_mutex_unlock*.

RETURN VALUE
This function returns 0 if successful, otherwise an error code is returned as shown below.

ERRORS
If this function fails, it returns with one of the following error codes:

EINVALif either *mutex* or the mutex’s *kind* is invalid or if the thread priority exceeds the mutex priority ceiling
EDEADLCK if the calling thread already owns the mutex.

pthread_mutex_unlock

NAME
pthread_mutex_unlock – mutex unlock.

SYNOPSIS
#include “rtai_pthread.h”
int **pthread_mutex_unlock**(pthread_mutex_t *mutex);

DESCRIPTION

This function is used to unlock a mutex owned by the calling thread. The mutex immediately becomes unowned and if any threads are waiting for it one is awakened (dependant upon scheduling policy, relative priorities etc).

RETURN VALUE
This function returns 0 if successful, otherwise an error code is returned as shown below.

ERRORS
If this function fails, it returns with one of the following error codes:

EINVALif either *mutex* or the mutex's *kind* is invalid.
EPERM if the calling thread does not own the mutex.

Condition Variable Functions (provided by module: rtai_pthread)

pthread_cond_init

NAME
pthread_cond_init – initialize a condition variable.

SYNOPSIS
#include "rtai_pthread.h"
int **pthread_cond_init**(pthread_cond_t *_cond_,
const pthread_condattr_t *_cond_attr_);

DESCRIPTION
This function is used to initialize the condition variable *cond* with the (optional) attributes specified by *attr.*

RETURN VALUE
This function returns 0 if successful, otherwise an error code is returned as shown below.

ERRORS
If this function fails, it returns with one of the following error codes:

EINVAL if *cond* is invalid

EBUSY if *cond* has already been initialized
ENOMEM if there's insufficient memory available
EAGAIN if there are insufficient resources available (other than memory)

pthread_cond_destroy

NAME
pthread_cond_destroy – destroy a condition variable.

SYNOPSIS
#include "rtai_pthread.h"
int **pthread_cond_destroy**(pthread_cond_t *cond);

DESCRIPTION
This function is used to destroy the condition variable *cond*.

RETURN VALUE
This function returns 0 if successful, otherwise an error code is returned as shown below.

ERRORS
If this function fails, it returns with one of the following error codes:

EINVALif *cond* is invalid
EBUSY if *cond* is in use.

pthread_condattr_init

NAME
pthread_condattr_init – initialize condition attribute object.

SYNOPSIS
#include "rtai_pthread.h"
int **pthread_condattr_init**(pthread_condattr_t *attr);

DESCRIPTION
This function is used to initialize a condition variables attributes object with default values specified in

attr.

RETURN VALUE
This function returns 0 if successful, otherwise an error code is returned as shown below.

ERRORS
If this function fails, it returns with one of the following error codes:

ENOMEM if there is insufficient memory available for the attributes object.

pthread_condattr_destroy

NAME
pthread_condattr_destroy – destroy condition attribute object.

SYNOPSIS
#include “rtai_pthread.h”
int **pthread_condattr_destroy**(pthread_condattr_t *attr);

DESCRIPTION
This function is used to destroy the condition variable attributes object specified by *attr.*

RETURN VALUE
This function returns 0 if successful, otherwise an error code is returned as shown below.

ERRORS
If this function fails, it returns with one of the following error codes:

EINVALif *attr* is invalid.

pthread_cond_wait

NAME
pthread_cond_wait – wait for a condition variable to be signaled.

SYNOPSIS

#include “rtai_pthread.h”
int **pthread_cond_wait**(pthread_cond_t *cond,
pthread_mutex_t *mutex);

DESCRIPTION
This function is used to wait for the condition variable *cond* to be signaled.

RETURN VALUE
This function returns 0 if successful, otherwise an error code is returned as shown below.

ERRORS
If this function fails, it returns with one of the following error codes:

EINVAL: if either *cond* or *mutex* is invalid, or
if there are different mutexes for concurrent waits, or
if *mutex* is not owned by the calling thread.

pthread_cond_timedwait

NAME
pthread_cond_timedwait – wait for a condition variable to be signaled with timeout.

SYNOPSIS
#include “rtai_pthread.h”
int **pthread_cond_timedwait**(pthread_cond_t *cond,
pthread_mutex_t *mutex,
const struct timespec *abstime);

DESCRIPTION
This function is used to wait for a condition variable to be signaled before the absolute time specified by *abstime* is reached.

RETURN VALUE
This function returns 0 if successful, otherwise an error code is returned as shown below.

ERRORS
If this function fails, it returns with one of the following error codes:

EINVAL: if either *cond, mutex or abstime* is invalid, or
if there are different mutexes for concurrent waits, or
if *mutex* is not owned by the calling thread.

ETIMEDOUT if the time specified by *abstime* has passed.

pthread_cond_signal

NAME
pthread_cond_signal – signal a condition variable awaking one thread.

SYNOPSIS
#include “rtai_pthread.h”
int **pthread_cond_signal**(pthread_cond_t *cond);

DESCRIPTION
This function is used to signal a condition variable to one of the waiting threads. The waiter that is actually awakened depends their relative priorities and on the scheduling policy in place.

RETURN VALUE
This function returns 0 if successful, otherwise an error code is returned as shown below.

ERRORS
If this function fails, it returns with one of the following error codes:

EINVAL: if the condition variable *cond* is invalid.

pthread_cond_broadcast

NAME
pthread_cond_broadcast – broadcast a condition variable awaking all current waiting threads.

SYNOPSIS
#include “rtai_pthread.h”
int **pthread_cond_broadcast**(pthread_cond_t *cond);

DESCRIPTION
This function is used to broadcast a condition variable to all waiting threads, it should be used when there's more than one waiting thread able to respond to predicate change or if any waiting thread may be unable to respond.

RETURN VALUE
This function returns 0 if successful, otherwise an error code is returned as shown below.

ERRORS
If this function fails, it returns with one of the following error codes:

EINVAL: if the condition variable *cond* is invalid.

LXRT Functions (provided by module: lxrt)

rt_task_init

NAME
rt_task_init – create a real-time agent task for this Linux process.

SYNOPSIS
#include "rtai_lxrt.h"
RT_TASK ***rt_task_init** (int *name*,
int *priority*,
int *stack_size*,
int *max_msg_size*);

DESCRIPTION
When using LXRT, **rt_task_init** is used to create the real-time agent task for the current Linux process. The agent task is the one responsible for the hard real-time characteristics associated with LXRT's user space, real-time process.

RETURN VALUE
If successful, **rt_task_init** returns a pointer to the real-time agent task's task structure. A NULL pointer

is returned if the call fails.

ERRORS
None.