HOWTO Port your C++ GNU/Linux application to RTAI/LXRT

25/05/2004 : Updated to RTAI 3.0
21/10/2003 : Added a new chapter about RTAI/Fusion
25/07/2003 : Added extra info about signals
23/06/2003 : Completed section about accessing hardware and device drivers
14/04/2003 : Suggested fixes
09/04/2003 : Added introduction chapter
24/03/2003 : Incorporated community feedback
20/03/2003 : First version

Index

0. Introduction

0.1 Who is this document for ?

Since you stumbled on this page, it's most likely meant for you : a technical educated person with some Linux experience and programming in C or C++ and who wishes to build a realtime application with RTAI. It is very technical and quite short. It assumes you have knowledge about using IPC (Inter Process Communication) and threads, and have general knowledge about realtime systems.
The Introduction is quite general though, sorta.

0.2 What is LXRT and how does it relate to RTAI ?

LXRT is an extension on RTAI , the RealTime Application Interface developed at the DIAPM (Dipartimento di Ingegneria Aerospaziale - Politecnico di Milano) of Milan, Italy. RTAI provides an interface, schedulers and many features to make realtime programs in the GNU/Linux operating system. RTAI can accomplish this mainly in two independent ways : The RTHAL (Real-Time Hardware Abstraction Layer) which was in-house made and Adeos (Adaptive Domain Environment for Operating Systems), which was developed by Karim Yaghmour and Philippe Gerum and is supported by RTAI. The choice between both does not influence the RTAI or LXRT API in any way. Some people might say that Adoes is a safer alternative to be sure you don't violate any patents.

LXRT allows hard realtime programs to run in userspace (as opposed to kernel space). Since the initial RTAI programs ran in kernel space, LXRT is seen as an extension that brings this kernel API to userland programs. The advantages are significant. It allows to define realtime and not realtime threads in the same program so that one could read/write a file (= not realtime) while the other runs with a deterministic period (=realtime). It also allows IPC with standard Linux processes or allows your realtime program to have a GUI integrated. LXRT extensions allow to communicate with hardware realtime from these programs.
This document explains how to make such programs.

0.3 What is NEWLXRT ?

NEWLXRT is a term invented by Paolo Mantegazza to make the distinction with the old way of scheduling LXRT tasks. Previously, one needed to load a realtime (`rtai_sched_up, rtai_sched_smp, rtai_sched_mup`) scheduler for scheduling kernel tasks and an additional `rtai_sched_lxrt` scheduler for scheduling LXRT tasks. Thus you had three schedulers running : one for kernel realtime tasks, one for LXRT tasks and the original scheduler for normal Linux processes.

The NEWLXRT implementation, on the other hand, has one scheduler (`rtai_sched_newlxrt`) for kernel and lxrt tasks. When a NEWLXRT task is made hard realtime, its process is removed from the Linux process scheduler and inserted in the NEWLXRT scheduler, where the other hard realtime tasks reside. When the task is made soft realtime again, it goes back to the Linux scheduler. The pleatora of available schedulers and combinations might confuse some people, but generally, it is safe to say that you can use the NEWLXRT scheduler by default and investigate how the others work if that one does not work for you. In the end, the term NEWLXRT might be replaced by LXRT again to avoid confusion. From the application programmer's point of view, there is no difference between NEWLXRT and LXRT.

0.4 Is it slow ?

Efficiency is a main concern for most realtime programs. The overhead of scheduling a realtime task in userspace instead of kernelspace is however minimal. Recent measurements on modest hardware (Intel PII 300Mhz) showed that an additional 3us overhead on top of the standard scheduler latency of 3us is for most applications worth the benefits of LXRT. The noted overhead includes the following times : user-kernel-user roundtrip time and MMU switching.
The maximum jitter is approximately 20us on such a system and is dominated by the architectural jitter (it is common of the shelf hardware). Reports of disastrous jitters and latencies have always been caused by faulty hardware, most notably on laptops. The RTAI mailinglist archive contains a number of discussions on that topic.

Next to the additional scheduler latency, there are also extra latencies on RTAI API calls in LXRT. Using a semaphore, for example, in kernelspace will always be faster than in userspace. This is because some additional microseconds are lost because each LXRT call is transfered to the kernel and the result back to userland. A good starting place for measuring switching times on you own system are the examples/switches for kernel space timings and lxrt/switches for userspace timings. A test, using these programs, I performed while writing this document revealed that no more than 3us are lost for the whole trip for calling suspend/resume on tasks and semaphore signalling.

0.5 Credits

Ah, the usual thank you notes go to the following people : Paolo Mantegazza, Philippe Gerum, Steven Krekels , Thomas Leibner-Druska and Michael D. Kralka for comments on the document and the whole RTAI team.

1. Getting started

back to top

A lot of info can already be found in the RTAI package you can download from http://www.aero.polimi.it/projects/rtai/.

1.1 Read (in that order):

- rtai/INSTALL : Installation instructions
- rtai/rtai-doc : For a short overview of all features
- RTAI Wiki : The RTAI Wiki Page
- RTAI Documentation Project : The RTAI 3.0 Online Documentation

...for all necessary background information.

1.2 Recommended software set :

- GCC 3.2.2 or higher
- Standard, recent, binutils and make tools
- Linux 2.4.x kernel (vanilla is recommended)
- RTAI 3.0r3 or higher with LXRT enabled

1.3 What you find after installation

RTAI installs by default in the `/usr/local/realtime` directory. The lib subdirectory contains the lxrt library, the modules directory contains all the kernel modules you compiled, the bin directory should be in your path (it contains usefull scripts) and the include directory contains all the header files. The share directory contains the compiled RTAI documentation.

Before you can use LXRT, you must load the RTAI kernel modules, usually I have a script which does : `insmod /usr/local/realtime/modules/rtai_lxrt; insmod /usr/local/realtime/modules/rtai_sem; ...` '`dmesg`' should mention something about the LXRT scheduler. Be sure that no other scheduler is loaded (like up or smp) the lxrt scheduler schedules kernel realtime threads also.

To inspect the status of RTAI and Adeos, you can use the `proc` interface of Linux. Use `cat /proc/adeos` to check if adeos is present, use `cat /proc/rtai/*` to inspect all the RTAI information like the scheduler, LXRT processes and memory info.

2. Starting the port

back to top

Take any multithreaded C/C++ program. Make sure it compiles and links under GNU/Linux.

1. **Step 1 : Initialise your Linux Scheduler.**

   Initialising the Linux Scheduler as SCHED_FIFO (see `man sched_setscheduler`) improves your native linux scheduler.

```
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  struct sched_param mysched;

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

  return 0;
}
```

   This only has to happen once in your program, at the beginning. It can be replaced in Step 2 by the rt_task_init_schmod() function call, but this is optional. This function takes two extra arguments : the scheduler priority (usually SCHED_FIFO) and the processor on which the Linux process is allowed to run on. This is very handy on MP systems for locating the normal Linux processes on one CPU. Nevertheless, setting the scheduler to SCHED_FIFO is fully optional and just helps response times when testing your program in soft realtime mode.

2. **Step 2 : Make every thread (including main) an RTAI task (RT_TASK).**

   This will register the (posix) thread with the RTAI system and allow it to use the RTAI synchronisation and communication primitives (semaphores, mailboxes, shared memory etc...) This task will run by default in not-realtime. This however does not prevent you from using the traditional posix calls. **As long as your thread is not runnning hard realtime, any conventional function call is allowed.** You are

not obliged to make every thread in your program (*not even main()*) an RT_TASK. However, no harm is done doing so and it allows you to communicate with hard realtime tasks from a not realtime task and vice versa.

[**NOTE :***You MUST make your thread an RTAI task from the moment you use in that thread any RTAI call. Examples are* `pthread_create_rt,rt_sem_init,...` *with the notable exception of* `rt_task_init()`
RTAI/LXRT is an extension to your posix threads API, It is only logical that if you want to use these extensions, you need to register the thread with RTAI through `rt_task_init`.]

To proceed, include the file 'rtai_lxrt.h' Example :

```
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

#include <rtai_lxrt.h>

int main(void)
{
  RT_TASK* task;    // Stores a handle.
  int priority=0;   // Highest
  int stack_size=0; // Use default (512)
  int msg_size=0;   // Use default (256)

  /*-----------------------*/
  struct sched_param mysched;

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

  task = rt_task_init( nam2num("Name"), priority, stack_size, msg_size);
  /*-----------------------*/

  // your program ... anything you can think of is allowed here.

  rt_task_delete(task);
  return 0;
}
```

Every RT_TASK in LXRT needs a `"Name"` to which it can be referenced from kernel and userspace. This name may be no longer than 6 alfanumeric characters and must be unique. Having names is an easy way to identify tasks and communication primitives when the program is running.
If you want automatic generation of `"Name"` use the function `rt_get_name(0)` to get a unique, valid, thread safe, name. You can always check yourself with `rt_get_addr("Name")` to see if a name is already taken.

One might replace these calls for the main() program with the following LXRT helper function :

```
/*-----------------------*/
unsigned int processor_mask = 0xFF; // all processors

if (!(task = rt_task_init_schmod( nam2num("Name"), priority, stack_size, msg_size, SCHED_FI
  printf("CANNOT INIT TASK %u\n", taskname(task));
  exit(1);
}
/*-----------------------*/
```

This function initialises the scheduler and the main() task. The processor_mask restricts the process being scheduled on one or more processors using a binary pattern with 0x1 denoting processor 0.

To make a thread function an RTAI task, add the rt_task_init() and rt_task_delete() function calls at the start and end of that thread function. The scheduler only needs to be initialised once from main().

3. **Step 3: Compile and then run your program AS ROOT (see docs how to avoid this using rt_allow_nonroot_hrt()).**

   You will need to link with `/usr/local/realtime/lib/liblxrt.a` to resolve undefined symbols if you do not inline the RTAI function calls.

   While it runs, take a look at `/proc/rtai/scheduler` and `"/proc/rtai/RTAI Names"` to see your process running.

4. **Step 4: Lock your processes memory.**

   To avoid being swapped out, you have to lock your process in to RAM. Insert the following code before calling `rt_task_init()` :

```
#include <sys/mman.h>
//...

int main(void)
{
   // ...
   mlockall(MCL_CURRENT | MCL_FUTURE);
   // ...
}
```

Expanding the stack after mlock_all *will break hard realtime.* To anticipate stack expansion, there is a macro called `grow_and_lock_stack(granted_stack_size)` which you can use to allocate and lock the stack before going hard realtime.

5. **Step 5: Convert all the posix IPC calls to RTAI native calls.**
   To be clear : There are two ways to allow Inter Process Communication in RTAI :
   - The stable, least problems, but a bit more work way : use the native RTAI API
   - The untested, least effort, non compliant way : use the `_rt` extensions defined in `include/rtai_posix.h`.

   The first way is a bit more work since you have to learn a new API, using the rtai rt_* function calls to create, lock, message semaphores etc. However, they work as expected in **HARD and SOFT** realtime. This is a very important aspect.

   The second way is easier, since you only have to add _rt to each posix call and you're set for 90% of the job. However, the _param argument is not or rarely used. Take a look at the file `include/rtai_posix.h` and look up the functions `sem_init_rt` and `pthread_create_rt` and pay attention on how the arguments are used.

   > **Big Fat Warning :** You can **only** call the *_rt calls from **within an LXRT task**. Try else and you get kernel null pointers all the way. Secondly, you do not make LXRT tasks with `pthread_create_rt`. It just allows you to create a normal posix thread from hard/soft realtime. *You* have to make it an RTAI/LXRT task by using `rt_task_init(...)`.

6. **Step 6: Decide which thread will run hard realtime.**

   Until now all your processes ran soft realtime and were scheduled with the native Linux FIFO scheduler. To get the deterministic realtime scheduling of RTAI, you will need to do two things :

   1. Start the realtime timer (do this ONCE)
      `rt_start_timer()` : The rt_timer (or scheduler) programs the interrupt controller of the processor to generate periodic interrupts (periodic_mode) or reprograms the timer each time to fire when the next task must be started (oneshot_mode). Oneshot mode is less efficient but more flexible than periodic mode. This is further explained in the RTAI documentation.
      You can check if the timer is started with `rt_is_hard_timer_running()`. Stopping the timer is normally not necessary, unless you want to reconfigure it, which requires stopping all your hard realtime tasks.
   2. Make your RT_TASK periodic and hard realtime
      `rt_make_hard_realtime()` and `rt_make_soft_realtime()` : These calls make the calling thread switch from the Linux scheduler to the RTAI scheduler and back.

   Example:

```
int main(void)
{
  // ...
  // after rt_init_task

  if (oneshot)
    rt_set_oneshot_mode();
  else
    rt_set_periodic_mode();

  // <period_in_nanosecs> is the clock rate of the realtime scheduler
  // (rt_timer) in nano seconds.
  period = (int) nano2count((RTIME)period_in_nanosecs);

  start_rt_timer(period);

  // Periodic HardRT tasks are able to run now.
  // ...

}

{
  // ...
  // In a RT_TASK thread function or in main, after the timer is started

  if (hard_realtime) {
    rt_make_hard_real_time();
  }

  // this determines when rt_wait_period() will wake the first time.
  rt_task_make_periodic(hrt_task, rt_get_time() + period, period);

  while (continue)
  {
```

```
    // put periodic functionality here
    // ...
    rt_wait_period();
  }

  if (hard_realtime) {
    rt_make_soft_real_time();
  }

  // end of program/thread
  // ...
}
```

Official RTAI examples can be found in the RTAI showroom (only available through RTAI's CVS server). The `/usr/local/realtime/testsuite/` directory contain some basic tests to see if the primary components of RTAI work in user and kernel space. These programs must be started with the `rtai-load` script from the bin directory. Just type in 'rtai_load' in one of the program directories and it will start the program.

3 Potential problems

back to top

It's not really a FAQ, but this chapter discusses shortly some common problems.

3.1 General problems

- Allowed library calls during Hard Realtime
  Not all library calls can be called in your hard realtime process. The following rules apply:

  1. Any system call is not allowed.
  2. Any function call causing a process context switch is not allowed.

  RTAI might intercept these cases if you try anyway and make your task soft realtime. You can uncomment the `#define ECHO_SYSW` lines in `newlxrt/newlxrt.c` and `newlxrt/scheduler/rtai_sched.c` and recompile these files if you want to be notified when this happens. Your tasks will only return to hard realtime when it calls any RTAI function except rt_task_[init|delete].

- Compiling
  Make sure that your kernel, modules and LXRT application are compiled with the same compiler. Differences in compiler will cause a hard to trace crash.
- Linking
  When linking multiple object files together, multiple definitions might be present when KEEP_STATIC_INLINE is not defined. Only include rtai_lxrt_user.h to avoid this problem, since this file only includes the function definitions. In the end you need to link your application with -llxrt.
- Headers
  You should only include native GNU/glibc headers and the RTAI headers. **The linux source directory should never be in your path.** Use the normal userspace headers. Mixing kernel headers and userspace headers will give you a major headache. It *is* possible that RTAI includes some kernel headers, but the RTAI developers are working on cleaning them out on the long term anyway.

3.2 C++ specific problems

- While not thoroughly investigated, the `rtai_lxrt()` call seems to crash when compiled with the GNU 2.95 C++ compiler. This call is used to 'transport' userspace function calls to kernel space. See Chapter 5 for how this call is used in LXRT.
- The GNU C/C++ 3.2 compiler can handle all RTAI and kernel code. However, the -O3 optimization flag is suspect. To be safe, keep optimization level to -O2 or lower.
- The `rtai_cpp.o` RTAI extension is not needed for using LXRT. It is merely for limited C++ support in kernel modules.

3.3 Debugging your program

You can not debug a hard realtime program using gdb. All your threads need to be soft realtime. Even then, the debugger can get confused when the rtai_lxrt system calls are made. Sometimes, users notice a `segmentation fault`, start up the debugger and can not find the place where it crashed, even more, the stack seems heavily corrupted. When you experience this, it is very likely that your kernel crashed during the execution of the program, for example, by passing a null pointer to a system call. The corruption of the stack is then most likely caused by the kernel freaking out and not restoring your processes stack correctly, thus confusing your debugger. When you experienced a kernel OOPS, you have at least to unload the RTAI kernel modules and reload them before you can go on testing. Even then, rebooting might be more appropriate if you are not sure.

4 About threads, processes and signals in LXRT

back to top

You might wonder how a realtime LXRT process differs from a standard Linux process with respect to signals (like kill) and stack allocation.

- Signals : When a HRT LXRT task receives a signal, it should be forced back to SRT.
- Stack : The NEWLXRT RT_TASK has the same stack as the Linux process it is created in. This means you can leave the rt_task_init() thread argument to zero, it will be ignored. Instead the posix defaults, or which you specified, will be used

More on Signals

The above point about signals uses the verb 'should be' instead of 'is'. This is because the current LXRT (24.1.11) signals implemenation is broken. Michael D. Kralka explains this in deeper detail on the RTAI mailing list :

```
AFAICS, LXRT and Linux signals DO NOT play very well together. There
seems to be some hooks in LXRT to implement support for Linux signal
handling (see RT_SET_LINUX_SIGNAL_HANDLER in rtai_lxrt.h, tabnewlxrt.c,
and lxrt_table.c) but it is unimplemented.

Before delving into signals, make sure you are familiar with how signals
are processed (in particular with multiple threads - if your app is
multithreaded).

First and foremost, Sending SIGTERM or SIGKILL to a process kills it
ASAP, unless you have installed a signal handler (which you can't do for
SIGKILL). If a process is UNINTERRUPTABLE, the signal will be "pending"
until the task is no longer UNINTERRUPTABLE. When this happens, Linux
will handle the signal. HARD realtime LXRT processes and those that are
blocked on an RTAI object (such as a semaphore) are "owned" by RTAI and
_NOT_ Linux. This is done by putting the task into the UNINTERRUPTABLE
state.
# man ps
# ps aux
Should confirm this.

My suggestion
-------------

You might want to try the following:
- Add a signal handler to your LXRT app that captures the SIGTERM signal
(man signal or man sigaction). In this handler, do something that shuts
down your app (e.g. set some flag).
- Change the wait forever, with a timed way, say 10 seconds. If [your task is idle],
check if the app is being shutdown (e.g. the shutdown flag),
if so, cleanup, if not, continue with your loop.

Worst case, you app will shut down 10 seconds later. It that is too
long, shorten the time. This is how I handle the SIGTERM signal in my
app.

E.g.

void sighandler (int sig) {
shutdown_flag=1;
}

while (!shutdown_flag) {
/* [Do HRT timed wait or timed read] */
}
```

This text confirms what most of us have experienced lately : a HRT task can not be killed, until it is made SRT. One can safely assume in the above example that the parent (non-rt) thread is getting the signal while the child thread is the HRT task.

5 Accessing device drivers from LXRT : Extending LXRT

back to top

Realtime scheduling in userspace is a great advantage, but for most applications, the realtime constraint comes from interaction with hardware, with the outer world. It is not possible to use the linux device driver model from a hard realtime LXRT thread (but it is possible from soft or non realtime threads). You need to *extend* LXRT in order to access kernel device drivers. Extending LXRT is no longer documented in RTAI, but I made the old document available.

5.1 Hello Foo

To give you a headstart, the concept is as follows. Extending LXRT allows you to call the device driver functions from userspace as if you were in kernel space. Our first example is a device driver with a function

```
int hello_foo(int number)
{
  int i;
  for (i=0; i < number; ++i)
  printk("Hello\n");
  return 0;
}
```

5.1.1 The user side

You can equally call the above function from userspace hard realtime as from the kernel. The mechanism is that you make a headerfile `hello_foo_lxrt.h` which you include when compiling your application for LXRT (which is *not* a kernel module) and which defines an inlined *proxy* function hello_foo() which does the actual call to the kernel. It looks like this :

```
#include "rtai_lxrt.h"

/* one index for your module */
#define MYIDX 15

#define HELLO_FOO 0
#define HELLO_WORLD 10
#define BYE_WORLD   11
/* other function numbers here...*/

#ifndef __KERNEL__
int hello_foo(int number)
{
  struct {int n; } arg = { number };
  return rtai_lxrt(MYIDX, SIZARG, HELLO_FOO, &arg).i[LOW];
}

/* other functions here ... */
#endif
```

The struct `arg` contains an ordered list of the arguments of the function and is initialised by the function's arguments. This is a way for storing the function arguments so that they can be found back by the kernel function. *The struct must always contain at least one variable, a dummy if needed.* Next the `rtai_lxrt` call will cause to switch to the kernel context with the following arguments :

- MYIDX : the index of your extension ranging from 0 to 15. See the `rtai/lxrt/LXRT_EXTS_IN_USE` file for more information.
- SIZARG : A marco defined as sizeof(arg) (defined in rtai_lxrt.h)
- HELLO_FOO : a number defined by you, which is unique in MYIDX calls for identifying your function. This number can range from 0 to max(unsigned int), with gaps between number allowed.
- & arg : The address of the arg struct.
- i[LOW] : the return type of rtai_lxrt, which contains the return value of the kernel function, is a union. You can select the lower 32 bits integer part of the union as i[LOW]. The union can only return `RTIME rt, int i[2], void* v[2]` which is rather limited.

This proxy function will only be available when compiling for userspace, LXRT.

5.1.2 The kernel side

To make the call succeed in the kernel and ending up calling the real hello_foo() driver function, you need to create a special kernel module looking like this :

```
/**
* File hello_foo_lxrt.c
*/

#include <linux/module.h>
#include "hello_foo.h"
#include "hello_foo_lxrt.h"
#include <asm/rtai.h>

MODULE_LICENSE("GPL");

#define NON_RT_SWITCHING 0
#define RT_SWITCHING     1

/**
* The first parameter is the bitmask and denotes what marshalling must
* happen and if the call can cause realtime task switching,
* the second parameter is the function to be called.
*/
static struct rt_fun_entry rt_hello_fun[] = {
[ HELLO_FOO   ] = { NON_RT_SWITCHING, hello_foo },
[ HELLO_WORLD ] = { NON_RT_SWITCHING, hello_world },
[ BYE_WORLD   ] = { NON_RT_SWITCHING, bye_world },
};

/* init module */
int init_module(void)
{

  if( set_rt_fun_ext_index(rt_hello_fun, MYIDX)) {
    printk("Recompile your module with a different index\n");
    return -EACCES;
  }

  return(0);
}

/*  cleanup module */
void cleanup_module(void)
{
```

```
  reset_rt_fun_ext_index(rt_apci_fun, MYIDX);
}
```

This file is quite straightforward. It creates the table for mapping the numbers you defined in the header file back to the real function calls. What you do next, is load your device driver, then load the kernel module above, include the `hello_foo_lxrt.h` file in your LXRT program, and you are set to call the device driver from that program.

5.2 RT_Switch : Functions that may rt_schedule()

Assume that you want to use a kernel function in LXRT userspace that sleeps in the kernel on a semaphore. This might cause a switch to another realtime task, which will corrupt LXRT's functioning. You need to indicate this in the `struct rt_fun_entry` table. The first field (containing a zero in the above example) needs to be a 1. You can make this more readable by defining the NON_RT_SWITCHING variable to 0 and the RT_SWITCHING to 1. A save bet is always setting RT_SWITCHING, put this will cause performance loss.

5.3 More advanced parameter passing

Of course, you want not only to pass ints, but also `char*, void*,struct, double,...`. This is all possible. **BUT : memory is not shared between kernel and userspace : you can return a pointer to userspace but it has no meaning there.** You *can* however use it as a handle to refer to a struct which has been allocated in the kernel. The address contained in the pointer will remain valid in the kernel. This is often what we do instead of copying a struct from kernel to userspace, we pass the pointer as handler, since you mostly don't change the struct but only use it as an argument for the following function calls.
You return a pointer to a struct as follows :

```
#ifndef __KERNEL__

/** as opposed to the kernel version of struct MyDevice_t */
typedef MyDevice_t void;

MyDevice_t* getDevice()
{
  struct {int n; } arg = { 0 }; /** mandatory one (unused) element in your struct. */
  return rtai_lxrt(MYIDX, SIZARG, GET_DEVICE, &arg).v[LOW];
}

int turnOnDevice( MyDevice_t* dev )
{
  struct { MyDevice_t* dev; } arg = { dev };
  return rtai_lxrt(MYIDX, SIZARG, TURN_ON_DEVICE, &arg).i[LOW];
}

/* other functions here ... */
#endif
```

Which might seam a bit unsafe, but it works perfectly. The void pointer is to userspace just a handler to pass on to the kernel. This way you do not need to change your application as long as it keeps pointers to MyDevice_t, and does not try to modify it (which wouldn't compile, since it is a void pointer). If you really want the data from the struct, you can use the copy method described below.

5.4 Copying complex data structures

Until now, all we did was passing arguments by value, that is, we fill the arg struct with whole copies of the functions arguments. Since the arg struct is located on the stack, the rtai_lxrt function call can find it back and pass the values to the kernel function. If you want to retrieve information from the struct after the rtai_lxrt call, you need to use pointers in the struct pointing to a temporary placeholder struct, which must be copied in the end to the struct pointed at by the function's argument.
The trick behind copying is that all what is located on the stack can be read to or written from. Since function arguments are stored on the stack, copying by value (int, void*,...) is always automatically done. If you want to access something that is pointed at, you need to make a copy locally first.
There are two ways of copying data to the kernel and back : manual and automatic. Both can be considered equivalent, and you can use what you feel as most pleasant (or understandable :-).

5.4.1 Manual copying data structures

Manual copying data structures to kernel and back is done by using the mempcy function and a locally defined variable for each argument you want to copy on the heap, or copied by value if it is on the stack.

> There is no need to adapt the rt_fun_entry table when using this method.

The easiest case is a pass by value (located on the stack), which is :

```
typedef stuct MyDeviceData {/*... */} MyDeviceData_t;

/** User pass by value to Kernel **/
int setDeviceData(MyDeviceData_t devData )
{
  struct { MyDeviceData_t devData; } arg = { devData };
  return rtai_lxrt(MYIDX, SIZARG, SET_DEVICE_DATA, &arg).i[LOW];
}
```

As you can see, the struct is copied twice, once on function entry, and again on initializing arg. You can make this more efficient by doing a pass by pointer :

```
/** User pass by pointer to Kernel 2 **/
int setDeviceData(MyDeviceData_t* devData )
{
  struct { MyDeviceData_t devData; } arg = { *devData; }
  return rtai_lxrt(MYIDX, SIZARG, SET_DEVICE_DATA, &arg).i[LOW];
}
```

We manually copy once the data to the struct. As a reminder, you need to make sure that all data in the struct is located on the stack. Since this copy operation is frequently done, RTAI provides some macros that make automatic copying possible.

5.4.2 Automatic copying

LXRT allows to automatic copying to and from kernel space, within one function call. This mechanism is also described in the `README.EXTENDING_LXRT`. The point is that you provide additional elements in your struct (after the normal function arguments), that contain the size of the data to be copied. The `UW1(x,y), UR1(x,y)` macros are provided in the kernel to indicate that a copy operation to (UW) or from (UR) userspace must be made of structure element number `x` with size given in structure element number `y`. You need to place them in the type field of the function in rt_fun_entry struct, UWx / URx functions always assume RT_SWITCHING. These macros can be OR'ed to combine multiple copies in both directions. If your function already contains these numbers (for example `write(char* string, int len)`), you can naturally use `UW1(1,2)` and do not need to extend the struct, since the char* and int are already there.

> You MUST adapt the rt_fun_entry table when using this method.

Going forth with the previous example:

```
typedef stuct MyDeviceData {/*... */} MyDeviceData_t;

/**
*  Automatic user pass by pointer to Kernel
*/
int setDeviceData(MyDeviceData_t* devData )
{
  struct { MyDeviceData_t* devData; int size; } arg = { devData, sizeof(devData) };
  return rtai_lxrt(MYIDX, SIZARG, SET_DEVICE_DATA, &arg).i[LOW];
}

/**************************/

/**
* In kernel rt_fun_entry table :
*/
...
fun_table[SET_DEVICE_DATA] = { UR(1,2), setDeviceData},
...
```

Some examples from the README :

```
RT_TASK *rt_rpc (RT_TASK *task, unsigned int to_do, unsigned int *result);

has in the kernel table the macro UW1 :

{ UW1(3, 0), rt_rpc }

Which means that the third parameter is written to user (UW), with no size, defaulting
thus to 32 bits.

int rt_mbx_send (MBX *mbx, void *msg, int msg_size);

has in the kernel table the macro UR1 :

{ UR1(2, 3), rt_mbx_send }

Which means that the second parameter (msg) points to data of size (msg_size, third parameter)
LXRT will copy the data from userspace (UR) to kernel space.
```

Take a look at the existing programs for more examples. The UR2 and UW2 are the additional copy- instructions, if you want to copy two buffers in either direction. You need to OR these with the UR1 or UW1 marcros.

5.5 More examples and templates

To automatically copy a struct from userspace to kernelspace and vice versa :

```
typedef stuct MyDeviceData {/*... */} MyDeviceData_t;

/** User copy to Kernel **/
int setDeviceData(MyDeviceData_t* devData )
{
  struct { MyDeviceData_t* devData; int sz; } arg = { devData, sizeof(MyDeviceData_t) };

  /** We added an element, use UR1(1,2) in the kernel table **/

  return rtai_lxrt(MYIDX, SIZARG, SET_DEVICE_DATA, &arg).i[LOW];
}

/** Kernel copy to User **/
int getDeviceData( MyDeviceData_t* devData )
{
  struct { MyDeviceData_t* devData; int sz; } arg = { devData, sizeof(MyDeviceData_t) };

  /** We added an element, use UW1(1,2) in the kernel table **/

  return rtai_lxrt(MYIDX, SIZARG, GET_DEVICE_DATA, &arg).i[LOW];
}
```

6 RTAI/Posix - Fusion : FAQ

back to top

This Chapter briefly addresses some questions which newcommers might ask about RTAI/Posix - Fusion. They were posted on the RTAI mailinglist and kindly answered by Fusion designer Philippe Gerum. His project Xenomai creates an industrial-grade real-time Free Software platform for GNU/Linux.

1. What is RTAI/Posix - Fusion ?

   It is a further evolution of RTAI to integrate even more with existing Posix applications. It allows RTAI HRT and SRT tasks to invoke normal Linux system (and Posix) calls. This feature is currently only present in the Kilauea (testing) branch of RTAI. The Fusion interface defines a handful of functions which allow a userspace program to move to the RTAI scheduler and be scheduled as a realtime task. The Fusion interface will allow the program to do system calls without changing its priority. Also, the standard posix timing, like nanosleep(), gets a high precision implementation behind the scenes, running on the RTAI realtime clock.

   A detailed description of this mechanism can be found in the `rtai-doc/services/posix.tex` file of the Kilauea CVS branch. An online html version of this document can be found here.

2. Does it replace LXRT ?

   That depends. It does replace the LXRT way of scheduling RT/Non-RT threads. The Fusion philosophy of mixing Linux and RTAI domain threads is common with LXRT, but the problem is solved in a far more fundamental way. You could say that LXRT was the hack and Fusion is the solution.

   It does however not replace the LXRT system calls. Fusion does not introduce a new scheme for realtime system calls or LXRT function calls. The 'old' LXRT calls for synchronisation and messaging are, as for now, kept. As Fusion matures, some extra functionality might be added to replace awkward LXRT system calls.

3. Does it solve the debugging problem of LXRT, meaning, can a Realtime-or-Linux domain fusion application be debugged ?

   With RTAI/fusion, the higher level of integration between mere LXRT tasks and the Linux process space is obtained by seamless migrations of tasks between RTAI and Linux schedulers. Debugging situations will trigger RTAI -> Linux migrations (e.g. when recovering from faults, tracing, breakpoints etc.), and the plan is to work on a reliable way to do this (the other way around is needed to, but quite simpler).

4. How can I install an interrupt handler in Fusion userspace code, or if it has to be installed in kernel space, how do I communicate the data to and from ?

   I tend to think that our best shot here is to have specialized user-space task(lets) synchronizing on kernel events which could be posted from kernel space e.g. when an interrupt occurs. These interrupt service tasks in (user-space) would fire in turn the appropriate handler. This is what the existing USI module does basically, but we'd do this using fusion's generalized framework. The framework could provide for a kernel-based mechanism for intercepting the IRQ and upon receipt, post any given event defined through the user-space API.

   The other advantage of such approach would be to allow for system mutexes (maybe future RTAI's ones, not Linux's ones) to be used to protect interrupt-free sections in user-space, instead of enforcing them with actual interrupt masking. This would be made possible by the interrupt handling tasks in user-space having a heavy-weight context.

   The cost is more usecs to spend to switch contexts when an interrupt occurs to activate the heavy-weight IRQ task; what it buys us is no actual interrupt masking from user-space, and a price to pay only when an access contention is detected on a mutex. Nothing is defined yet, but it's a possibility.

   At the beginning, communication between user/kernel spaces should be no different than with LXRT, such as shmem, fifos etc. After that, we'll see.

5. What about scheduling a Fusion program from an interrupt handler ?

From a Linux interrupt handler, by using the standard wake up kernel routine. From a RTAI interrupt handler, like LXRT today, i.e. immediate switch on behalf of RTAI to the awaken task.

- If the RT task was treading in the mere LXRT space (kernel or user-space RTAI realm) before suspension, it simply resumes from its suspension/preemption point.
- If the RT task was suspended inside the Linux kernel space, it has the opportunity to resume when the next rescheduling point is reached by the next-to-be-preempted task. This is why having kpreempt + lolat inside the kernel is part of the solution btw, so that fine grained preemptibility in such a case is favourable to us.
- If the RT task was suspended in user-space under Linux's control, it has the opportunity to resume immediately since the long return path from interrupts is a regular rescheduling point.

6. Are my device drivers automatically 'RT'-enabled, by Fusion ?

   If speaking of Linux drivers, no, I can't say that. Fusion is using an Adeos-based interrupt shield to prevent Linux bottom halves/tasklets from preempting real-time tasks that tread on kernel code. Therefore, RTAI tasks which end up suspending themselves on a Linux kernel resource waiting for a Linux interrupt to occur (e.g. I/O wait) will be woken up only after the shield is deactivated. Since the deactivation happens when no other RTAI task is running into the Linux kernel domain, this is a source of priority inversions, since low priority RTAI tasks could prevent hi-priority ones to wake up until they relinquish the processor. What fusion grants though, is that your RTAI task executing kernel code won't be preempted by Linux-related activity (including async ones like tasklets) while fiddling with the driver code, upstream or downstream.

7. Can I use the normal userspace comedilib for example to do realtime measurements ?

   Since this is a matter of timing constraints, fusion will at least improve the overall determinism; one will still have to check that the guaranteed time bounds are short enough to fit his needs, though.

8. How do I port my device driver to Fusion, Is this the same way as porting to LXRT, or are these completely unrelated ?

   There won't be a specific 'fusion' way of doing things. The idea is to reuse the Linux code (kernel and user-space) as much as possible while keeping the determinism high (i.e. higher than with straight kpreempt and/or lolat patches alone), or relying on our usual LXRT way when we depend on very short delays. Then, the migration scheme between Linux and RTAI brought by fusion will greatly alleviate the burden of having two mutually-exclusive schedulers for managing a single RT task, from the application programmer POV.

9. Can I mix rtai system calls (like rt_malloc) with Fusion programs ?

   With fusion, you can even call glibc's malloc() from real-time code. This will work, correctly enforcing any POSIX mutex thanks to the proper migration between RTAI and Linux as needed.

   Syscalls mixing is available, and I would say, fundamental to the approach. For instance, the RTAI/vm system has a set of private syscalls controlling the virtual machine, and also issues regular Linux syscalls to create threads, get system memory and so on.

10. Are the Low-Latency patch and preemption patch obligatory or do they just decrease RT-latencies or what is the impact if they are not applied or on the contrary enhanced ?

    The impact if they are not applied is quite simple: if a RT task suspended inside the Linux kernel space needs to be rescheduled, the worst case for doing this will be...worse. Same for a real-time task previously running in the RTAI domain, which then needs to wait for the current Linux task to reach a rescheduling point before it can reenter the Linux space itself.

    On the other hand, if you don't need to reenter the Linux space when running in real-time mode (e.g. mere LXRT mode), you don't need these patches. For instance, a virtual pSOS machine hosted by the RTAI/vm subsystem probably doesn't need to issue Linux syscalls while performing.

11. Fusion is the next big thing, isn't it ?

    As 2.6 develops, we should expect to get finer preemptibility, now that it is a mainline feature, but probably never enough to have a very strict determinism, because this would imply an unacceptable performance trade-off between thoughput and determinism for Linux as a GPOS. And that's why IMHO the LXRT technology has an edge.

    However, I'm convinced that the dual kernel technique requiring strict isolation between Linux and components has reached a limit. The trend of complexity makes the constraints this approach imposes on the application design, harder and harder to cope with.

    A way to solve this is to leverage the work ongoing downstream with 2.6 to improve the Linux kernel preemptibility, by coupling to it the LXRT technology, in order get the right degree of determinism when it's needed. This way, we should be able to get the best of both worlds in terms of determinism and timing constraints, but without rebuilding all the Linux services aside of...Linux.

12. Is there a demo available ?

    Yes there is ! You can download a limited demonstration of a Fusion application right here. (You will need to install RTAI Kileau and compile the Xenomai nucleus, see the README files.)

13. What about signals ? Are they delivered to a RT-Fusion application ?

    In the LXRT approach, a signal was only delivered to the RT thread if it went back to the Linux domain. Is is not yet sure how they will be handled in Fusion. The ideal approach would be that a process can tell the kernel if it wants to react to signals or not. Reacting to a signal could mean for example that the task can be kill'ed, even if it is running in the RTAI domain.

To be continued ...


W3C
HTML
4.01