Building Your Way Through RTAI

Version 1 - 3/2008

João Monteiro
jmonteiro@alunos.deec.uc.pt

March 4, 2008

Abstract

This article's purpose is to present a robust and detailed guide to help people start working with RTAI, right after the installation process. With the help of fully explained practical examples, the readers will be able to start their way through building their own Real Time applications. The need to create concise documentation about this subject is well known, and the present text aims to fulfill this gap.

1 Introduction

After understanding and writing a guide about the RTAI installation, I wanted to take the next step: start working with RTAI. I found out that there were not many useful documents for real beginners, which made this step very hard. Questions like "What the heck is LXRT that those guys talk on the forums?" or "How can I make a task schedulable by the RTAI kernel layer?" or "Ok, now that I know this, but how can I start writing my code?" or even "What headers must I include, and how do I create a makefile with all the necessary flags to compile my app?" run over my head. Of course, there is a very good document that explains all of this – the RTAI 3.4 user manual –, but only after understanding the basics, one can look at it and fully understand what it has to offer. What I mean is, for a person who wants to start writing code with RTAI having absolutely no knowledge on the subject, he – or she – will find very few concise support for his level.

This guide will follow practical examples existant on the RTAI user manual to make the subject as clear as possible. Rather than having long theoretical explanations about the various RTAI functions, this text will be the stair for one to reach the top and then feel comfortable to learn more and more.

2 Knowing the Basics

What one first needs to understand, is that the main objective of RTAI is to provide a precise scheduler for tasks – other than the soft GNU/Linux scheduler –, making them be able to run at a pre-planned temporal fashion. With this, a task can activate at the precise needed time (hard real time), or with some acceptable delay (soft real time). Any project can be planned so that every single event on our program is predicted. To this we call predictability, which exists once we have a real time based system[1]. At this point, it's important to see RTAI as a kernel layer that lays down between the hardware and the Linux kernel, capable of superimposing the soft Linux kernel scheduler, to make objects defined as "real time tasks", run with the desired timing requisites.

One of the most important aspects to realize before starting to work with RTAI, is to know exactly how it can be used. RTAI can be used to program in Kernel Space and/or User Space, where the latter is most commonly designated as LXRT. RTAI possesses two types of schedulers: one that provides scheduling interaction between user tasks –processes or threads– and the kernel, and another that supports this, and also intra kernel task scheduling for light RTAI kernel tasks. The former is known as *rtai_lxrt*, and the latter, *rtai_sched* [1]. LXRT supports hard real time only for schedulable Linux objects that we are used to work with, namely threads and processes. The other scheduler provides hard real time for all Linux schedulable objects, and also to RTAI own kernel tasks. From a user space point of view, only one scheduler is used for the Linux schedulable objects. When in kernel space, another is used to make proper scheduling of RTAI kernel tasks, which are commonly used, for instance, if one needs to access a hardware board in no time. I like to think of this subject as follows: there exists only one RTAI scheduler capable of scheduling Linux objects when I declare in my C program that I want to use the LXRT schedule option (as we will further see), realizing that I'm programing in user space. This lonely scheduler is also capable of handling very light real time kernel tasks, when I also declare in my program that I wish to use the *rtai_sched*, knowing that, when I deal with it's function calls, I'm programming tasks to run within the kernel (kernel space programming). If in my program I don't need to have kernel tasks which possess the benefit of having an even more precise task activation – and are also more difficult to work with –, I simply don't activate this capability of the scheduler, programming all the time in user space. Simple.

[1] Personally, I recommend that any software project, specially those that possess real time capabilities, be modeled using the Unified Modeling Language 2.x before the implementation phase.

3 Prepare to Start

One important aspect that you will need to start developing Real Time apps with RTAI, is to understand the Linux PATH environment. Understanding this subject is very important, since you will have to help Linux know where to get the RTAI headers and executables from, when you want to compile your applications.

The PATH is simply a list of directories separated by colons ":", where files are looked for. This directories are searched by Linux to find files or commands. If one calls an application in a shell, all the directories of the PATH are searched in order to find the application with that name. The same goes to the process of compiling and linking an application: the header files are searched within the directories available in the PATH. To check the already loaded directories in your PATH, simply open a terminal and type the following command:

```
$ $PATH
```

noting that the $ character comes attached to the PATH call. You will get something like,

```
lite@lite~$ $PATH
  bash: /usr/local/sbin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/bin/X11
```

which tells you the directories that are identified by the PATH variable. For us to compile applications using RTAI function calls declared in the RTAI headers (*rtai_lxrt.h*, *rtai_sched.h*, among others), we need to include the RTAI *include* and *bin* directory. To accomplish this, and supposing you have installed RTAI in the default /usr/realtime dir, we have to run the following,

```
$ export PATH=$PATH:/usr/realtime/include
$ export PATH=$PATH:/usr/realtime/bin
```

and with this, we are adding the necessary directories to the current PATH environment directory list $PATH, separated with the standard separator character ":". You can also accomplish this in one line of code,

```
$ export PATH=$PATH:/usr/realtime/include:/usr/realtime/bin .
```

The boring part here, is that, once rebooted, Linux doesn't store the added directories, and you have to export them every time you boot. One thing that we can do to fix this, is to add the above instructions to the boot process. For this, open the *profile* file with a text editor (like mousepad) having root privileges,

```
$ sudo -s
# mousepad /etc/profile.
```

and now, at the end of the document, add the following lines:

```
PATH=$PATH:/usr/realtime/include
export

PATH=$PATH:/usr/realtime/bin
export
```

and you are done. Every time you start your computer and login with your user name – not root –, you can type $PATH and see the RTAI directories already added.

4 The First Real Time Application

The time has come to put our hands on the code. I have created a simple application in which I will rely to explain step by step the basics of RTAI, in order for you to understand how a task is created, how the scheduler works, etc..

4.1 The API

To develop my code, I first wanted to use KDevelop, since it has been my programming API of choice for years. After spending significant time on forums and googling, I realized that the effort to develop RTAI apps with KDevelop was not worth it, since we have Kate. This simple text editor is more than enough and, sincerely, I started to like it a lot. It has the same color scheme of KDevelop, and it's very simple to use. To get it, if you use a Debian based distro – which I assume you do since you may have read my RTAI installation guide :) –, do the following,

```
$ sudo apt-get install Kate
```

and then you can open it by just typing Kate at the command line. The problem is that we have to create ourselves the makefiles. But don't worry, this is kind of fun for small projects. Nevertheless, I intent to continue my struggle on porting my real time apps to KDevelop later on..

4.2 Start Coding

Open Kate, and create a new file. Save it as *rt_skeleton.c* on a folder dedicated to this mini project. Next, create a new file, and save it as globals.h on the same folder. Finally, create a new one, name it functions.c, and save it in the same folder.

The globals.h file is the first to concern us. It will possess, among other things, the necessary header inclusion for our project. Since we are going to program in user space, we will not need to include *rtai_sched.h* but *rtai_lxrt.h*. So, in *globals.h*, include the latter header, which is saved in the folder /usr/realtime/include that we early added to the path.

```
#include <rtai_lxrt.h>
```

This header possesses all the necessary instruction declarations that we will need for this mini project. Programming in user space is very straightforward after we know how to do the basics.

We will also need to use Linux POSIX threads and standard Input/Output for printf(), so lets also include the necessary headers, making the *globals.h* file look like this,

```
// globals.h file -- Contains the global declarations and necessary header includes.
//
#include <rtai_lxrt.h>
#include <pthread.h>
#include <stdio.h>
```

Now, switch to the *rt_skelet.c* file, which is where the main() function will be. To start, we need to include the local *globals.h* file, and then initialize the main function. So, let's place the first bone of our skeleton program, making this file look like this,

```
// rt_skelet.c -- Contains the main() function, where the magic starts.
//
#include "globals.h"

int main(int argc, char *argv[])
{
        //The main function code will go here.
}
```

where argc is the number of input arguments, and each component of the char array of pointers argv[] will point to the memory address of the first char of each input argument. By calling argv[0] and printing it as string %s within a printf, you will get something like ./test, in case the linked final file is called test.

Ok, let's declare some variables that we will need.

```
// rt_skelet.c -- Contains the main() function, where the magic starts.
//
 #include "globals.h"

int main(int argc, char *argv[])
{
  // Local Main Declarations
  int hard_timer_running = 1;
  static RTIME sampling_interval;

  //The main code will go here.
}
```

The first variable, *hard_timer_running* is a flag that will be necessary to check if a hard real time timer is running, before starting our app. *sampling_interval* will contain the period of a task that will further be created, and it's declared as static, so its reserved memory region can never be changed by functions declared elsewhere.

Now, we will initialize a simple flag that we will declare on our globals.h file as not static. This variable will be easily accessible to all of the tasks that we will create. So, let's change the *globals.h* and *rt_skelet.c*.

```
 // globals.h file -- Contains the global declarations and necessary header includes.
 //
 #include <rtai_lxrt.h>
 #include <pthread.h>
 #include <stdio.h>

 int keep_on_running;
```

and,

```
// rt_skelet.c -- Contains the main() function, where the magic starts.
//
 #include "globals.h"

int main(int argc, char *argv[])
{
  // Local Main Declarations
  int hard_timer_running = 1;
  static RTIME sampling_interval;

  keep_on_running = 1; // This flag will have the power of terminating the app!
}
```

For the next step, we will need to initialize the real time timer, which is responsible of triggering the Real Time events of our application, and declare the sampling interval of the periodic task that will run aside of the main task.

```
 // globals.h file -- Contains the global declarations and necessary header includes.
 //
 #include <rtai_lxrt.h>
 #include <pthread.h>
 #include <stdio.h>

 int keep_on_running;

 #define TICK_TIME 3E9 // Three seconds
```

and,

```
// rt_skelet.c -- Contains the main() function, where the magic starts.
//
 #include "globals.h"

int main(int argc, char *argv[])
{
  // Local Main Declarations
  int hard_timer_running = 1;
  static RTIME sampling_interval;

  keep_on_running = 1; // This flag will have the power of terminating the app!

  // Initialize the timer
  if ( (hard_timer_running == rt_is_hard_timer_running() ))
  {
        printf(" Skip Hard Real Time Setting\n");
        sampling_interval = nano2count(TICK_TIME); //Converts a value from
                                                   //nanoseconds to internal
                                                   // count units.
  }
  else
  {
        printf("Starting Real Time Timer...\n");
        rt_set_oneshot_mode();
        start_rt_timer(0);
        sampling_interval = nano2count(TICK_TIME); // Sets the period of the concurrent task
                                                   // that will be launched later.
  }
}
```

It is critically important to initialize the timer only once, and it is equally important not to stop the timer when some task exits, otherwise the timer will be stopped for all the tasks. If the timer is initialized more than once, the second call will reset the timer and start it with the new period. The *rt_is_hard_timer_running*() call, will verify if a hard real time timer is running and, in our code, if it is, the timer is not initialized again, and only the *sampling_interval* variable will be updated to fit our needs. If no timer is running, the type of timer is set to *oneshot* which allows tasks to be timed arbitrarily. This means that we can have periodic and aperiodic/sporadic tasks running in our program. At last, the timer is initialized with the call *start_rt_timer*(0).

Now, its time to make our main process a real time schedulable task. This is done with the call *rt_task_init_schmod*(), which is a more complex version of *rt_task_init*(). Both can be used, but the former allows us to define more specific parameters about the real time task. One might note now that, in fact, the process does not start as real time. Yes, it's correct. Only after we make this function call, the process can be schedulable by the RTAI kernel layer instead of the Linux kernel. So, let's declare the main task *my_task* as real time in the globals.h, and then add the call to the main() function.

```
 // globals.h file -- Contains the global declarations and necessary header includes.
 //
 #include <rtai_lxrt.h>
 #include <pthread.h>
 #include <stdio.h>

 int keep_on_running;

 #define TICK_TIME 3E9 // Three seconds

 static RT_TASK *my_task; // Main task will be Real time schedulable!
```

and,

```
// rt_skelet.c -- Contains the main() function, where the magic starts.
//
 #include "globals.h"

int main(int argc, char *argv[])
{
  // Local Main Declarations
  int hard_timer_running = 1;
  static RTIME sampling_interval;

  keep_on_running = 1; // This flag will have the power of terminating the app!

  // Initialize the timer
  if ( (hard_timer_running == rt_is_hard_timer_running() ))
  {
        printf(" Skip Hard Real Time Setting\n");
        sampling_interval = nano2count(TICK_TIME); //Converts a value from
                                                   //nanoseconds to internal
                                                   // count units.
  }
  else
  {
        printf("Starting Real Time Timer...\n");
        rt_set_oneshot_mode();
        start_rt_timer(0);
        sampling_interval = nano2count(TICK_TIME); // Sets the period of the concurrent task
                                                   // that will be launched later.
  }

 // Make the main process schedulable by the RTAI kernel layer
 if (!(my_task = rt_task_init_schmod(nam2num( "MAINATSK" ),   // Name
                                     1,                      // Priority
                                     0,                      // Stack Size
                                     0,                      // max_msg_size
                                     SCHED_FIFO,             // Policy
                                     0 ) ) )                 // cpus_allowed
  {
        printf("ERROR: Cannot initialize main task\n");
        exit(1);
  }
}
```

The parameters of the *rt_task_init_schmod*() call are very straightforward. We make its priority

"1", and make it also schedulable by linux with the policy $SCHED\_FIFO$ (default scheduling policy of the linux kernel). This last part might be tricky, but remember that the main process is, at the beginning, a Linux schedulable object. Setting the policy is needed just to have a quicker Linux response during the soft period right before the process becomes a real time task. We can set the Linux scheduling policy to Round Robin – $SCHED\_RR$ – in case we have a lot of concurrent tasks, which will not be the case for now.

The time has come to launch another task to run concurrently with the main task. This will help us to understand how the RTAI scheduler works. So, let's declare a *pthread_t* variable in our globals for our new POSIX thread (latter an RTAI schedulable object), and also declare the thread function in globals:

```
// globals.h file -- Contains the global declarations and necessary header includes.
//
#include <rtai_lxrt.h>
#include <pthread.h>
#include <stdio.h>

int keep_on_running;

#define TICK_TIME 3E9 // Three seconds

static RT_TASK *my_task; // Main task will be Real time schedulable!

static pthread_t main_thread; // Points to where the thread ID will be stored

void *main_loop(void *args); // Thread launched by the main task.
```

And now let's launch the thread passing the local *sampling_interval* variable as argument – to make things a little bit more interesting :) –, and create a loop for the main function, controlled by the global *keep_running* variable. We can also now conclude the main() routine by placing the termination command that will self kill the main real time task upon exit. Now, we are done with *rt_skelet.c*.

```
// rt_skelet.c -- Contains the main() function, where the magic starts.
//
 #include "globals.h"

int main(int argc, char *argv[])
{
  // Local Main Declarations
  int hard_timer_running = 1;
  static RTIME sampling_interval;

  keep_on_running = 1; // This flag will have the power of terminating the app!

  // Initialize the timer
  if ( (hard_timer_running == rt_is_hard_timer_running() ))
  {
        printf(" Skip Hard Real Time Setting\n");
        sampling_interval = nano2count(TICK_TIME); //Converts a value from
                                                   //nanoseconds to internal
                                                   // count units.
  }
  else
  {
        printf("Starting Real Time Timer...\n");
        rt_set_oneshot_mode();
```

```
        start_rt_timer(0);
        sampling_interval = nano2count(TICK_TIME); // Sets the period of the concurrent task
                                                   // that will be launched later.
  }


 // Make the main process schedulable by the RTAI kernel layer
 if (!(my_task = rt_task_init_schmod(nam2num( "MAINATSK" ),   // Name
                                     1,                      // Priority
                                     0,                      // Stack Size
                                     0,                      // max_msg_size
                                     SCHED_FIFO,             // Policy
                                     0 ) ) )                 // cpus_allowed
  {
        printf("ERROR: Cannot initialize main task\n");
        exit(1);
  }

  pthread_create(&main_thread,NULL,main_loop, (void *)sampling_interval);

  while(keep_on_running)
  {
        // Do nothing, let the concurrent task do its work.
  }

  // Program termination
  rt_task_delete(my_task);
  printf( "Program %s will be finished!\n", argv[0] );
  return( 0 );
}
```

What we still need to do, is produce the code for the concurrent task. So now, open the *functions.c* file and let's start to code it. Let's, for now, include the *globals.h* file that will also be needed by this new task, and create the skeleton of the pthread function that we have earlier placed in the globals.h.

```
// functions.c -- The function that will run concurrently to the main task.
//
 #include "globals.h"

 void *main_loop(void *args)
 {
        // The concurrent task code will go here.
 }
```

We can see that the function receives a pointer as argument. In our case, it's a void pointer to the *sampling_interval*'s variable memory location. This is needed, since we will need to access this location to set this task's loop time (Yup, it will be periodic!). Now, let's fetch the sampling interval from the function argument passed over the *pthread_create*(), and make this Linux schedulable object a real time task, by first declaring the new real time task in the globals.h, and then call *rt_task_init_schmod*().

```
 // globals.h file -- Contains the global declarations and necessary header includes.
 //
 #include <rtai_lxrt.h>
 #include <pthread.h>
 #include <stdio.h>
```

```
 int keep_on_running;

 #define TICK_TIME 3E9 // Three seconds

 static RT_TASK *my_task; // Main task will be Real time schedulable!

 static pthread_t main_thread; // Points to where the thread ID will be stored

 void *main_loop(void *args); // Thread launched by the main task.

 static RT_TASK *loop_Task; // Concurrent task
```

and,

```
// functions.c -- The function that will run concurrently to the main task.
//
 #include "globals.h"

 void *main_loop(void *args)
 {
        RTIME sampling_interval = (RTIME *) args;
        int i = 0;
        unsigned char temp,data;

        // Make the thread RT schedulable.
        if (!(loop_Task = rt_task_init_schmod(nam2num( "RTAI01" ),   // Name
                                      2,                     // Priority
                                      0,                     // Stack Size
                                      0,                     // max_msg_size
                                      SCHED_FIFO,            // Policy
                                      0 ) ) )                // cpus_allowed
  {
        printf("ERROR: Cannot initialize main task\n");
        exit(1);
        }

}
```

Note that this task will possess higher priority than the main task, so it can superimpose it when it's activated. Since we have a little number of tasks running, we will keep the Linux scheduling policy the same as before, and leave the other fields as default.

Now, to play with things a little bit, lets make this task periodic, and hard real time. For this, we will have to call the *rt_task_make_periodic*() function, and also *rt_make_hard_real_time*(). The former call accepts as parameters the task ID to became periodic, the expected start time, and the sampling interval. The functions.c file will now look like this,

```
// functions.c -- The function that will run concurrently to the main task.
//
 #include "globals.h"

 void *main_loop(void *args)
 {
        RTIME sampling_interval = (RTIME *) args;
        static RTIME expected; // The expected start time of the task.
        int i = 0;
        unsigned char temp,data;
```

```
        // Make the thread RT schedulable.
        if (!(loop_Task = rt_task_init_schmod(nam2num( "RTAI01" ),   // Name
                                    2,                      // Priority
                                    0,                      // Stack Size
                                    0,                      // max_msg_size
                                    SCHED_FIFO,             // Policy
                                    0 ) ) )                 // cpus_allowed
        {
        printf("ERROR: Cannot initialize main task\n");
        exit(1);
        }

        // Let's make this task periodic..
        expected = rt_get_time() + sampling_interval;
        // ..with start time imposed by the variable expected.
        rt_task_make_periodic(loop_Task, expected, sampling_interval); //period in counts
        // And now it will become hard real time.
        rt_make_hard_real_time();
}
```

To evaluate the precision of the hard real time, we will include a fancy loop to check for the time between activations. This task will end after five loops,

```
// functions.c -- The function that will run concurrently to the main task.
//
 #include "globals.h"

 void *main_loop(void *args)
 {
        RTIME sampling_interval = (RTIME *) args;
        static RTIME expected; // The expected start time of the task.
        int i = 0;
        unsigned char temp,data;

        // Make the thread RT schedulable.
        if (!(loop_Task = rt_task_init_schmod(nam2num( "RTAI01" ),   // Name
                                    2,                      // Priority
                                    0,                      // Stack Size
                                    0,                      // max_msg_size
                                    SCHED_FIFO,             // Policy
                                    0 ) ) )                 // cpus_allowed
  {
        printf("ERROR: Cannot initialize main task\n");
        exit(1);
}

        // Let's make this task periodic..
        expected = rt_get_time() + sampling_interval;
        // ..with start time imposed by the variable expected.
        rt_task_make_periodic(loop_Task, expected, sampling_interval); //period in counts
        // And now it will become hard real time.
        rt_make_hard_real_time();

        // Concurrent function Loop
        while(1)
```

```
        {
                i++; // Count Loops.
                printf("LOOP -- Period time: %f %f\n",(float)rt_get_time_ns()/1E9 -
                       old_time,count2nano((float)sampling_interval)/1E9);
                old_time = (float)rt_get_time_ns()/1E9;
                if (i== 5)
                {
                        keep_on_running = 0;
                        printf("LOOP -- run: %d %d\n ",keep_on_running,&keep_on_running);
                        break;
                }
                rt_task_wait_period(); // And waits until the end of the period.
        }
}
```

The *rt_task_wait_period*() call is very important, since it will suspend the task until its next activation time, freeing the CPU to process other tasks in the meanwhile. If you, for example, place and instruction after this call, it will only execute at the beginning of the next period. Feel free to place a printf or something like it to test. Now, we can complete the task without forgetting to terminate it before the thread ends. And our program is complete.

```
// functions.c -- The function that will run concurrently to the main task.
//
 #include "globals.h"

 void *main_loop(void *args)
 {
        RTIME sampling_interval = (RTIME *) args;
        static RTIME expected; // The expected start time of the task.
        int i = 0;
        unsigned char temp,data;

        // Make the thread RT schedulable.
        if (!(loop_Task = rt_task_init_schmod(nam2num( "RTAI01" ),   // Name
                                    2,                      // Priority
                                    0,                      // Stack Size
                                    0,                      // max_msg_size
                                    SCHED_FIFO,             // Policy
                                    0 ) ) )                 // cpus_allowed
        {
        printf("ERROR: Cannot initialize main task\n");
        exit(1);
        }

        // Let's make this task periodic..
        expected = rt_get_time() + sampling_interval;
        // ..with start time imposed by the variable expected.
        rt_task_make_periodic(loop_Task, expected, sampling_interval); //period in counts
        // And now it will become hard real time.
        rt_make_hard_real_time();

        // Concurrent function Loop
        while(1)
        {
                i++; // Count Loops.
                printf("LOOP -- Period time: %f %f\n",(float)rt_get_time_ns()/1E9 -
```

```
                        old_time,count2nano((float)sampling_interval)/1E9);
                old_time = (float)rt_get_time_ns()/1E9;
                if (i== 5)
                {
                        keep_on_running = 0;
                        printf("LOOP -- run: %d %d\n ",keep_on_running,&keep_on_running);
                        break;
                }
                rt_task_wait_period(); // And waits until the end of the period.
        }
rt_task_delete(loop_Task); //Self termination at end.
return 0;
}
```

5 Build the Application

Now it's time to build our application. For this, we need to compile and link the code we have created. To help us with this task, we can create a makefile, making us simply run the *make* command within the application folder every time we want to build it.

Create a new file in Kate, call it Makefile (without extension) and save it in the same folder of the other three code files. We will use gcc (which I suppose you already have), and include the necessary flags to compile our application. The Makefile should look something like the following,

```
 sample: rt_skelet.o functions.o
        gcc -L /usr/realtime/lib -lpthread  -O2 -I /usr/realtime/include rt_skelet.o
            functions.o -o sample

rt_skelet.o: rt_skelet.c functions.c globals.h
        gcc -c -L /usr/realtime/lib -O2 -I /usr/realtime/include rt_skelet.c

functions.o: functions.c globals.h
        gcc -c -L /usr/realtime/lib -O2 -I /usr/realtime/include functions.c

clean:
        rm *.o
        rm sample
```

where the -O2 flag is needed so that the RTAI headers are recognized. You can check the necessary flags present in the above Makefile by typing *rtai-config - -lxrt-cflags* and *rtai-config - -lxrt-ldflags* on a terminal.

Now, open a terminal, *cd* to your mini project folder, and run *make*. This will create an executable file called sample.

6 Run the Application

Finally, we will need to run the application. Rather than just simply run ./sample, we need to load it with an rtai script located in /usr/realtime/bin, called *rtai-load*. So, in a terminal, do the following,

```
 $ rtai-load sample
```

You then will be asked for root password so that the script can load the necessary RTAI modules to the kernel, and voilà! If everything is OK, you will have an output like,

```
$ rtai-load sample
 Password:
 Starting Real Time Timer...
 134520592 3579540
MAIN -- run: 1 134520632
 MAIN -- run: 1 134520632
 LOOP -- Period time: 1253.943984 3.000000
MAIN -- run: 1 134520632
 MAIN -- run: 1 134520632
 MAIN -- run: 1 134520632
 LOOP -- Period time: 2.999747 3.000000
MAIN -- run: 1 134520632
 MAIN -- run: 1 134520632
 MAIN -- run: 1 134520632
 LOOP -- Period time: 2.999870 3.000000
MAIN -- run: 1 134520632
 MAIN -- run: 1 134520632
 MAIN -- run: 1 134520632
 LOOP -- Period time: 2.999871 3.000000
MAIN -- run: 1 134520632
 MAIN -- run: 1 134520632
 MAIN -- run: 1 134520632
 LOOP -- Period time: 2.999869 3.000000
LOOP -- run: 0 134520632
 MAIN -- run: 0 134520632
 Program ./sample will be finished!
 $
```

which will show the time between activations when the loop concurrent task activates as expected! :)

7 Conclusion

This text was created to help people start their way through RTAI. I hope that you fully understand everything, since I made an effort to write and explain the subject as clear as possible. Note, however, that this text is still under development, which means that if you detect a bug, or have any questions or doubts about what has been stated, please e-mail me.

Happy coding,
JMonteiro

References

[1] *RTAI 3.4 User Manual, rev 0.3*, chapter 5.