RTOS, Spring 2015 – Lab #6: RTAI Tasks

Paolo Torroni, paolo.torroni@unibo.it
Davide Chiaravalli, davide.chiaravalli@studio.unibo.it

Objective: to learn how to release tasks in RTAI.

1. Background

Make sure you have understood Lab #1: RTAI and Linux kernel modules. Then use the following text as a reference throughout.
(source: http://www.cs.ru.nl/lab/rtai/)

Scheduling in RTAI Linux

- The real-time scheduler is started together with the real-time timer by the command *start_rt_timer.*
- In RTAI Linux, the real-time scheduler is driven by timer interrupts from the PC's 8254 Programmable Interval Timer (PIT) chip, which can be programmed to generate timing interrupts at certain moments. (This chip is also used by the normal Linux scheduler.) The resolution is based on frequency of the 8254 chip frequency (1,193,180 Hz), i.e., there is tick of the 8254 chip every 838 nano seconds. In addition we can use the Time Stamp Clock (TSC) of the PC, based on an internal crystal which drives the PC.
- Each separate CPU has its own real-time timer and each can run in ***periodic* or *one-shot* mode** independent of all the other CPU's. *Internal count units* measure the amount of elapsed time; how this is done depends on the mode. The type of these internal count units is RTIME.

- In **pure periodic mode**:
  - The internal count units are the ticks of the 8254 chip.
  - The parameter of the *start_rt_timer* is used to specify a so-called *base period* in terms of internal count units (i.e., 8254 ticks). The 8254 PIT is programmed to expire repeatedly after the specified amount of internal counts, providing interrupts with the frequency of the base period. **All tasks will run at a multiple of this base period.**
  - The function *rt_get_time()* returns the numbers of internal count units (i.e., 8254 ticks) passed since *start_rt_timer* was called.
  - The advantage is that timer reprogramming is avoided, which saves about 3 microseconds.
  - The disadvantage is that all tasks are forced to run at multiples of the base period.

- In **one-shot mode**:
  - The **internal count units used equal the TSC count**.
  - The 8254 PIT is reprogrammed at the end of each task, such that it expires exactly when the next task is scheduled to run.

- Function *rt_get_time()* again returns the numbers of internal count units, now this corresponds to the TSC count.
- The advantage is that the **timing of tasks can vary dynamically** and new tasks can be added without having to bother about any base period.
- The disadvantage is repeated timer reprogramming overhead. Notice that the timer will always run on a frequency lower than the TSC of the CPU, but the TSC frequency is usually much higher than that of the 8254 chip (300 times for a 300 MHz CPU).

Starting the Timer and Realtime Scheduler

The first thing we always have to do at the start of a real-time program is to start the real-time timer and the scheduler using the *start_rt_timer* command.

- We start the timer to expire in **pure periodic mode** by calling the functions

```
void rt_set_periodic_mode(void);
RTIME start_rt_timer(RTIME period);
```

  The value passed to 'start_rt_timer()' in periodic mode is the, above mentioned, desired base period, specified in internal counts (i.e. 8354 ticks). The return value of type RTIME yields the realized number of counts.

- We start the timer to expire in **one-shot mode** by calling the functions

```
void rt_set_oneshot_mode(void);
RTIME start_rt_timer(RTIME period);
```

  The **argument for start_rt_timer is chosen randomly** because we don't have a period, and it will therefore be ignored by rt_start_timer. So we can for instance use start_rt_timer(1).

The function

```
RTIME nano2counts(int nanoseconds);
```

can be used **to convert time in nanoseconds to these RTIME internal count** units. In periodic mode this is quantized to the resolution of the 8254 chip frequency (1,193,180 Hz).

The Task Function

Each task is associated with a function that **is called when the task is scheduled to run**. This function is a usual C function running in period mode that typically reads some inputs, computes some outputs and waits for the next cycle. Such a task function should enter an endless loop, in which is does its work, and then calls

```
void rt_task_wait_period(void);
```

to **wait for its next scheduled cycle**.

Typical code looks like this:

```
void task_function(int arg) {
   while (1) {
      /*       Do your thing here      */
      rt_task_wait_period();
   }
return;
}
```

Setting Up the Task Structure

- An **RT_TASK data structure** is used to hold all the information about a task.
- The task structure is **initialized** by calling

```
rt_task_init(
    RT_TASK *task,
    void *rt_thread,
    int data,
    int stack_size,
    int priority,
    int uses_fp,
    void *sig_handler);
```

  - 'task' is a pointer to an RT_TASK type structure which must have been declared before and whose structure is filled.
  - **'rt_thread' is the entry point of the task function.**
  - 'data' is a single integer value passed to the new task.
  - 'stack_size' is the size of the stack to be used by the new task.
  - 'priority' is the priority to be given the task. The highest priority is RT_SCHED_HIGHEST_PRIORITY (which equals 0), while the lowest is RT_SCHED_LOWEST_PRIORITY (which equals 1,073,741,823) (both are defined in rtai_sched.h).
  - 'uses_fp' is a flag. Nonzero value indicates that the task will use floating point, and the scheduler should make the extra effort to save and restore the floating point registers.
  - 'sig_handler' 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 suspended state**. It can be made active either with 'rt_task_make_periodic()', 'rt_task_make_periodic_relative_ns()' or 'rt_task_resume()'.

Scheduling the Task

periodic mode

- The task can now be started by passing a pointer to the initialized RT_TASK structure to the function

```
int rt_task_make_periodic(
    RT_TASK *task,
    RTIME start_time,
    RTIME period);
```

- 'task' is the address of an earlier initialized RT_TASK structure,
- 'start_time' is the absolute time, in RTIME units, when the task should begin execution. Typically this is "now" (i.e., calling 'rt_get_time()').
- 'period' is the task's period, in RTIME units, which will be rounded to the nearest multiple of the base period.

- The task will now execute indefinitely every 'period' counts.

one-shot mode

- The task can be simple start immediately by calling the function

```
int rt_task_resume(RT_TASK *task)
```

- However the task can be **started after a delay** with

```
int rt_task_make_periodic(
    RT_TASK *task,
    RTIME start_time,
    RTIME period);
```

- 'task' is the address of the RT_TASK structure,
- 'start_time' is the absolute time, in RTIME units, when the task should begin execution. Typically this is "now" (i.e., calling 'rt_get_time()').
- 'period' is now a dummy value which is **not used by the scheduler in one-shot mode**

Clean up

Always remember to clean up by deleting the tasks and the timer! For example:

```
void cleanup_module(void)
{
    stop_rt_timer();
    rt_task_delete(&first_task);
    rt_task_delete(&second_task);
}
```

2. Aperiodic Task activation

Define two tasks, tau_1 and tau_2, and activate them at different times
You can download an example source file at
http://lia.deis.unibo.it/RTOS
NOTICE: before you install the modules of the example, you should install the RTAI modules:
/usr/realtime/modules/rtai_hal.ko and
/usr/realtime/modules/rtai_sched.ko

3. Periodic task activation

Modify the code, in order to turn tau_1 and tau_2 into periodic tasks.