Real Time Operating Systems Design and Programming

LAB 2

OS Task Lab

RTX Basics

Contents

1 Overview

In this lab you will configure the RTX and create and destroy some tasks using the RTX RTOS. You are expected to walk through how the OS performs task creation, termination and switching operations. Additionally, this lab will involve some concepts from the Scheduler Lecture - don't panic if you do not yet understand some of the advanced features of the RTX. More specific real time OS features and advanced embedded software development topics will be covered in later lectures and labs.

2 Setting up RTX

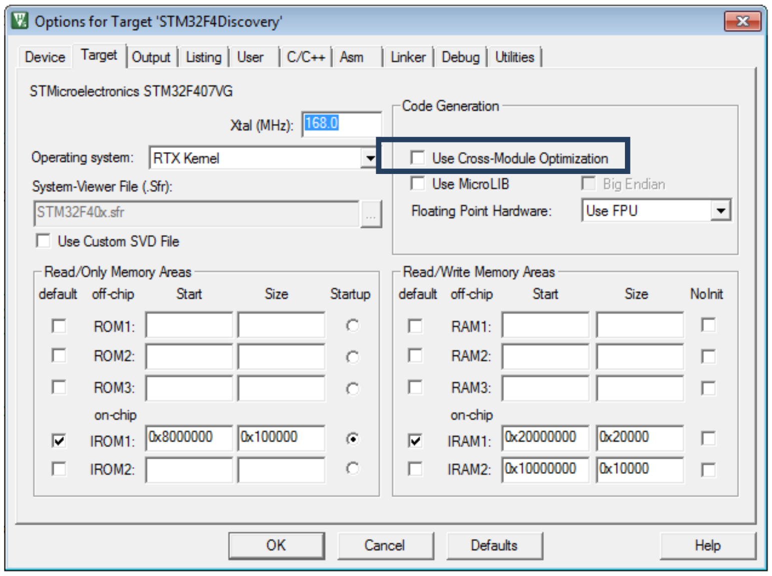
RTX is included by the µVision4 but not directly by µVision5 so you have to download the legacy support pack for cortex-M devices from: http://www2.keil.com/mdk5/legacy.

You can then always check the source code in:

<<Your Keil Directory>>\ARM\RL\RTX\SRC

Setting up RTX for your project is easy:

1. Check the Options for Target (under the Project menu) – click on the target tab and select the RTX Kernel from the Operating system selector;



STMicroelectronics STM32F407VG

Xtal (MHz): 168.0

Operating system: RTX Kernel

System-Viewer File (.Sfr): STM32F40x.sfr

☐ Use Custom SVD File

Code Generation

☐ Use Cross-Module Optimization

☐ Use MicroLIB ☐ Big Endian

Floating Point Hardware: Use FPU

Read/Only Memory Areas

default	off-chip	Start	Size	Startup
☐	ROM1:			○
☐	ROM2:			○
☐	ROM3:			○
	on-chip			
✓	IROM1:	0x8000000	0x100000	●
☐	IROM2:			○

Read/Write Memory Areas

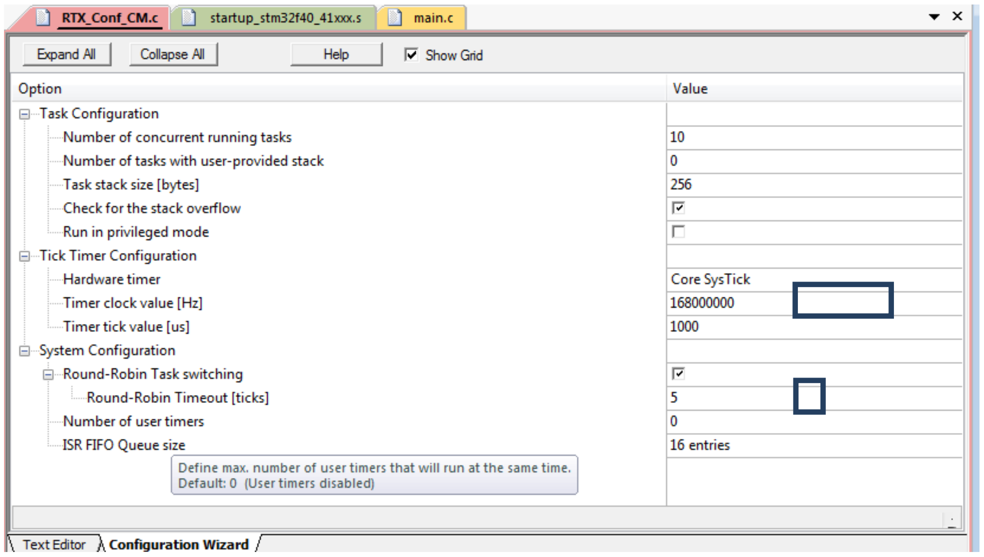
default	off-chip	Start	Size	NoInit
☐	RAM1:			☐
☐	RAM2:			☐
☐	RAM3:			☐
	on-chip			
✓	IRAM1:	0x20000000	0x20000	☐
☐	IRAM2:	0x10000000	0x10000	☐

Figure 1 Select the RTX Kernel

2. In you main.c file, include the header: #include <RTL.h>;
3. Copy the RTX_Conf_CM.c file from <<Your Keil Directory>>\ARM\RL\RTX\Config to your project directory and add to your project items.

And everything else will be done by the IDE.

You can configure the RTX by either modifying the RTX_Conf_CM.c file directly or selecting the Configuration Wizard tab to select options using a more graphically driven interface. Make sure the Round-Robin Task switching is **selected** and the timer clock value matches your CPU's. Note the "Timer tick value" as well.



Option	Value
Task Configuration	
Number of concurrent running tasks	10
Number of tasks with user-provided stack	0
Task stack size [bytes]	256
Check for the stack overflow	✓
Run in privileged mode	☐
Tick Timer Configuration	
Hardware timer	Core SysTick
Timer clock value [Hz]	168000000
Timer tick value [us]	1000
System Configuration	
Round-Robin Task switching	✓
Round-Robin Timeout [ticks]	5
Number of user timers	0
ISR FIFO Queue size	16 entries

Define max. number of user timers that will run at the same time.
Default: 0 (User timers disabled)

Figure 2 Configuration Wizard with its default values

3 RTX Task

RTX uses the word "task" for a process. This is basically represented by a C function with the keyword **"__task"** (note the two underscores) preceding the function return type. This indicates that the compiler will not handle the function entry and exit codes, which are usually responsible for managing the stack (as the RTX scheduler will handle the stack as will be seen later).

A typical task looks like this:

```
__task void task(void){
        for(;;){
                ……
        }
}
```

Notice, a task in embedded systems usually never terminates and should be considered as a self-contained mini program. You may start or end a task through OS functions which will be covered later in this lab. Also the return type is usually always **void**.

4 Task Creation

There are several functions to create a task:

```
os_tsk_create(task,priority);
os_tsk_create_user(task,priority,&stack,sizeof(stack));
os_tsk_create_ex(task,priority,parameter);
```

You are allowed to determine the priority of the task. You can also assign a large size stack to the task using the _user function; otherwise the RTX will assign a stack with default size. It is also possible to pass parameter to the task by using the _ex function.

All these functions will return the task ID (0S_TID type), and a standard way of creating task is shown as follows:

```
OS_TID taskID;
__task void task(void);
taskID=os_tsk_create(task,0);
```

The task ID is a convenient handle and will be used in some other functions related to managing the task.

A task can be created by other tasks. And usually you will initialize the OS by creating the first task by using the function:

```
os_sys_init(task);
```

This does necessary configuration for the OS and begins the first task. First task usually creates other useful tasks and then destroy itself. See the task termination.

5 Task Termination

It is also possible to terminate a task by using the following functions:

```
os_tsk_delete(taskID);
os_tsk_delete_self();
```

The first function deletes another task while the second function deletes the task executing this function (itself).

The delete function returns the OS_RESULT type, which is basically an error code.

6 Context Switching

A task maintains mainly two sets of information: the Task Control Block (TCB) and the Task Stack. TCB is dynamically allocated when you call the os_task_create() functions. All tasks in preemptive settings will have their own call stack, the size is also dynamically allocated when you call os_task_create().

Upon switching from one task to another, the OS will save the state of all the task variables to a task (some may be saved in the system stack by interrupts) and store the runtime information related to that task in its TCB. Current CPU registers will be pushed to the current task's stack and the stack pointer will loaded the next task's saved stack pointer value to pop the next task's context into CPU registers. The next task will be resumed by loading its program counter value to the PC register.

7 Lab Procedure

1. Start with the Template Project in "RTX Basics/Lab Template".
2. If not already present, copy the main.c file provided to your project.
3. Set up the RTX as described above.
4. Compile the code.
5. Load it onto your Discovery board.
6. Press the Reset button (Black joy stick) on the board.
7. Describe what you see from the board.
   The red LED is on.
8. Modify the Turn_GreenLed_On and Turn_GreenLed_Off functions to tasks. Then create one of each task in the init task with priority 0. Make sure you get the returned TID.

   __task keyword and create function. See the solution project.

9. Then run your program on the board. Describe what you see from the board.
   The red LED is on. The green LED is flashing.

   You should see the green LED flashing. As you have created two tasks, one keeps turning on the green LED and the other keeps turning it off. As they have the same priority, the scheduler will assign CPU time equally to them and preempt them as the time slice expires, resulting in the flashing. In later labs you will understand the details of the scheduling and how to control the frequency of flashing and make use of the scheduler's features to achieve your design goal.

10. Set a breakpoint at one of your "create" functions. Step through it using the F11 key to see what it does. List the major steps for creating the task. (Note that brown text labels in the

dissassembly window are the names of the C functions which the following assembly code implements). List the major steps (function names) for creation the task.,)

SVC_Handler, rt_tsk_create, rt_alloc_box, rt_init_context, rt_alloc_box, rt_init_stack, rt_dispatch, rt_put_prio

rt_alloc_box allocates a memory block and return the start addrss

rt_init_context initializes general part of the TCB e.g. giving null pointers and assigning zero values.

rt_init_stack initialize the stack and context registers

rt_dispatch dispatch the task if it has higher priority than the current one

rt_put_prio puts the task into the list based on its priority

Conceptually, assign memory for the stack, initialize the stack, initialize the TCB, find an entry in the OS TCB table (os_active_TCB), assign new TID, dispatch the task() if possible, join the task into the list.

You can also check the source code to get the answer. You don't have to go into the assembly details as long as you have the general idea what is necessary for the OS to create a task.

11. There is also a function to create a task: os_tsk_create_user_ex and you need to pass 5 parameters to this function. What are the parameters and what does this function do? Have a guess first, then look up the source code to see if you are correct.

This function allows you to create a task with specified stack and also to pass a parameter to the task on startup: `taskID = os_tsk_create_user_ex (task, priority, &stack, sizeof (stack),parameter);`

12. Similarly, you can set a break point at the os_tsk_delete_self() line. Step through it to see what it does. List the major steps for terminating the task.

SVC_Handler, rt_tsk_delete, rt_get_PSP, rt_stk_check, rt_free_box, rt_dispatch

rt_get_PSP returns the process stack pointer (ARM SP are banked with PSP and MSP)

rt_stk_check checks if there is a stack overflow

rt_free_box frees the memory space

So conceptually, the task will change its states to inactive, make sure the stack and other memory is properly freed, release the mutex if it has any, unlink the TCB from lists and dispatch the next task.

You can also check the source code to get the answer. You don't have to go into the assembly details as long as you have the general idea what is necessary for the OS to create a task.

13. Now try to add `os_tsk_pass ();` to the end of both your tasks' loops. It tells the OS the task is voluntarily giving up its execution time to the next ready task. Context switching will occur at these lines. Describe how this will affect the behaviour of the green LED. Run your program again to verify your hypothesis.

    The green LED is still flashing. However much faster so that it is hard to tell if the LED is constantly On. A task will not consume all of its time slice and the OS performs the context switching immediately after the LED is turned on or off. So you may use an oscilloscope to see the flashing.

14. Set a breakpoint at one of the os`_tsk_pass()` line. Step through it to see what it does. List the major steps for context switching. Also pay attention to the green LED while you are stepping through.

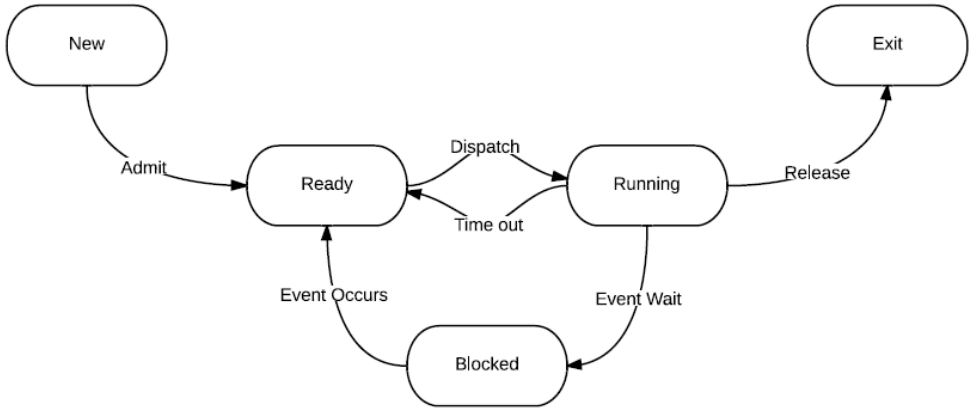
    SVC_Handler, rt_tsk_pass, rt_get_same_rdy_prio, rt_put_prio, rt_switch_req, rt_stk_check

    rt_get_same_rdy_prio get the first task from the list with the same priority as the current task.

    So conceptually, get the next 'ready' task with the same priority and put the current task to the ready list and change the state of the current task to 'ready' and dispatch the next 'ready' task by making its state 'running'.

    This is relatively simple and you can check the RTX source code to see how the OS simplifies the programmer's job.

15. You have been introduced to the State Transition concept in the lecture.



Try to match the following functions to the transitions in the State Transition model:

os_task_create() - Admit

os_task_delete() - Release

os_tsk_pass() – Time out

rt_dispatch() - Dispatch