OS Task Lab  
RTX basics

# Overview

In this lab you will configure the RTX and create and destroy some tasks using the RTX and you are expected to walk through how the OS perform task creation, termination and switching operations. Additionally, this lab will involve some concepts from the Scheduler Lecture, feel free to skip any concepts you are not familiar with. Don’t panic if you do not understand 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.

# Setting up RTX

RTX is included by the µVision4 but not directly by the 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, the target tab, select the RTX Kernel from the Operating system selector;

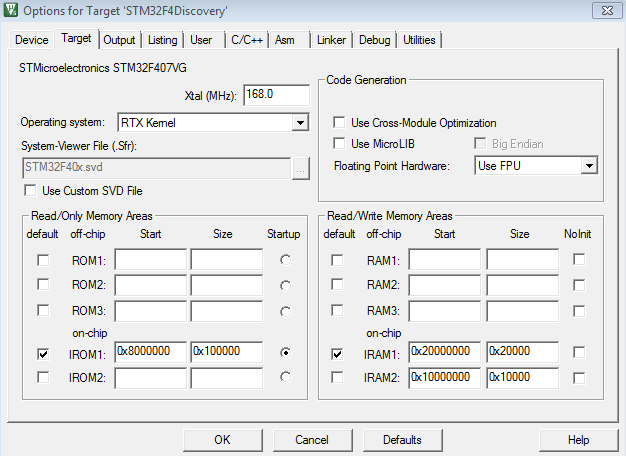


Figure Select the RTX Kernel

1. In you main.c file, include the header: #include <RTL.h>;
2. 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 or selected the RTX\_Conf\_CM.c file from the Project tab and click on the Configuration Wizard to choose options from different configurations. Make sure the Round\_Robin Task switching is **selected** and the timer clock value matches your CPU’s.

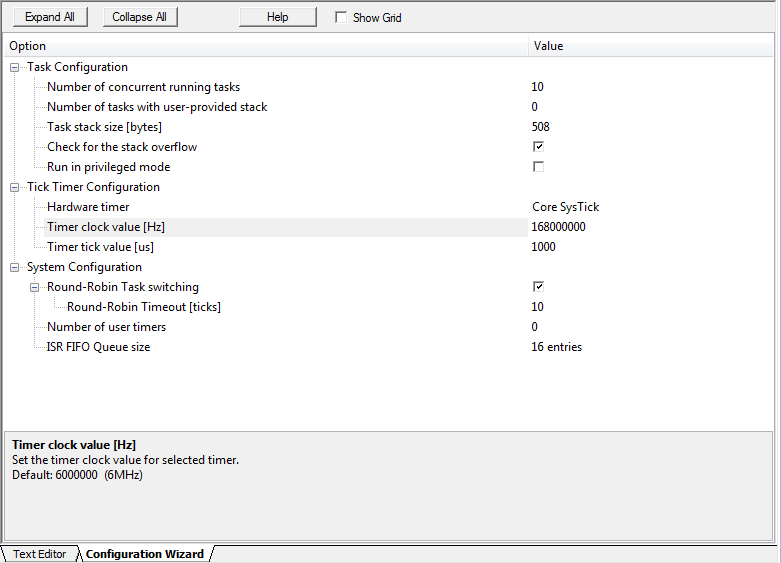


Figure Configuration Wizard and its default value

# RTX Task

RTX uses the notation of “task” as a process. And it is basically a C function with the keyword **“\_\_task”** (two underscores) preceding the function return type. This indicates that the compiler will not handle the function entry and exit code, which are usually responsible for managing the stack as RTX scheduler will handle the stack as we will see later.

A typical task looks like this:

\_\_task void task(void){

for(;;){

……

}

}

Notice, a task in embedded system never terminates it 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 always **void**.

# 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.

# Task Termination

It is also possible to terminate a task by using 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.

# Context Switching

A task maintains majorly two parts 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.

# Lab Procedure

1. Copy the main.c file provided to your empty project.
2. Do not use the run-time environment, i.e. nothing should be included automatically by the IDE, include system\_stm32f4xx.c and startup\_stm32f4xx.s files 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.

1. Modify the Turn\_GreenLed\_On and Turn\_GreenLed\_Off functions to tasks. And 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.

1. 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 may see the green LED flashing. As you have created two tasks, one keeps turning on the green LED and the other keeps turnning off the green LED. As they have the same priority, the scheduler will assign CPU time equally to them and preempt them as the time slice expired, thus the flashing. In later labs you will know the details of the sceduling and how to control the frequency of flashing and make use of the scheduler’s features to achieve your design goal.

1. Set a breakpoint at one of your create function. Step through it to see what it does. List the major steps for creating the task. (Not the assembly, the sub functions the create function calls, the brown line of the assembly windows reminds you the corresponding C function of the assembly code)

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

Conceputally, 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.

1. 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 s tartup: taskID = os\_tsk\_create\_user\_ex (task, priority, &stack, sizeof (stack),parameter);

1. 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.

1. Now try to add os\_tsk\_pass (); to the end (not the end of the loop) of both your tasks. 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 hard to tell if the LED is constantly On. A task will not consume all its time slice and the OS perform the context switching immediately after the LED is turned on or off. So you may use a oscilloscope to see the flashing.

1. 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.

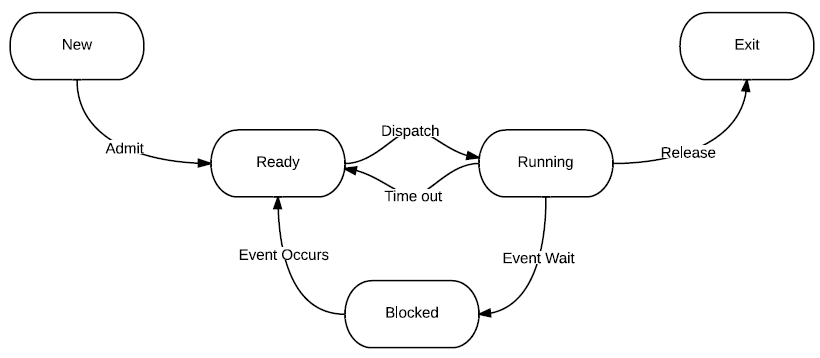
SVC\_Handler, rt\_tsk\_pass, rt\_get\_same\_rdy\_prio, rt\_put\_prio, rt\_switch\_req, rt\_stk\_check

rt\_get\_smae\_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 OS simplifies everything.

1. You are introduced to some of the



Try to match the following functions to the transitions in the state transition models:

os\_task\_create() - Admit

os\_task\_delete() - Release

os\_tsk\_pass() – Time out

rt\_dispatch() - Dispatch