and

Present

Developing with CMSIS-RTOS , RTX and the STM32F0 Discovery Board

Table of Contents

1.0 RTX Workshop for Beginners and Intermediate Users

This workshop and workbook are designed to provide an introduction to the CMSIS-RTOS API and the open source edition of the RTX Real Time Operating System.

RTX is provided by ARM as the reference implementation for CMSIS-RTOS compliant RTOS implementations. It is available under a simple BSD license making it straightforward to use and redistribute.

The examples in this workbook are written to work on the STM32F05x Discovery Board and make use of the STM32 Standard Peripheral Library and could therefore be easily translated to any other STM32 platform.

1.1 Equipment needed

1. STM32F05x Discovery Board
2. MDK-ARM v4.5 or higher (The free version called MDK-Lite is adequate for running the examples in this workshop)

1.2 Hardware Connection

Connect the hardware as shown in Figure 1.

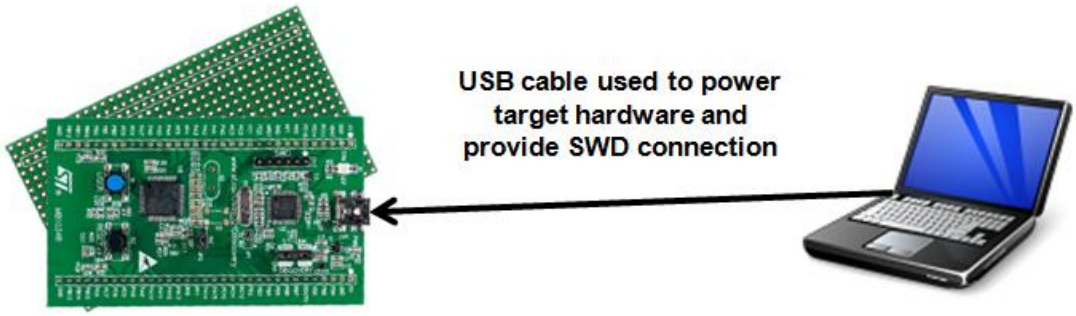


Figure 1. Hardware Connection

1.3 Learning Structure

Each session will focus on a key learning area and will make extensive use of hands-on examples.

When you look at the example files, you will notice that there is a Hardware Abstraction Layer (or HAL). While this may seem over complicated, it is done this way deliberately to ensure that the learning experience is focused on RTX rather than configuration of the MCU it is being used on.

Session 1

- Basic setup and configuration of CMSIS-RTOS : RTX in a MDK project
- Writing a task for RTX
- Use of the RTX aware debug tools built in to MDK

Session 2

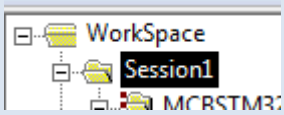
- Use of Interrupts in an RTX application
- Introduction to RTX events system

1.5 Document Conventions

Where there are actions for the user of the document to take they will appear in blue-background boxes like this:

Actions for the user will be in a box like this.

There may also be pictures or icons to help guide you to the correct place in the example.



Any additional information will appear as standard text.

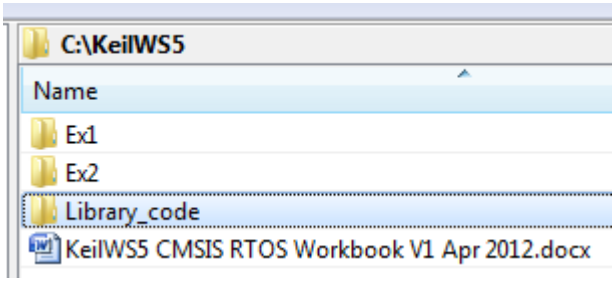
1.6 Installing the workbook Files

You will have been provided with a zip file called KeilWS5.zip, this needs ot be copied to your PC and unzipped to be used in the workshop.

Unzip the file KeilWS5.zip to C:\

If you choose another path, you must remember it as the workbook instructions will assume you have use the path above.

When the workbook is extracted to the default location, the directory structure will look like this:



There are folders for the examples, an electronic copy of the work book and a folder called 'Library_code'.

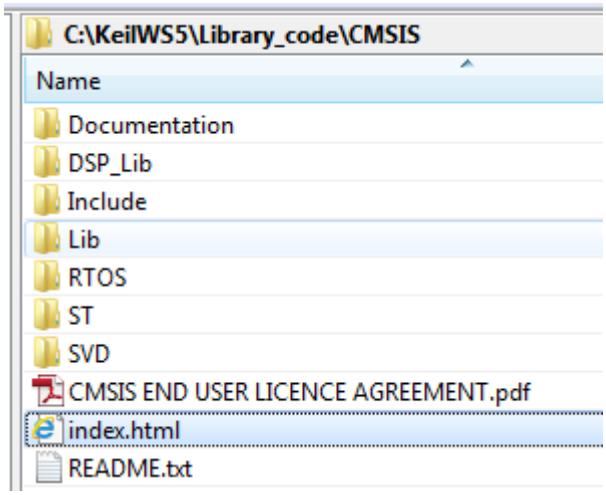
The Library_code folder contains all the source material used for CMSIS, RTX and for the STM32 standard peripheral libraries. The various examples in this workbook link to the files in this directory as required.

There is also full documentation for the CMSIS v3 implementation included, this documentation can be viewed using a standard web browser such as internet explorer or Chrome.

The CMSIS documentation can be found in:

C:\KeilWS5\Library_code\CMSIS

And double click the index.html file to open the documentation.



2.0 Session 1 – Setting up RTX and debugging

This Session will run through the configuration of CMSIS-RTOS : RTX in a MDK project, the basic creation of tasks (and the Idle Daemon) and will then move on to using the RTX aware debug tools that are part of MDK.

2.1 Configuration of RTX

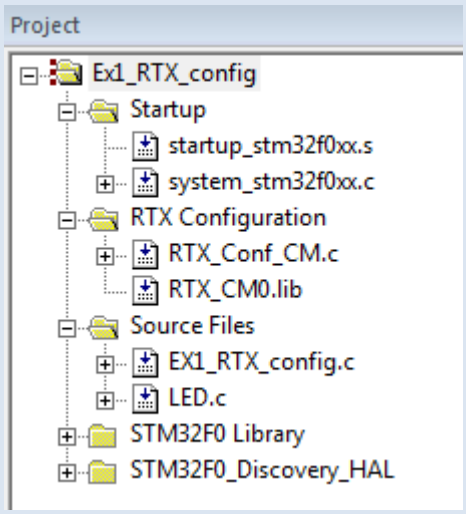
Open the project for Example 1.

Assuming you followed the default installation instructions (section 1.6) you will find this in the following directory:

```
C:\KeilWS2\EX1
```

The workspace project is called:

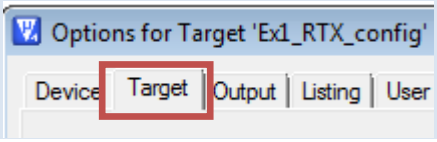
```
EX1_RTX_config.uvproj
```



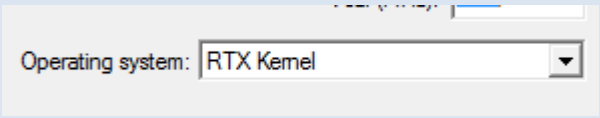
Next, we need to make sure that the project will be ready to use the RTX Real Time Operating System. We do this using the project options dialog.

Open the project option dialog using the magic wand icon. 

In the project Options dialog, select the Target tab.



In the Target tab, find the drop-down box for Operating System Support and select the RTX option.



Click on OK to close the Target Option Dialog.

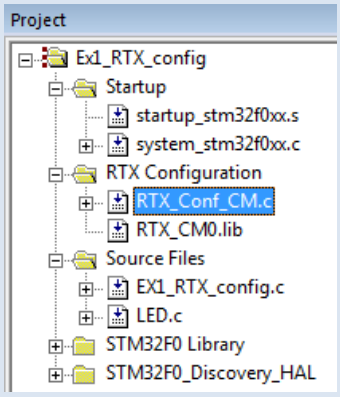
This action ensures that MDK is fully aware that we are using the RTX RTOS and will enable the RTX aware debug windows.

The next action is to configure the RTX RTOS to work the way we need it to for our application.

The project needs to have access to the file `RTX_Conf_CM.c` as this is the main configuration file for RTX. In this example the file is already added to the project source list, however if you should need to locate this file for a different project you can copy it from this example project or it can be found in the standard Keil installation directory at `C:\KEIL\ARM\STARTUP`.

As we have the file in our application already we can move directly to the configuration phase.

In the project, open the file `RTX_Conf_CM.c` (double click on the file to open it)

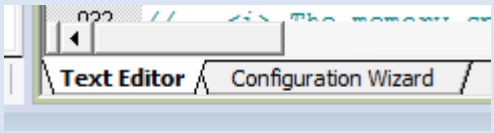


Scroll to line 35, notice the include:

```
#include "cmsis_os.h"
```

This is a call to include the CMSIS-RTOS API header.

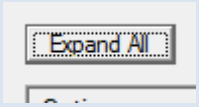
Notice at the bottom of the window you can see a tab for the source file and a tab for a Configuration Wizard.



Click on the Configuration Wizard Tab.

In the RTX Configuration Wizard we can change all the major elements of how RTX will operate.

Click the Expand All button to see all the configuration options.



Any changes made in the Configuration Wizard will appear directly in the source code for RTX_Conf_CM.c

We can now configure RTX to make sure it will operate the way we need it to for our application.

Figure 4 shows how we need RTX to be configured:

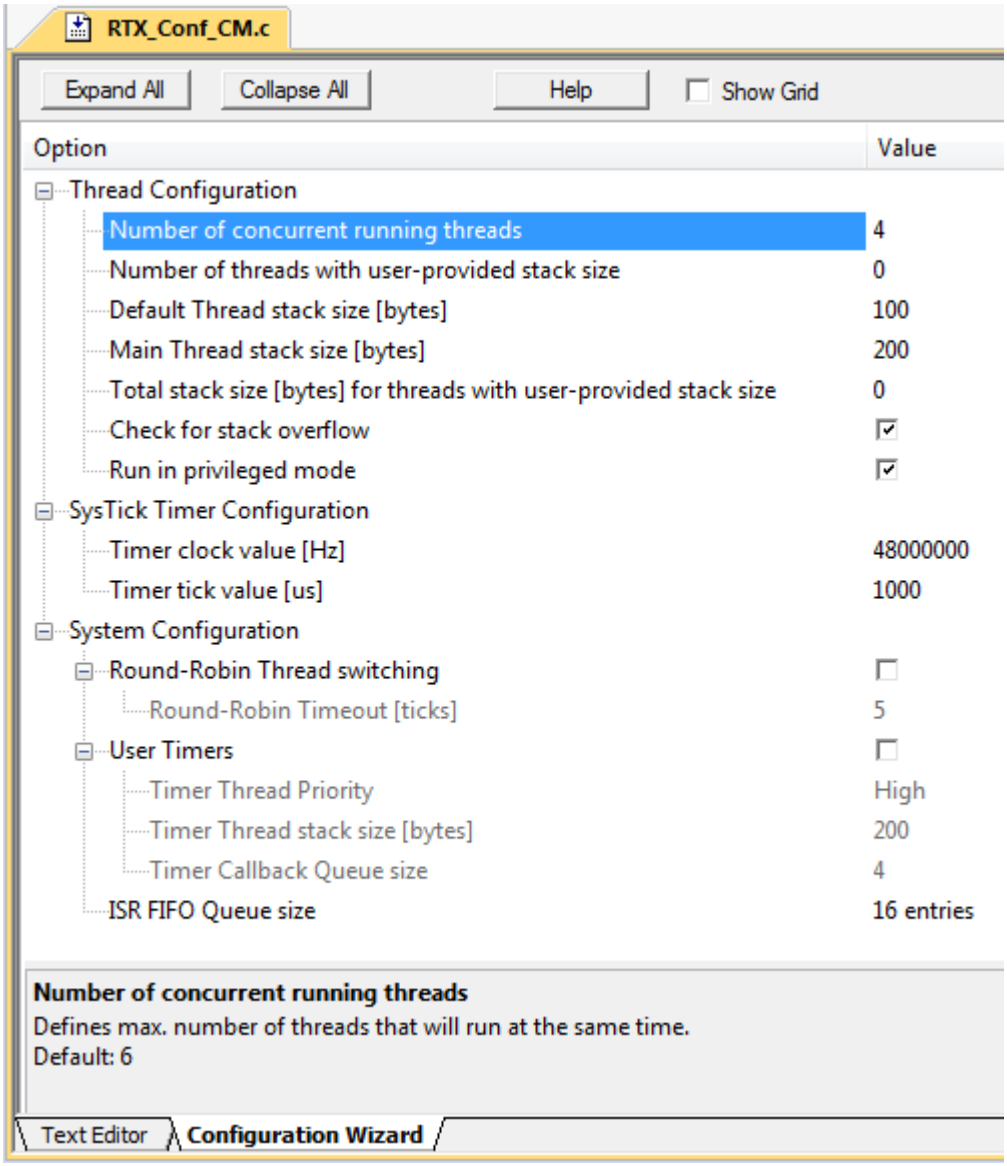


Figure 4. RTX Configuration for KeilWS5 Ex1

When you left click on a parameter the information panel at the bottom of the configuration wizard offers some explanation for the chosen parameter.

The Number of Concurrent Running Tasks can be set to a maximum of 250 but you can have an unlimited number of Defined Tasks.

The Task Stack Size in Bytes when set here will apply to all tasks in the application. It is possible to set individual stack sizes for tasks if required.

The only hardware resource that RTX consumes is the Systick Timer on the Cortex-M microcontroller. Here we can configure the clock that is applied to the Systick and subsequently the Timer Tick value that will be used as a Heart Beat for RTX to manage the tasks and timing in the application.

In our example we are using a Timer Tick of 1 ms.

If required, using this dialog you can also select Round Robin task switching and fix the delay at which the Round Robin switch will occur.

For our application all task switching will be co-operative (so when a thread has finished what it needs to do, it passes runtime on to the next available thread).

We will also make use of thread Priority levels – a higher priority thread can take runtime from an active low priority task.

Ensure your RTX Configuration window shows the same values as in Figure 4, correct anything that is different!

2.3 Basic Task Creation

In an RTOS, the basic unit of execution is a "Task". A task is very similar to a C procedure, but has some fundamental differences.

Procedure	Task

`unsigned int procedure (void) {`
`  …`
`  …`
`  return (val);`
`}`
	`__task void task (void) {`
	`  for (;;) {`
	`  …`
	`  }`
	`}`

Table 1. Task and Procedure differences

We always expect to return from C functions, however, once started an RTOS task must contain an endless loop, so that it never terminates and thus runs forever.

You can think of a task or thread as a mini self-contained program that runs within the RTOS. While each task runs in an endless loop, the task itself may be started by other tasks and stopped by itself or other tasks.

When a thread is created, it is allocated its own thread ID. This is a variable, which acts as a handle for each task and is used when we want to manage the activity of the task.

Some 'system' threads are created automatically by RTX, such as the idle daemon. We will look at the idel daemon in more detail later in this workbook.

Let's create the first thread for our application.

Open the file `EX1_RTX_config.c`

In our application, this will be the main source file and will contain all out threads and their definitions.

The thread we create now will be called `LED_0`, it will simply toggle and LED on the target hardware every 500 msec.

The HAL (LED.c in this case) will take care of the manipulation of the hardware registers so that you can focus on learning about CMSIS-RTOS and RTX.

In the file `EX1_RTX_config.c` at line 55 add the following (use the comments in the file to guide you):

```
void led_0 (void const *argument) {

  for (;;){

          }

}
```

This gives us the basic elements of the thread. The first line:

```
void LED_Task (void const *argument) {
```

is used to create the basic element of the thread.

Next, the `for(;;)` command sets up the main loop in the thread that will run forever. Now we need to add some action into our thread to make the LED on the board flash.

Inside the **`while`**`(1)` loop add the following:

```
LED_On(0);              /* Turn LED 0 On   */

osDelay(500);
                        /* Delay running task for 500 ticks  */

LED_Off(0);
                        /* Turn LED 0 On                     */

osDelay(500);
                        /* Delay running task for 500 ticks  */

LED_0_toggle_counter++; /* Increment LED 0 toggle counter          */
```

The `LED_On` and `LED_Off` calls are into the HAL for control of the MCU hardware. IF you are interested to know how the HAL works you can look into the files but they are not relevant for learning RTX.

The line `osDelay(500)` will put the task into a state where it is waiting for a fixed period of time (500 OS Ticks, which in our application is 500 msec).

When a thread is in this state, it is no longer the running/active thread so other threads in the RTOS application are able to execute. When the delay specified has elapsed, the thread returns to a ready state and is scheduled by RTX.

A thread can be in one of four basic states:

RUNNING, READY, WAITING, or INACTIVE.

In a given system, only one thread can be running, that is, the CPU is executing its instructions while all the other threads are suspended in one of the other states.

RTX has various methods of inter-thread communication: signals, semaphores, and messages. Here, a thread may be suspended to wait to be signalled by another thread before it resumes its READY state, at which point it can be placed into RUNNING state by the RTX scheduler.

At any moment a single thread may be running. Threads may also be waiting on an OS event. When this occurs, the threads return to the READY state and are scheduled by the kernel.

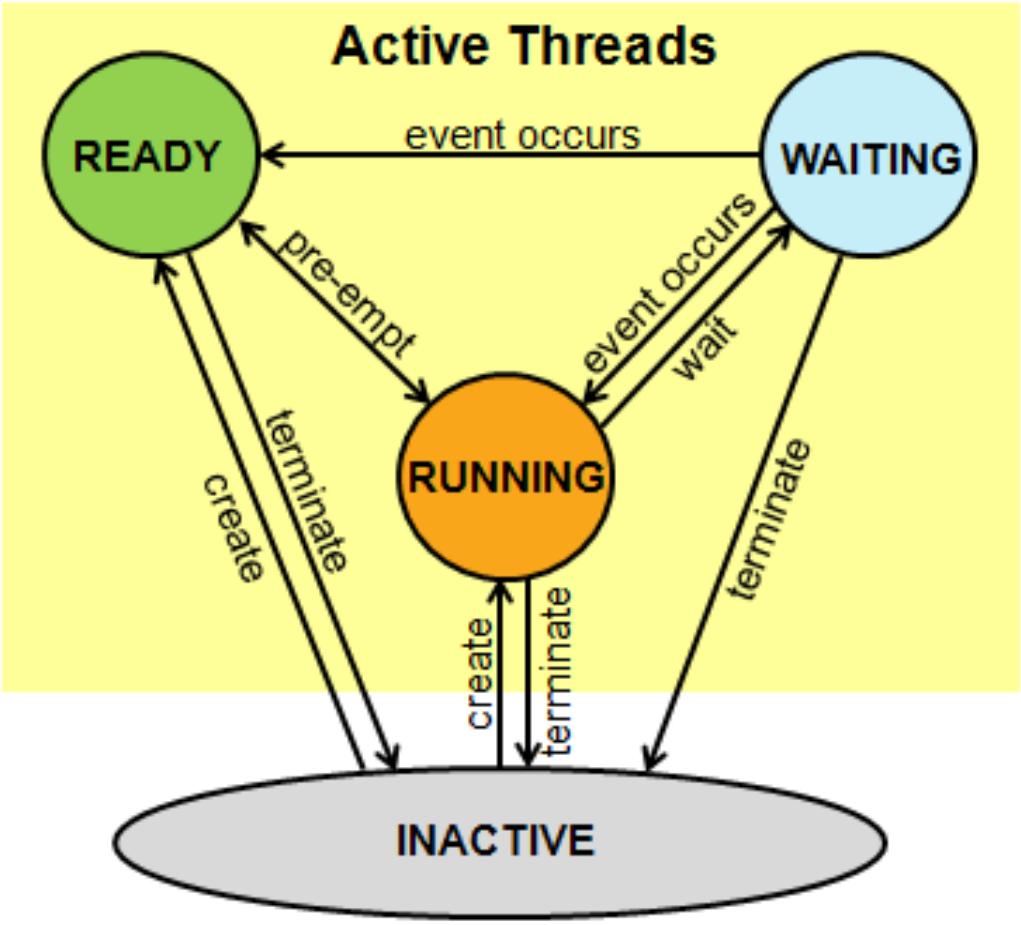


Figure 5. Possible Task states and transitions in RTX

If you have entered all the code correctly, your finished task should look like this:

```
/*----------------------------------------------------------------------------
  Thread 1 'led_0': Toggle LED 0
 *----------------------------------------------------------------------------*/
void led_0 (void const *argument) {
  for (;;) {
    LED_On(0);                          /* Turn LED 0 On                      */
    osDelay(500);                       /* Delay running task for 500 ticks   */
    LED_Off(0);                         /* Turn LED 0 On                      */
    osDelay(500);                       /* Delay running task for 500 ticks   */
    LED_0_toggle_counter++;             /* Increment LED 0 toggle counter     */
    }
}
```

Figure 6. Completed LED_0 thread for Session 1

Next we need to declare the Thread ID using the CMSIS-RTOS API.

In the file `Ex1_RTX_config.c`

At line 28 insert the following:

```
osThreadId T_led_0;
```

The line `osThreadId T_led_0;` makes a declaration to RTX of the name of the thread, in this case `T_LED_0`.

This is a variable, which acts as a handle for each thread and is used when we want to manage the activity of the thread.

We need to link this thread ID to the actual code of the thread itself. We can do that in the same line of code we use to initialise the thread in the application. Before we do that however, we need to define our thread. This is achieved using the CMSIS-RTOS API function osThreadDef(). This function has four parameters:

name	name of the thread function.
priority	initial priority of the thread function.
instances	number of possible thread instances.
stacksz	stack size (in bytes) requirements for the thread function.

In the file `Ex1_RTX_config.c`

At line 42 insert the following

```
osThreadDef(led_0, osPriorityNormal, 1, 0);
```

This defines the thread to have:

name	led_0
priority	normal
instances	1
stacksz	0 (uses the default thread stack size we configured earlier)

We also need to create a Forward reference so the compiler can handle the thread declaration efficiently.

In the file `Ex1_RTX_config.c`

At line 35 insert the following

```
void led_0 (void const *argument);
```

Now we need to initialise our RTX application.

We do this in `main()` int the file Ex1_RTX_config.c and this is where we will also link the thread ID we made earlier to the thread definition and the thread code.

Main() is also treated like a thread in CMSIS-RTOS and RTX but as it is a 'system' function it doesn't need a thread ID or a definition.

In the file Ex1_RTX_config.c at line 94 add the following:

```
T_led_0 = osThreadCreate(osThread(led_0), NULL);
```

`T_led_0`	This section determines which task the rest of the line affects. Its the thread Id we created earlier.
`osThreadCreate`	This is a CMSIS-RTOS API command to RTX to actually create the task, it is followed by parameters for the task to be created.

Table 2. Breakdown of Thread Creation code line

The thread function receives the *argument* pointer as function argument when the function is started. When the priority of the created thread function is higher than the current **RUNNING** thread, the created thread function starts instantly and becomes the new **RUNNING** thread.

As main() is also treated as a thread, we need to make sure it doesn't take runtime in the application unless needed. To do this we can add a simple CMSIS-RTOs API command to surrender runtime from the main() thread to the rest of the application.

In the file Ex1_RTX_config.c

At line 97 insert the following:

osThreadYield();

This line will cause the main() thread to delete yield its requirement for runtime once it has completed. We only need it to run once, so having it yield frees up resources in the application for other threads to use in run time.

2.4 RTX Idle Demon

The thread we have created will spend a lot of its time in the WAIT DELAY state. You will have noticed as we went through the exercise that there was also another thread, similar to the one you created, that will control a different LED at a different speed. Both tasks will spend a majority of their time in WAIT DELAY.

As there are no other threads running in our application, there could be a lot of 'down time' where there is no active running thread. In this case we could use an idle demon.

RTX creates the idle demon for you, there is no need to add or modify any code to have this idle demon active in your application but you can control what happens when the application is running the idle demon.

In this example we do not want anything special to happen in the idle demon, but a variable has been added to act as a counter to provide a visualisation for you in the debugger so you can see the time spent in the active threads and the time spent in the idle demon.

The code for the Idle Demon is in the file `RTX_Conf_CM.c`, the default location in the file is line 211:

```
209  /*---------------------------- os_idle_demon ---------------------------------*/
210
211  void os_idle_demon (void) {
212    /* The idle demon is a system thread, running when no other thread is      */
213    /* ready to run.                                                           */
214
215    for (;;) {
216    /* HERE: include optional user code to be executed when no thread runs.*/
217    idle_count++;   /* counter used just to show activiy in idle task */
218    //__wfe();        /* wfe instruction mimimises power consumption in idle time */
219
220    }
221  }
```

Figure 7. Idle Demon in RTX Configuration File

Now we have all the elements we need to complete the session 1 code.

Build the Session 1 application buy pressing the Re-Build All Button 

The project should build with zero errors or warnings.

If you have any problems please ask the person running the workshop for help!

2.4 Debugging an RTX application with MDK

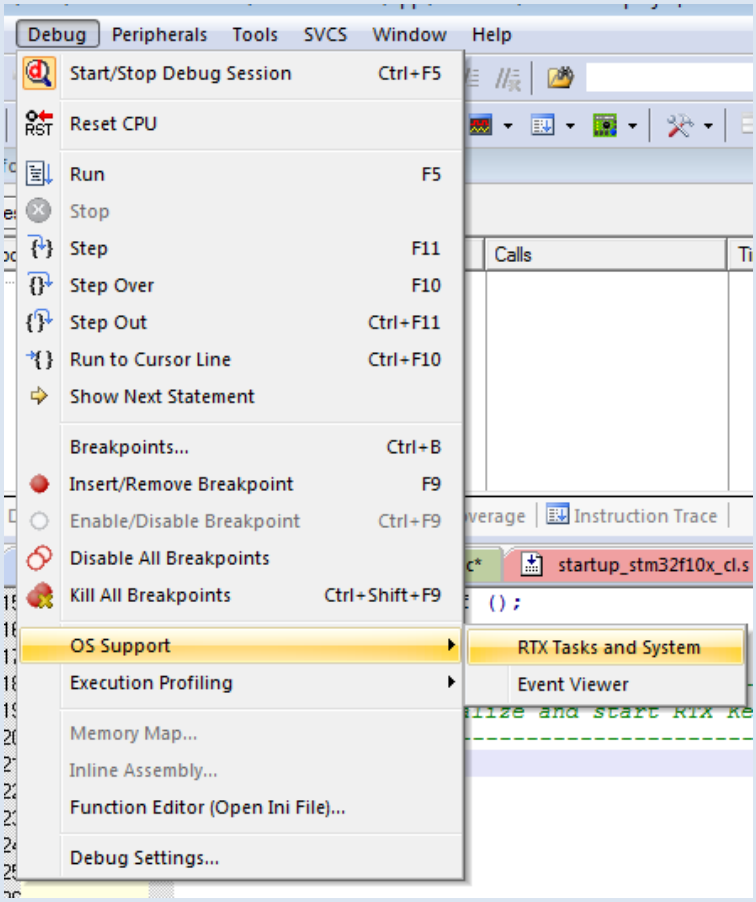
Now we have the application successfully built, we can enter debug. In this workbook and associated exercises, all of the debug settings are pre-configured so we can simply enter directly into the debug session. (If you want to learn more about configuration of the debug settings, please refer to KeilWS1.)

Enter the debug session by pressing the Start/Stop Debug button  (or by pressing Ctrl+F5)

Now we need to open the special RTX aware debug windows. We can do this through the Debug menu in the debug environment.

From the Menu bar select:

Debug > OS Support > RTX Tasks and System



This will open a new window on the right hand side of the MDK debug view. The RTX Tasks and System window will update while the application is running and will show you information about the configuration of RTX as determined by the

settings you put into the Configuration Wizard earlier in this exercise (making it easy to spot any configuration mistakes) and information on all threads that have been created in the application, such as thread name and priority, running state and stack usage.

We also want to watch a few variables to give us an indication of how the program is behaving. We can do this easily using the command line to set the variables to watch in watch window 1.

Open the Watch1 window using:

View \ Watch Windows \ Watch 1

Check for the following variables:

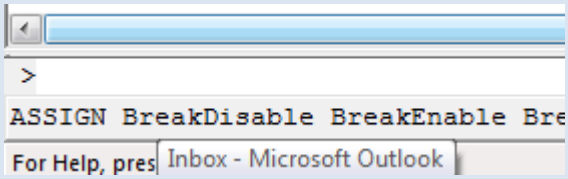
```
idle_count
```

```
LED_0_toggle_counter
```

```
LED_1_toggle_counter
```

If they are NOT present, add them using the following commands:

Locate the Command Input line near the bottom left of the MDK-ARM window



At the command prompt type:

```
WS 1, `idle_count
```

And press 'enter'

Then type

```
WS 1, `LED_0_toggle_counter
```

And press enter

And lastly type

```
WS 1, `LED_1_toggle_counter
```

And press enter

Look to the right hand side of the MDK-ARM debug environment and you will see the Watch 1 window with the variables you just added.

Note, you could also add variable to the watch window by locating them in the source code window and using the right-click context menu.

We have now configured the debug environment to show us what we need for this example.

Start the application running with the run button. 

Look in the RTX Task and System window to see the data changing.

See also on the target hardware that the LEDs are flashing at different rates.

Note:

The speed at which this data updates is directly related to the debug adaptor that is being used. The ST-Link hardware can only run the SWD interface at a speed of 1MHz which limits the rate at which this data can be extracted from the target MCU. Switching to the ULINK*pro* debug adaptor would give you the option of running this interface much faster (up to 50MHz) and the data in this window would then update much faster.

After a few seconds, stop the application using the stop button. 

The RTX Tasks and System window will look similar to that shown in Figure 8.

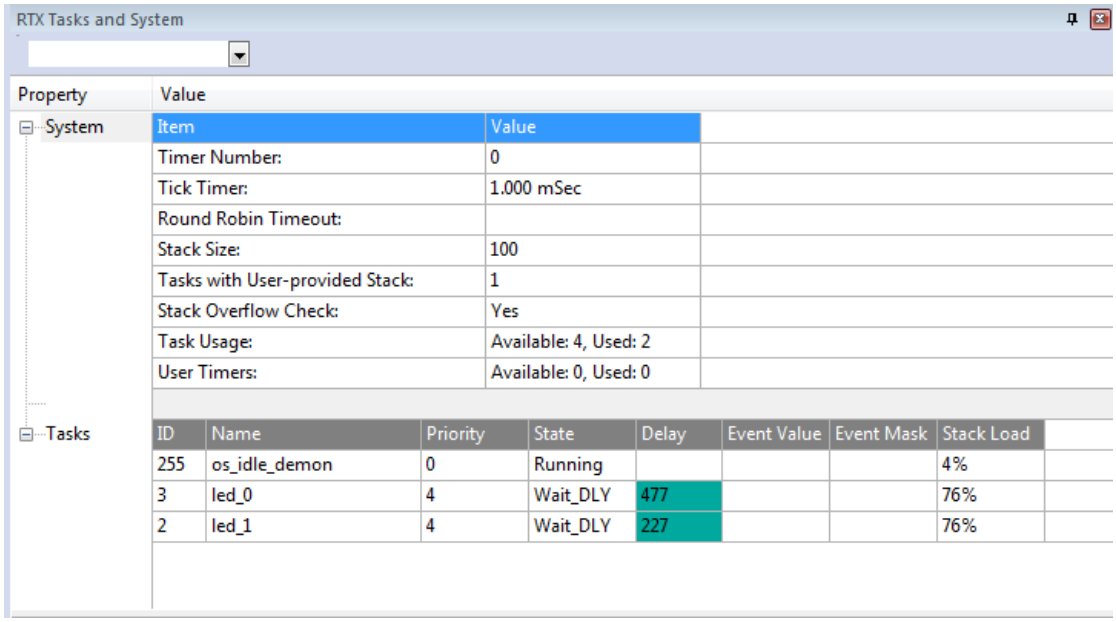


Figure 8. RTX Task and System Viewer in Session 1

The sections of data highlighted in the teal colour show they have been updated since the last refresh of the window.

You can see clearly the information on the tasks in the system too, showing the task name, priority, current operating state along with other elements.

You can see using the Watch 1 window that a significantly greater amount of time has been spent in the idle daemon than in the LED flash threads.

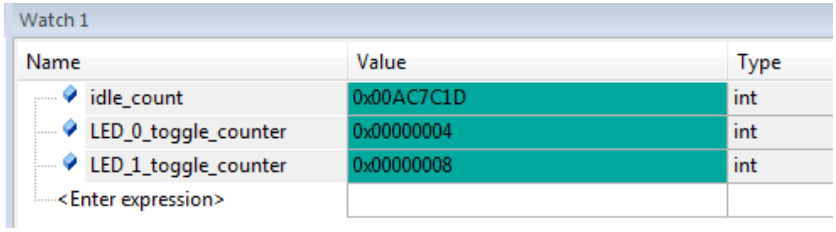
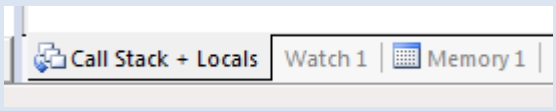


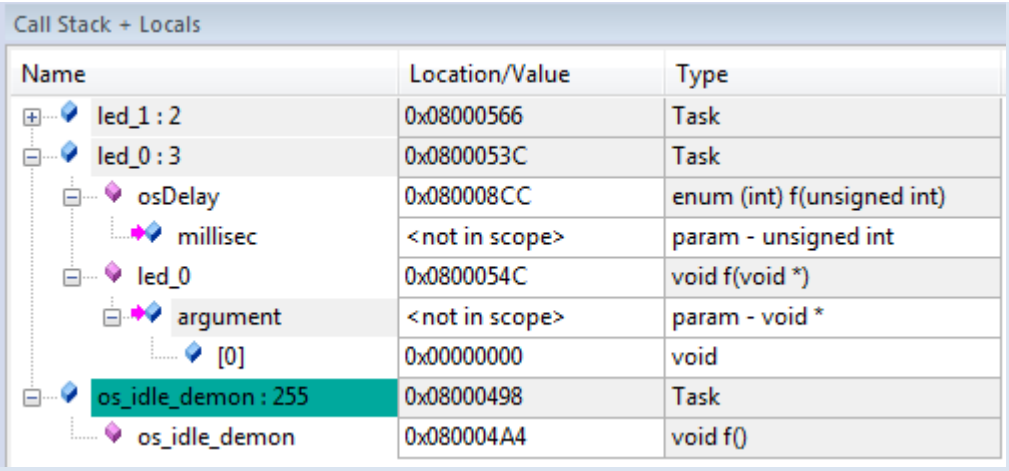
Figure 9. Watch window from example 1

You can see more information about the resources used by threads in the application by looking at the Call Stack + Locals window.

Click on the Call Stack + Locals tab (near the bottom right of the debug window)



Expand the + sign next to led_0



The Call Stack + Locals window provides the developer with up to date information for the threads running in the RTOs application.

You can see each function in a given thread and the data associated with that function. All entries are labelled so it is easy to identify what you are looking for.

Note:

The Cortex-M0 does not support the SWO (Serial Wire Output) debug interface so we cannot use MDK-ARM's RTX event viewer, but it is worth pointing out that this same application run on a Cortex-M3 or Cortex-M4 would be able to show a complete historical timeline of the thread transitions and present various data for each thread such as min, max and average runtime, number of calls etc. There are also measurement cursors in the event viewer so the developer can easily determine the behaviour of their RTX based application.

An active event viewer window will look similar to this:

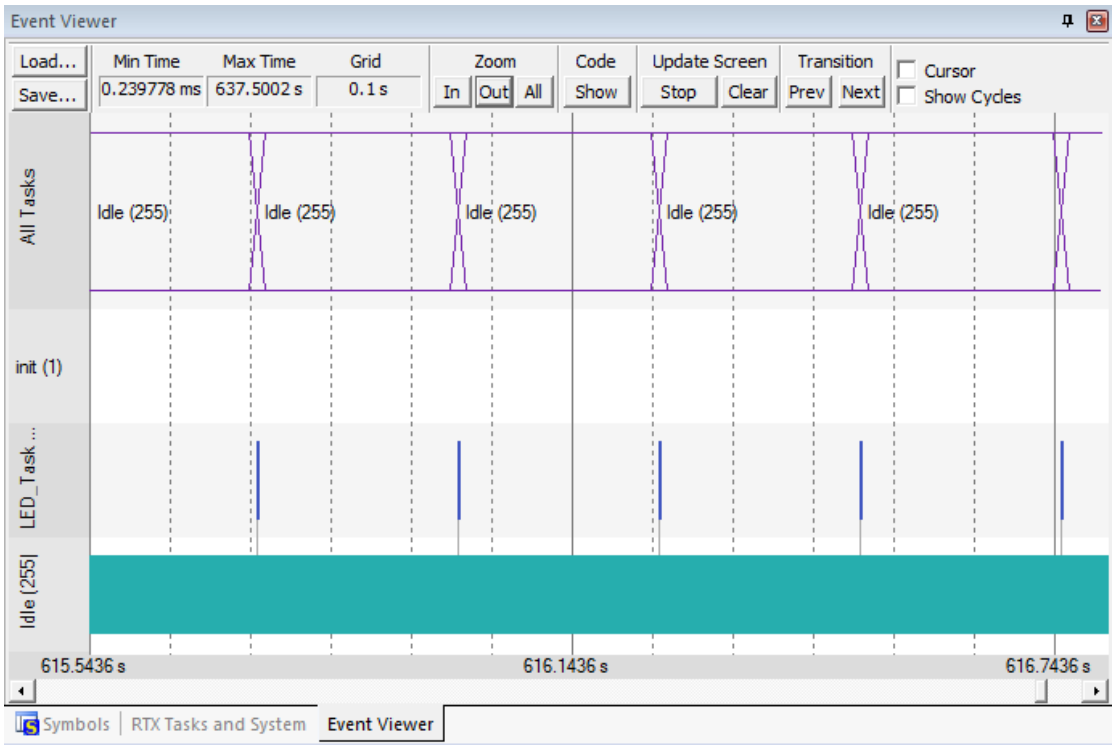


Figure 10. Event Viewer sample screenshot

There are many other debug functions available in MDK-ARM but they are beyond the scope of this workbook (we are concentrating on RTX!) .

When you are finished, exit debug using the start/stop debug button. 

This concludes Session 1. You have seen how to configure RTX, how to create a thread and how to debug an RTX application using the RTOS aware debug tools in MDK.

3.0 Session 2 – Using Interrupts and Signals

In this session, we will introduce the usage of interrupts in an RTX application, primarily to demonstrate how using RTX has no negative impact on the interrupt system in a Cortex-M Microcontroller but additionally to show that it is easy to implement as well! Additionally we will use RTX signals to re-activate the thread from a waiting state.

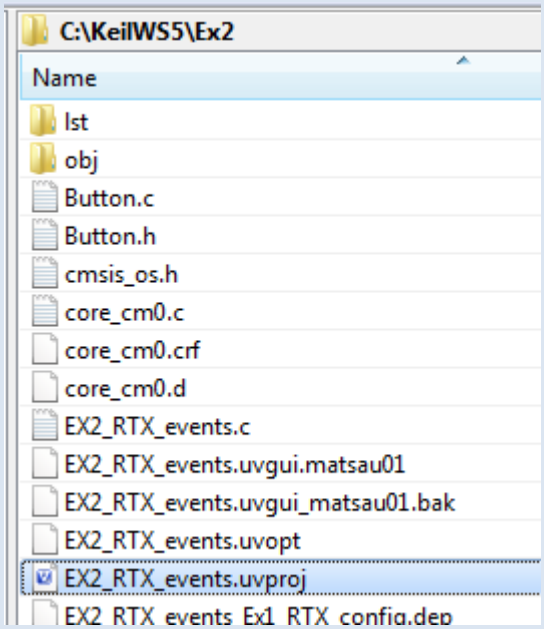
3.1 Starting up the project

We need to be working in the Example 2 project for this session.

Close the project from session 1 if it is still open.

Navigate in the workshop directories to find the project for example 2:

C:\KeilWS5\Ex2\Ex2_RTX_events.uvproj



As with all exercises in this workbook, the HAL will manage the necessary registers in the MCU for configuration and hardware recourses, the interrupt settings in this case. This is of course hardware specific. The way RTX works with them however is always the same and that is what we are more interested in here.

The LED.c file from the previous exercise is present, but this time there is also a file called Button.c, this has the configuration in it for setting up the interrupts on the STM32F05x. This setup is done using the STM32 Standard Peripheral Library.

3.2 Adding code to deal with the interrupt

To demonstrate the use of interrupts we will use the 'user' button on the Discovery board to generate an interrupt. When this intruupt occurs we will use the Interrupt Service Routine to set one of the signal flags in another task to allow it to move out of the waiting state. To achieve this we need to create a new task, called `button_control`, to take care of the button press and manage the Interrupt Service Request (ISR).

In the file Ex2_RTX_events.c add the following at line 30:

```
osThreadId T_button_control;
```

The Forward Reference has been added for you already.

And at line 44 replace the '`xxx`' with the information below to complete the thread definition

Name: `button_control`

Priority: `osPriorityRealtime`

Instances: `1`

Stack: `0`

Setting the priority to `osPriorityRealtime` will put this thread as the highest priority thread in the system, so if there is a clash for required runtime in the RTOS, this task will win and be able to serve the ISR.

At Line 114 replace the 'xxx' with the information necessary to create the thread link it to its thread ID.

Thread ID: `T_button_control`

Thread Name: `button_control`

That will cover everything we need in the `Ex2_RTX_events.c` file to include and create the new thread for the button press. Now we need to create the thread itself and the handler for the Interrupt Request.

For this exercise, when we get a button press, we will simply increment a counter variable but you could easily modify this example to do much more.

It is important to note that RTX will not add any delay to the way an interrupt works on a Cortex-M microcontroller, so in the case of the Cortex-M3 the fixed 12 cycle entry to the ISR will remain valid.

First we will look at the interrupt handler for the Interrupt Request and we are going to make use of the signals system in RTX to trigger the button thread from the Interrupt handler.

When each thread is first created, it has sixteen signal flags. These are stored in the Thread Control Block. It is possible to halt the execution of a thread until a particular signal flag or group of signal flags are set by another thread in the system.

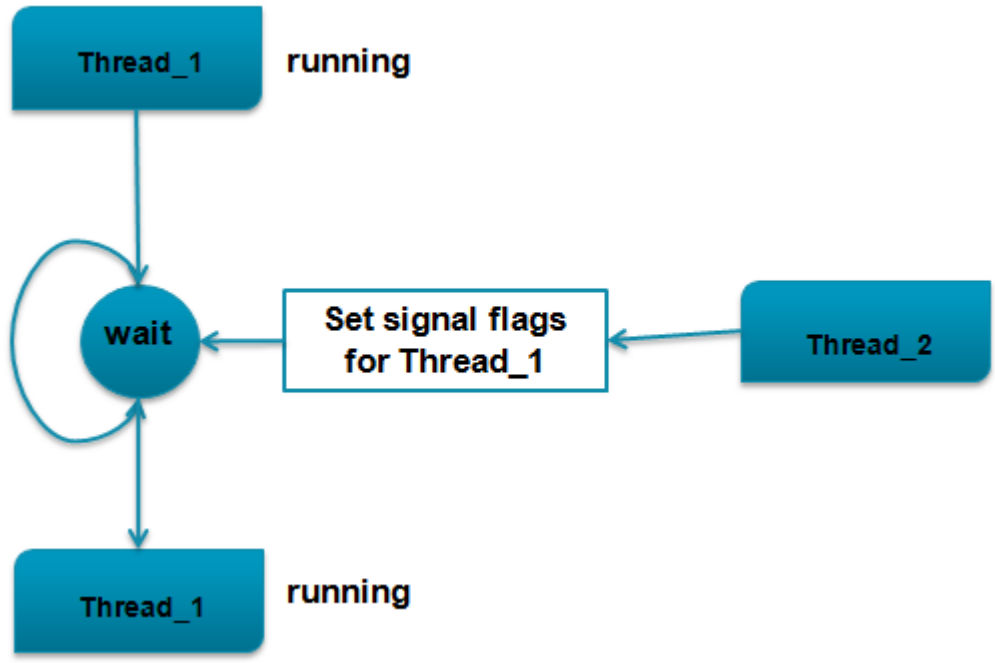


Figure 11. Event Flags

Each thread has upto 31 signal flags (set by a define in the CMSIS-RTOS API). A thread may be placed into a waiting state until a pattern of flags is set by another thread. When this happens, it will return to the READY state and wait to be scheduled by the kernel.

The `osSignalWait` call to the CMSIS-RTOS API suspends execution of the thread and places it into the WAITING state. We can wait for a group of event flags to be set or until one flag in a selected group is set. It is also possible to define a periodic timeout after which the waiting task will move back to the READY state, so that it can resume execution when selected by the scheduler. Additionally, a value of `osWaitForever` can be used to define an infinite timeout period.

Any thread can set the signal flags of any other thread with the CMSIS-RTOS API call `osSignalSet` or clear them with `osSignalClear`. The flags can also be set and cleared by Interrupt Service Routines using the same API calls The `Thread ID` is used to determine the target for those API calls.

When a thread resumes execution after it has been waiting for an `osSignalWait` function to complete, it may need to determine which event flag has been set. The `osSignalGet` function allows you to determine the signal flag that was set in the current running thread. You can then execute the correct code for this condition.

In the file `Ex2_RTX_rvents.c` look at line 52 for the function

```
void Button_IRQHandler (void)
```

This function `Button_IRQHandler()` is the handler for the Interrupt Service Routine used by RTX (the hardware level for the IRQ handler is dealt with in the HAL in button.c). In here we need to set an signal that will control the behavior of the `button_control` thread.

In the function void Button_IRQHandler (void) insert the following:

```
osSignalSet (T_button_control,0x0001);
```

We have now set an signal flag (flag 1) from inside the Interrupt Service Routine for the thread `button_control`. Now we can move on to the `button_control` thread content itself.

In the file `Ex2_RTX_events.c` find at line 87:

```
void button_control (void const *argument) {
```

You will notice there is some code in this thread already, the basics are there but we will now add the lines to deal with the signal flags.

The way we want this thread to run is as follows:

Upon the thread being created and run for the first time, it will enable the IRQ for the button and then go into a wait mode until the signal flag from the button interrupt is set. When this flag is set, the thread will continue to run – clear the signal flag and increment the counter variable, run a debounce delay for the button (during which time other threads can run) then enable the IRQ again (it was disabled by the IRQ request). It will then go back to the wait state (so other threads can run) until another interrupt occurs to set the signal flag. See figure 12 below for flow diagram.

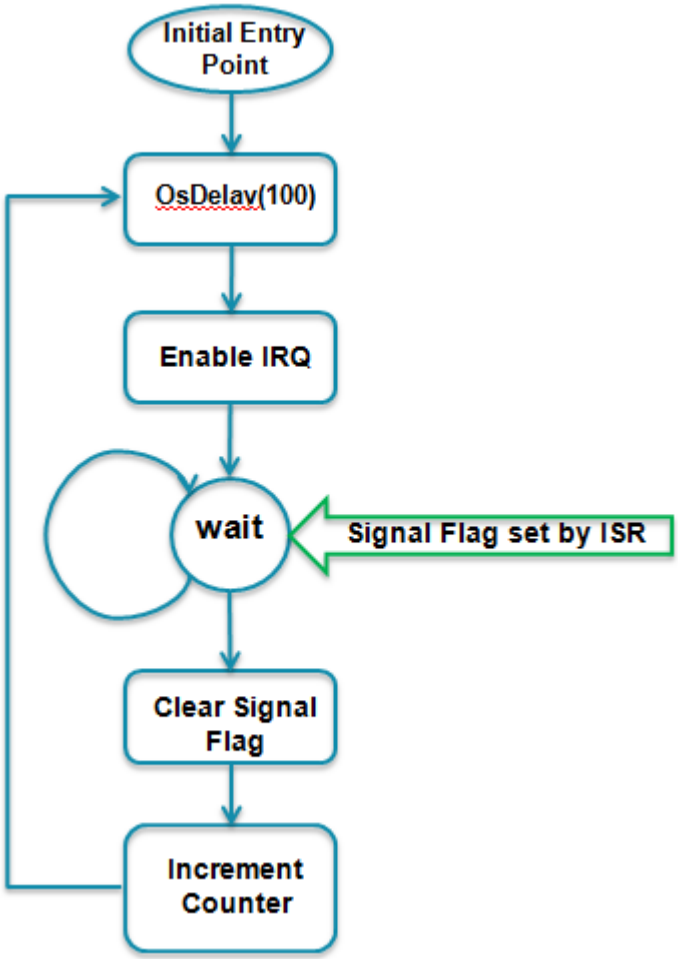


Figure 12. button_control thread Flow Diagram

In the `button_control` thread enter the following:

After the line `EnableButtonIRQ();` insert

```
osSignalWait      (0x0001,osWaitForever);
```

This tells the thread to wait for signal flag `0x0001`. The `osWaitForever` is the time to wait – in this case an infinite period – you could also specify a value of OS ticks here instead.

On the next line add

```
osSignalClear     ( T_button_control, 0x01);
```

This clears signal flag `0x0001` in thread `button_control`.

If you have entered all the lines correctly your `button_control` thread should look like this:

```
/*----------------------------------------------------------------------------
  Thread 3 'button_control': Responds to user input on hardware button
 *----------------------------------------------------------------------------*/
void button_control (void const *argument) {
  for (;;) {
    osDelay(100);                             // debounce button
    EnableButtonIRQ();                        // enable the IRQ for the button
    osSignalWait  ( 0x0001,osWaitForever);      // wait (forever) for Flag 0x0001
    osSignalClear ( T_button_control, 0x01);  // clear flag 0x0001
    button_counter++;
    }
}
```

Figure 13. Completed button_control thread for Session 2

Now you can compile the project and enter debug to see how it works!

Compile the project using the Rebuild All button 

Enter debug using the Start/Stop Debug Button 

If the project does not build, ask for help from your instructor.

The configuration of the project for entering debug is already setup, and is the same as for example 1. An additional variable has been added to the Watch 1 window to count and show the number of times the button gets pressed.

3.3 Analyze the program in Debug

In the debug environment we can see how this application behaves. The button on the target hardware we will use to generate the interrupt is labeled on the board as 'USER'. It is located on the bottom edge of the board on the left hand side.

When in the debug environment, open the RTX Task and System Window using the menu options:

Debug > OS Support > RTX Task and System

Notice the Watch 1 window has 4 variables in it, the two LED counters and the idle counter from session 1 and now a button press counter aswell.

Watch 1

Name	Value	Type
idle_count	0x00000000	int
LED_0_toggle_counter	0x00000000	int
LED_1_toggle_counter	0x00000000	int
button_counter	0x00000000	int
<Enter expression>		

Next, start the application running using the start button 

In the RTX Task and System window we can that see the tasks are correctly created and have the priority we set for them.

RTX Tasks and System

Property	Value
System	

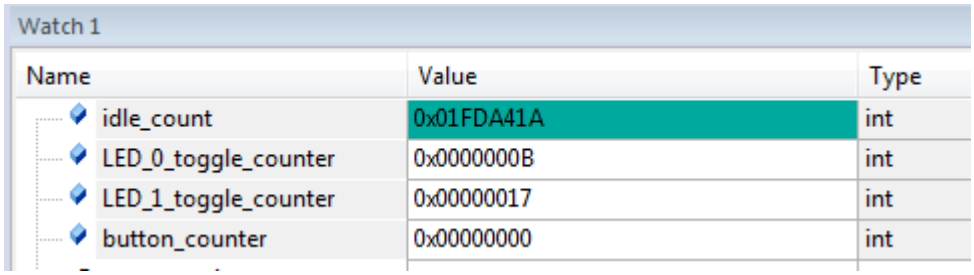
Item	Value
Timer Number:	0
Tick Timer:	1.000 mSec
Round Robin Timeout:	
Stack Size:	100
Tasks with User-provided Stack:	1
Stack Overflow Check:	Yes
Task Usage:	Available: 4, Used: 3
User Timers:	Available: 0, Used: 0

Tasks

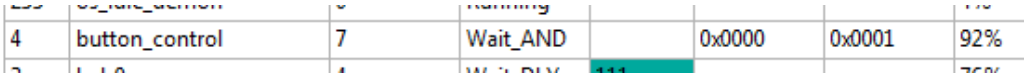
ID	Name	Priority	State	Delay	Event Value	Event Mask	Stack Load
255	os_idle_demon	0	Running				4%
4	button_control	7	Wait_AND		0x0000	0x0001	92%
3	led_0	4	Wait_DLY	370			76%
2	led_1	4	Wait_DLY	120			76%

Figure 14. RTX System Viewer in Session 2

With the application running you can see the variables in the watch window incrementing the same as in example 1.

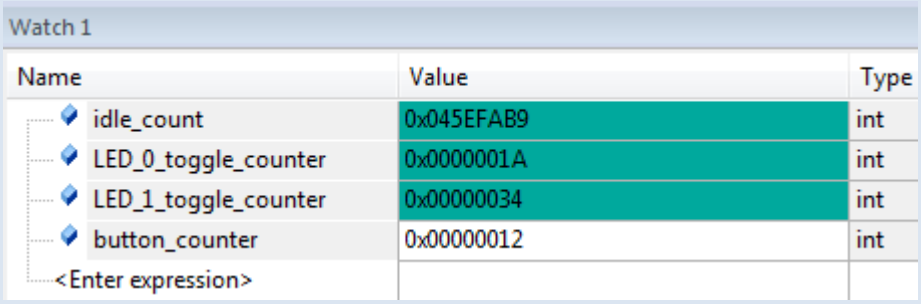


You can also see in the RTX system window that the thread button_control is sat waiting for a signal flag and you can verify that the signal mask is set to what you intended.



Press the USER button on the discovery board.

Note that the `button_counter` variable has incremented!

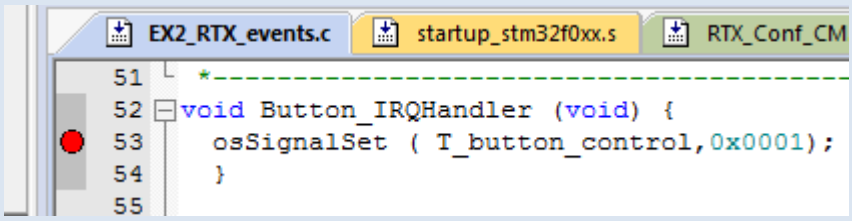


Press the button a few more times to see the variable increment – how exciting!!

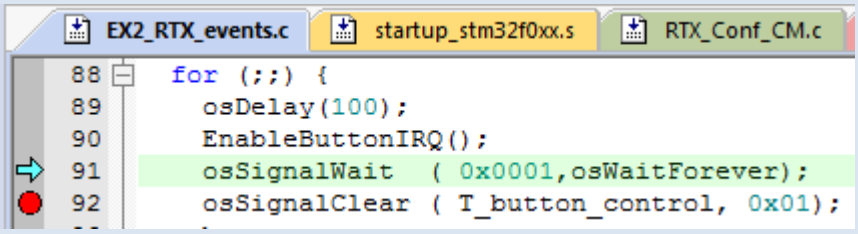
We can see the signals flags being cleared if we set a couple of breakpoints. We don't need to stop the application to add the breakpoints.

In the source window, ensure the file `Ex1_RTX_events.c` is in focus.

At line 53 insert a breakpoint (left click in the margin to the left of the source code)



Scroll to line 92 in the same file and insert another breakpoint



With the application running, press the USER button on the hardware, the first breakpoint (in the ISR handler) will be hit. At this point the signal flag is not set to the mask in the RTX system viewer still shows:

ID	Name	Priority	State	Delay	Event Value	Event Mask	Stack
255	os_idle_demon	0	Running				36%
4	button_control	7	Wait_AND		0x0000	0x0001	92%

Press the run button  to move on to the next breakpoint

Now we can see that the event flag has been set as we have been able to progress past the 'wait forever' condition inside the `button_control` thread. Also in the RTX system viewer we can see the pending signal mask has been satisfied.

ID	Name	Priority	State	Delay	Event Value	Event Mask	Stack
255	os_idle_demon	0	Ready				68%
4	button_control	7	Running				20%

Press the run button  again and you will see that the RTX system viewer once again shows the signal mask that the `button_control` thread is waiting for.

ID	Name	Priority	State	Delay	Event Value	Event Mask	Stack Load
255	os_idle_demon	0	Running				4%
4	button_control	7	Wait_AND		0x0000	0x0001	92%

This concludes session 2. We have covered the usage of interrupts from the system into an CMSIS-RTOs based application and we did not have to do anything 'special' to the interrupts to make them work, there were defined just the same as in a system that did not make use of an RTOS.

We have also seen how to use the signal system to control the behavior of threads in the RTOS application. In this example we used a real-world interrupt to control the behavior of one of the threads, but you could also pass these signals between threads inside the RTOS.

4.0 Workbook Conclusion

This workbook is not intended to make you an expert on using CMSIS-RTOS and RTX, but it does give you a solid base from which to start.

You now have first hand, practical experience of using both the CMSIS-RTOS API and the RTX real time operating system and have manipulated some of the major elements.

You also have good projects to use as a reference platform from which to build your own application.

Further information on using RTX can be read at:

www.keil.com/arm