

Introduction

This tutorial is an excerpt from “The Designers Guide to the Cortex-M Processor Family” by Trevor Martin and is reproduced with permission of Elsevier. For more details please see the Further Reading section at the end of this tutorial.

In this tutorial we are going to look at using a small footprint RTOS running on a Cortex-M based microcontroller. Specifically we are going to use an RTOS that meets the ‘Cortex Microcontroller Interface Standard’ (CMSIS) RTOS Specification. This specification defines a standard RTOS API for use with Cortex-M based microcontrollers. The CMSIS-RTOS API provides us with all the features we will need to develop with an RTOS, we only need to learn it once and then can use it across a very wide range of devices. CMSIS-RTOS also provides a standard interface for more complex frameworks (Java Virtual Machine, UML). It is also a standard interface for anyone wanting to develop reusable software components. If you are new to using an RTOS it takes a bit of practice to get used to working with an RTOS but once you have made the leap the advantages are such that you will not want to return to writing bare metal code.

Getting Started- Installing the tools

To run the examples in this tutorial, it is first necessary to install the MDK-ARM toolchain. First download the MDK-Core Version 5 using the embedded URL below and run the installation file.

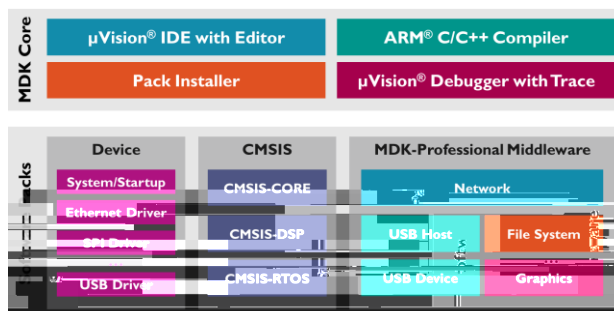
<http://www.keil.com/mdk5/install>

Download

Download MDK-Core Version 5

MDK-Lite Version 5 with 32KB Code Size limit. Use a current Product Serial Number (PSN) to activate other [MDK Editions](#).

This installs the core toolchain which includes the IDE, compiler/linker and the basic debugger. It does not include support for specific Cortex-M based microcontrollers. To support a given microcontroller family we need to install a ‘Device Family Pack’. This is a collection of support files such as startup code, flash programming algorithms and debugger support that allow you to develop with a specific microcontroller family.



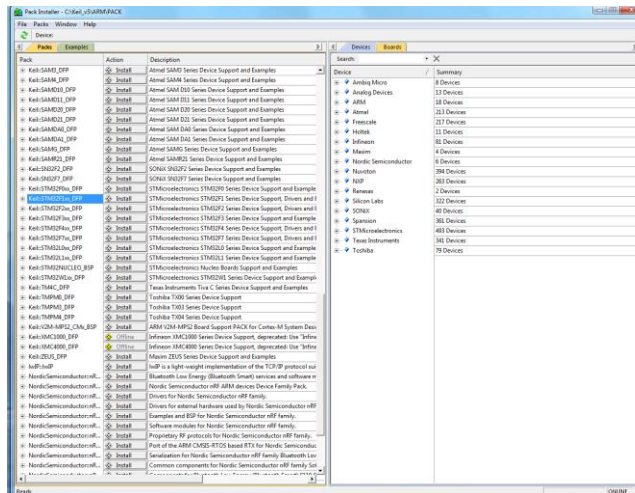
The MDK-ARM toolchain consists of a Core Installation (IDE, Compiler and Debugger) plus additional software packs added through a pack installer

In the exercises we are going to use an STM32F103RB so we need to install support for this device using the ‘Pack Installer’ within the μ Vision IDE. When the MDK-Core finishes installing the pack installer will start automatically, alternatively you can start the μ Vision IDE and access Pack Installer from the toolbar by pressing the icon shown below



Pack Installer Icon

Once the pack installer is open it will connect to cloud based pack database and display the available device packs.



The Pack Installer. Use this utility to install device support and third party software components

Select the Keil::STM32F1xx_DFP and press the install button. This will take a few minutes to download and install the STM32F1xx support files.

Keil::STM32F0xx_DFP		STMicroelectronics STM32F0 Series Device Support and Examples
Keil::STM32F1xx_DFP		STMicroelectronics STM32F1 Series Device Support, Drivers and Examples
Keil::STM32F2xx_DFP		STMicroelectronics STM32F2 Series Device Support, Drivers and Examples

Install support for the STM32F1xx Family

If the pack installer has any problems accessing the remote pack you can download it manually using the URL below

<http://www.keil.com/dd2/Pack/>

STMicroelectronics STM32F1 Series Device Support, Drivers and Examples

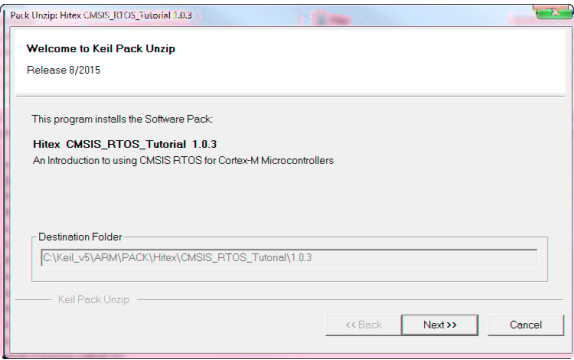
BSP DFP 1.1.0



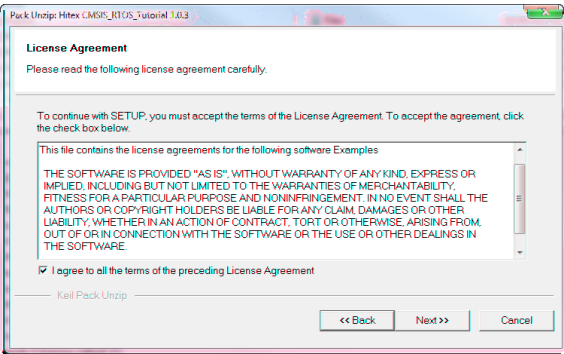
Again select the STM32F1xx pack and save it to your hard disk. The file may be saved as a .zip file depending on the browser you are using. If it is saved as a .zip change the .zip extension to .pack, you can then install it locally by double clicking on the STM32F1xx.pack file.

Installing the examples

The examples for this tutorial are provided as a CMSIS pack. You can install the pack into the MDK-ARM by simply double clicking on the Hitex.CMSIS_RTOS_Tutorial.1.0.3. pack file.

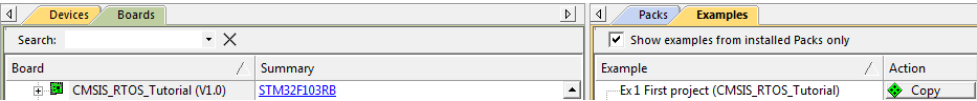


Once the pack has started installing click next



Here you must accept the license and again click next to continue the installation

Once the examples have been installed into MDK-ARM they are part of the toolchain and can be accessed through the pack installer. The tutorial examples can be found in the boards section under ‘CMSIS_RTOS_Tutorial’.



What Hardware do I need?

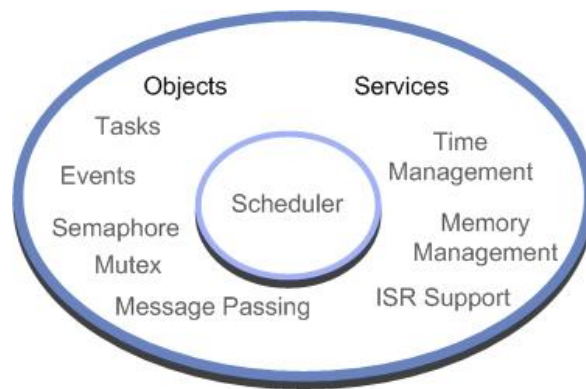
Simple answer: none! The Keil toolchain contains simulators for each of the Cortex-M processors. It also contains full simulation models (CPU + Peripherals) for some of the earlier Cortex-M microcontrollers. This means we can run the examples in the debugger using the simulation models and explore every aspect of using the RTOS. In fact this method of working is a better way of learning how to use the RTOS than going straight to a real microcontroller.

Overview

In this tutorial we will first look at setting up an introductory RTOS project for a Cortex-M based microcontroller. Next, we will go through each of the RTOS primitives and how they influence the design of our application code. Finally, when we have a clear understanding of the RTOS features, we will take a closer look at the RTOS configuration options. If you are used to programming a microcontroller without using an RTOS i.e. bare metal, there are two key things to understand as you work through this tutorial. In the first section we will focus on creating and managing Threads. The key concept here is to consider them running as parallel concurrent objects. In the second section we will look at how to communicate between threads. In this section the key concept is synchronization of the concurrent threads.

First steps with CMSIS-RTOS

The RTOS itself consists of a scheduler which supports round-robin, pre-emptive and co-operative multitasking of program threads, as well as time and memory management services. Inter-thread communication is supported by additional RTOS objects, including signal triggering, semaphores, mutex and a mailbox system. As we will see, interrupt handling can also be accomplished by prioritized threads which are scheduled by the RTOS kernel.



The RTOS kernel contains a scheduler that runs program code as tasks. Communication between tasks is accomplished by RTOS objects such as events, semaphores, mutexes and mailboxes. Additional RTOS services include time and memory management and interrupt support.

Accessing the CMSIS-RTOS API

To access any of the CMSIS-RTOS features in our application code it is necessary to include the following header file

```
#include <cmsis_os.h>
```

This header file is maintained by ARM as part of the CMSIS-RTOS standard. For the CMSIS-RTOS Keil RTX this is the default API. Other RTOS will have their own proprietary API but may provide a wrapper layer to implement the CMSIS-RTOS API so they can be used where compatibility with the CMSIS standard is required.

Threads

The building blocks of a typical 'C' program are functions which we call to perform a specific procedure and which then return to the calling function. In CMSIS-RTOS the basic unit of execution is a "Thread". A Thread is very similar to a 'C' procedure but has some very fundamental differences.

```

unsigned int procedure (void)
{
    .....
    return(ch);
}

void thread (void)
{
    while(1)
    {
        .....
    }
}

```

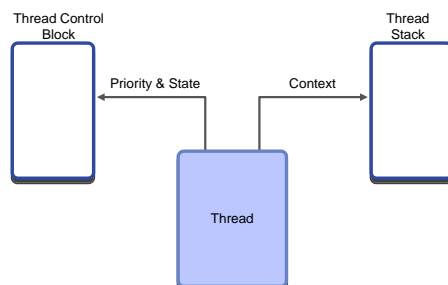
While we always return from our 'C' function, once started an RTOS thread must contain a loop so that it never terminates and thus runs forever. You can think of a thread as a mini self-contained program that runs within the RTOS.

An RTOS program is made up of a number of threads, which are controlled by the RTOS scheduler. This scheduler uses the SysTick timer to generate a periodic interrupt as a time base. The scheduler will allot a certain amount of execution time to each thread. So thread1 will run for 5ms then be de-scheduled to allow thread2 to run for a similar period; thread 2 will give way to thread3 and finally control passes back to thread1. By allocating these slices of runtime to each thread in a round-robin fashion, we get the appearance of all three threads running in parallel to each other.

Conceptually we can think of each thread as performing a specific functional unit of our program with all threads running simultaneously. This leads us to a more object-orientated design, where each functional block can be coded and tested in isolation and then integrated into a fully running program. This not only imposes a structure on the design of our final application but also aids debugging, as a particular bug can be easily isolated to a specific thread. It also aids code reuse in later projects. When a thread is created, it is also allocated its own thread ID. 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.

```
osThreadId id1,id2,id3;
```

In order to make the thread-switching process happen, we have the code overhead of the RTOS and we have to dedicate a CPU hardware timer to provide the RTOS time reference. In addition, each time we switch running threads, we have to save the state of all the thread variables to a thread stack. Also, all the runtime information about a thread is stored in a thread control block, which is managed by the RTOS kernel. Thus the "context switch time", that is, the time to save the current thread state and load up and start the next thread, is a crucial figure and will depend on both the RTOS kernel and the design of the underlying hardware.



Each thread has its own stack for saving its data during a context switch. The thread control block is used by the kernel to manage the active thread.

The Thread Control Block contains information about the status of a thread. Part of this information is its run state. In a given system only one thread can be running and all the others will be suspended but ready to run. The RTOS has various methods of inter-thread communication (signals, semaphores, messages). Here, a thread may be suspended to wait to be signaled by another thread or interrupt before it resumes its ready state, whereupon it can be placed into running state by the RTOS scheduler.

Running	The Currently Running Thread
Ready	Threads ready to Run
Wait	Blocked Threads waiting for an OS Event

At any given moment a single thread may be running. The remaining threads will be ready to run and will be scheduled by the kernel. Threads may also be waiting pending an OS event. When this occurs they will return to the ready state and be scheduled by the kernel.

Starting the RTOS

To build a simple RTOS program we declare each thread as a standard ‘C’ function and also declare a thread ID variable for each function.

```
void thread1 (void);  
void thread2 (void);  
  
osThreadId thrdID1, thrdID2;
```

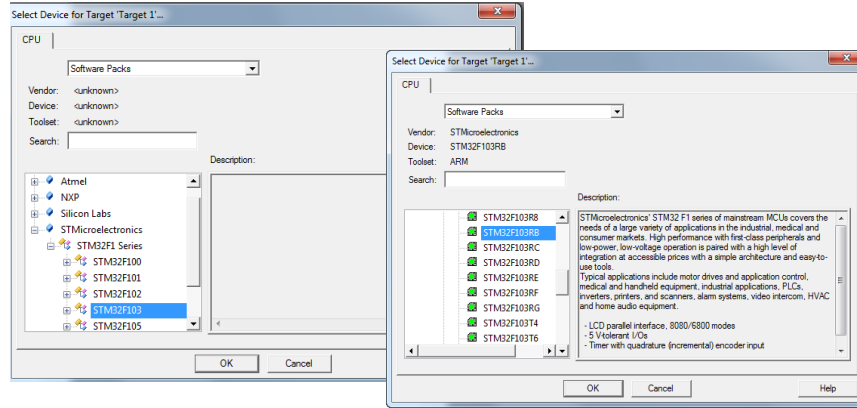
By default the CMSIS-RTOS scheduler will be running when main() is entered and the main() function becomes the first active thread. Once in main(), we can stop the scheduler task switching by calling osKernelInitialize (). While the RTOS is halted we can create further threads and other RTOS objects. Once the system is in a defined state we can restart the RTOS scheduler with osKernelStart().

You can run any initializing code you want before starting the RTOS.

```
void main (void)  
{  
    osKernelInitialize ();  
    IODIR1 = 0x00FF0000;           // Do any C code you want  
    Init_Thread();                 // Create a Thread  
    osKernelStart();               // Start the RTOS  
}
```

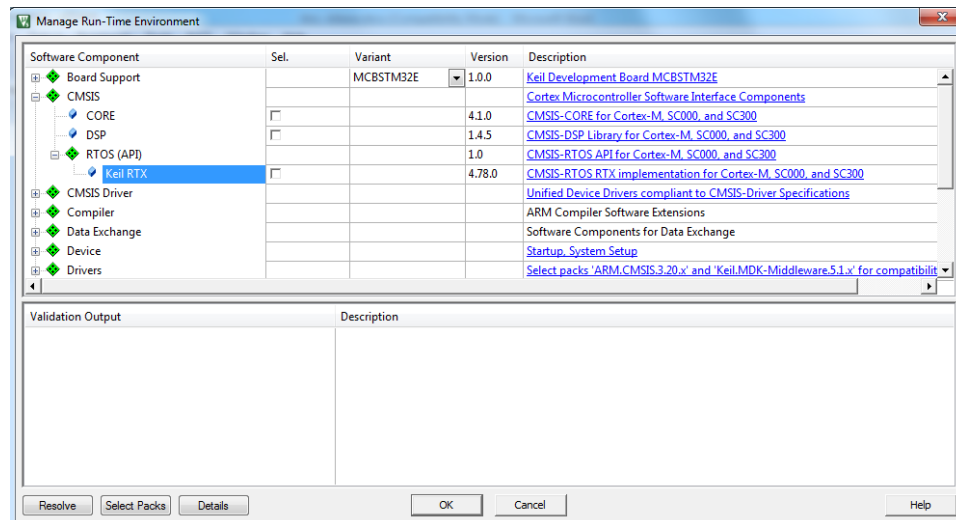
When threads are created they are also assigned a priority.

CMSIS-RTOS Tutorial



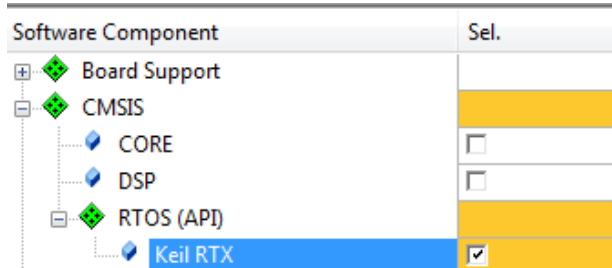
Once you have selected this device click ok.

Once the microcontroller variant has been selected the Run Time Environment Manager will open.



This allows you to configure the platform of software components you are going to use in a given project. As well as displaying the available components the RTE understands their dependencies on other components.

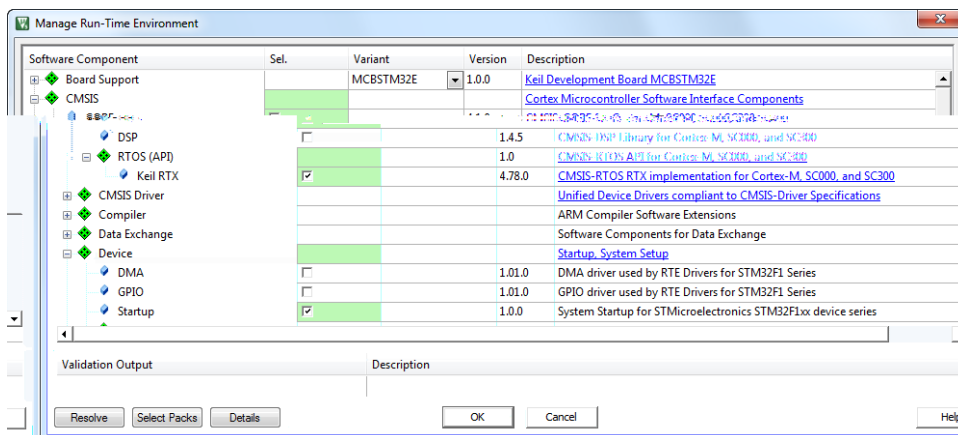
To configure the project for use with the CMSIS-RTOS Keil RTX, simply tick the CMSIS::RTOS (API):Keil RTX box.



This will cause the selection box to turn orange meaning that additional components are required. The required component will be displayed in the Validation Output window.

Validation Output	Description
ARM::CMSIS:RTOS:Keil RTX	Additional software components required
require Device:Startup	Select component from list
Keil::Device:Startup	System Startup for STMicroelectronics STM32F1xx device series

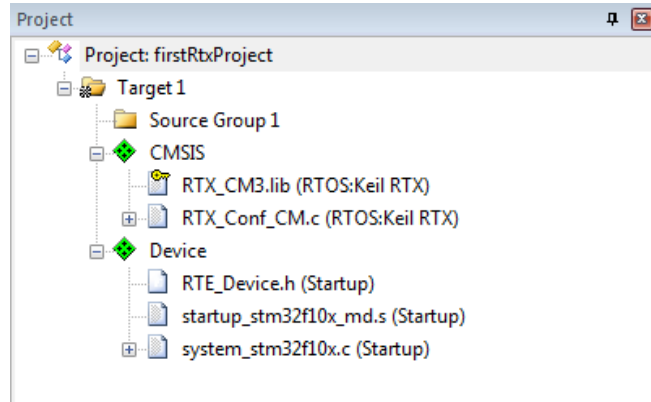
To add the missing components you can press the Resolve button in the bottom left hand corner of the RTE. This will add the device startup code and the CMSIS Core support. When all the necessary components are present the selection column will turn green.



It is also possible to access a components help files by clicking on the blue hyperlink in the Description column.

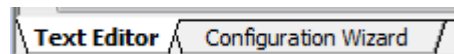
Now press the OK button and all the selected components will be added to the new project

CMSIS-RTOS Tutorial

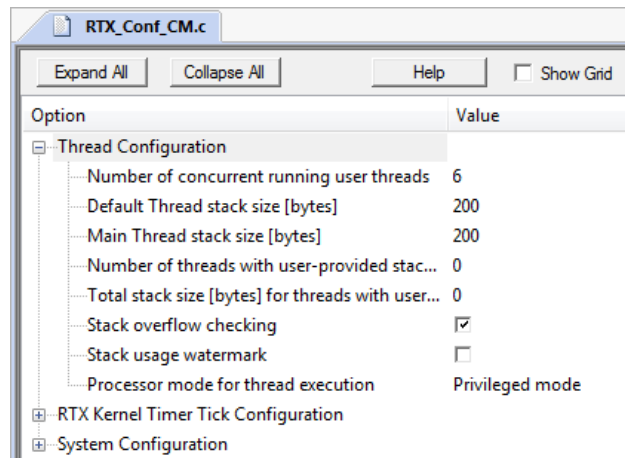


The CMSIS components are added to folders displayed as a green diamond. There are two types of file here. The first type is a library file which is held within the toolchain and is not editable. This file is shown with a yellow key to show that it is 'locked' (read-only). The second type of file is a configuration file. These files are copied to your project directory and can be edited as necessary. Each of these files can be displayed as a text files but it is also possible to view the configuration options as a set of pick lists and drop down menus.

To see this open the RTX_Conf_CM.c file and at the bottom of the editor



Click on **Expand All** to see all of the configuration options as a graphical pick list:



For now it is not necessary to make any changes here and these options will be examined towards the end of this tutorial.

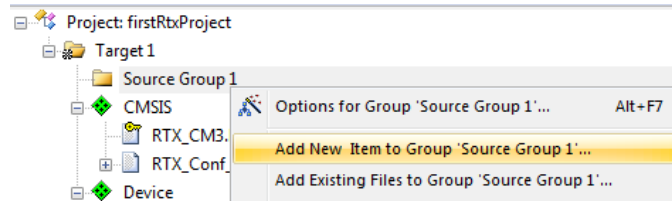
Our project contains four configuration files three of which are standard CMSIS files

Startup_STM32F10x_md.s	Assembler vector table
System_STM32F10x.c	C code to initialize key system peripherals, such as clock tree, PLL external memory interface.
RTE_Device.h	Configures the pin multiplex
RTX_Conf_CM.c	Configures Keil RTX

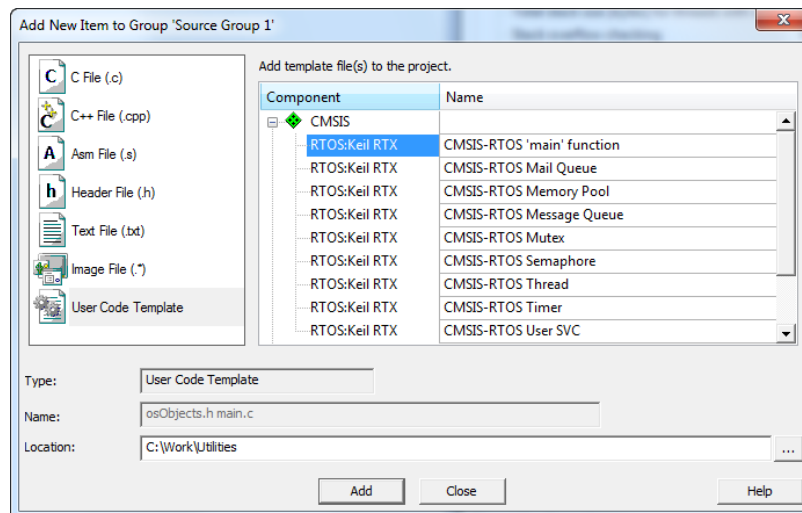
Now that we have the basic platform for our project in place we can add some user source code which will start the RTOS and create a running thread.

**To do this right-d d ! f! T df!H !2!g ef !b e! f fd! Bee! f ! f !
!T df!H !2**

CMSIS-RTOS Tutorial

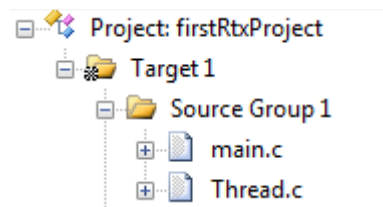


! f!Bee! f ! f !e b h! f fd! f! V f !d ef! f bf! d !b e! ! f!
DN T T! fd ! f fd! f! DN T T-S UP T! b !g d !b e!d d !Bee



S f fb! !c ! ! f! f fd! DN T T-S UP T!U fbe !

This will now add two source files to our project main.c and thread.c



Open thread.c in the editor

We will look at the RTOS definitions in this project in the next section. For now this file contains two functions `Init_Thread()` which is used to start the thread running and the actual thread function.

Copy the Init_Thread function prototype and then open main.c

Main contains the functions to initialize and start the RTOS kernel. Then unlike a bare metal project main is allowed to terminate rather than enter an endless loop. However this is not really recommended and we will look at a more elegant way of terminating a thread later.

In main.c add the Init_Thread prototype as an external declaration and then call it after the osKernelInitialize function as shown below.

```
#define osObjectsPublic

#include "osObjects.h"

extern int Init_Thread (void);          //Add this line

int main (void) {

    osKernelInitialize ();

    Init_Thread ();                    //Add this line

    osKernelStart ();

}
```

Build the project (F7)

In this tutorial we can use the debugger simulator to run the code without the need for any external hardware.

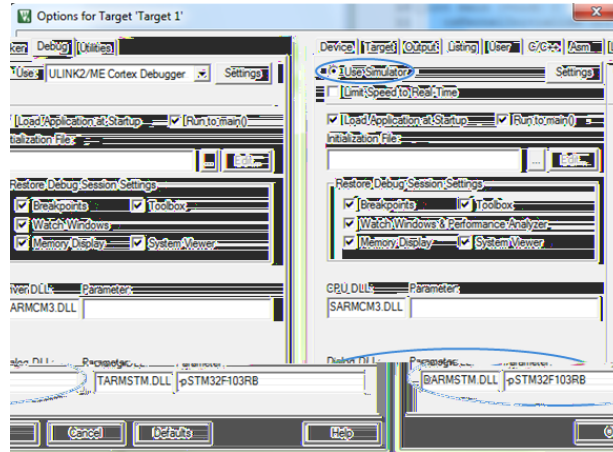
Highlight the Target 1 root folder, right click and select options for target 1

Select the debugger tab

This menu is in two halves the left side configures the simulator the right half configures the hardware debugger

Select the Simulator radio button and check that Eialog EMM is set to DARMSTM.DLL with parameter -pSTM32F103RB

CMSIS-RTOS Tutorial



Click ok to close the options for target menu

Start the debugger (Ctrl+F5)

This will run the code up to main

Open the Debug OS Support System and Thread Viewer

System and Thread Viewer							
Property	Value						
	Item	Value					
	Tick Timer:	1.000 mSec					
	Round Robin Timeout:	5.000 mSec					
	Default Thread Stack Size:	200					
	Thread Stack Overflow Check:	Yes					
	Thread Usage:	Available: 7, Used: 3 + os_idle_demon					
ID	Name	Priority	State	Delay	Event Value	Event Mask	Stack Usage
1	osTimerThread	High	Wait_MBX				32%
2	main	Normal	Running				0%
255	os_idle_demon	None	Ready				

This debug view shows all the running threads and their current state. At the moment we have three threads which are main, os_idle_demon and osTimerThread.

Start the code running (F5)

ID	Name	Priority	State	Delay	Event Value	Event Mask	Stack Usage
1	osTimerThread	High	Wait_MBX				32%
3	Thread	Normal	Running				16%
255	os_idle_demon	None	Ready				32%

Now the user thread is created and main is terminated.

Exit the debugger

While this project does not actually do anything it demonstrates the few steps necessary to start using CMSIS-RTOS

Creating Threads

Once the RTOS is running, there are a number of system calls that are used to manage and control the active threads. By default, the `main()` function is automatically created as the first thread running. In the first example we used it to create an additional thread then let it terminate by running through the closing brace. However, if we want to we can continue to use main as a thread in its own right. If we want to control main as a thread we must get its thread ID. The first RTOS function we must therefore call is `osThreadGetId()` which returns the thread ID number of the currently running thread. This is then stored in its ID handle. When we want to refer to this thread in future OS calls, we use this handle rather than the function name of the thread.

```
osThreadId main_id; //create the thread handle

void main (void)

{

/* Read the Thread-ID of the main thread */

main_id = osThreadGetId ();

while(1)

{

.....

}

}
```

Now that we have an ID handle for main we could create the application threads and then call `osTerminate(main_id)` to end the main thread. This is the best way to end a thread rather than let it run off the end of the closing brace. Alternatively

we can add a while(1) loop as shown above and continue to use main in our application.

As we saw in the first example the main thread is used as a launcher thread to create the application threads. This is done in two stages. First a thread structure is defined; this allows us to define the thread operating parameters.

```
osThreadId thread1_id;                //thread handle

void thread1 (void const *argument);   //function prototype for thread1

osThreadDef(thread1, osPriorityNormal, 1, 0); //thread definition structure
```

The thread structure requires us to define the name of the thread function, its thread priority, the number of instances of the thread that will be created, and its stack size. We will look at these parameters in more detail later. Once the thread structure has been defined the thread can be created using the osThreadCreate() API call. Then the thread is created from within the application code, this is often the within the main thread but can be at any point in the code.

```
thread1_id = osThreadCreate(osThread(thread1), NULL);
```

This creates the thread and starts it running. It is also possible to pass a parameter to the thread when it starts.

```
uint32_t startupParameter = 0x23;

thread1_id = osThreadCreate(osThread(thread1), startupParameter);
```

When each thread is created, it is also assigned its own stack for storing data during the context switch. This should not be confused with the native Cortex processor stack; it is really a block of memory that is allocated to the thread. A default stack size is defined in the RTOS configuration file (we will see this later) and this amount of memory will be allocated to each thread unless we override it to allocate a custom size. The default stack size will be assigned to a thread if the stack size value in the thread definition structure is set to zero. If necessary a thread can be given additional memory resources by defining a bigger stack size in the thread structure.

```
osThreadDef(thread1, osPriorityNormal, 1, 0); //assign default stack size to this thread

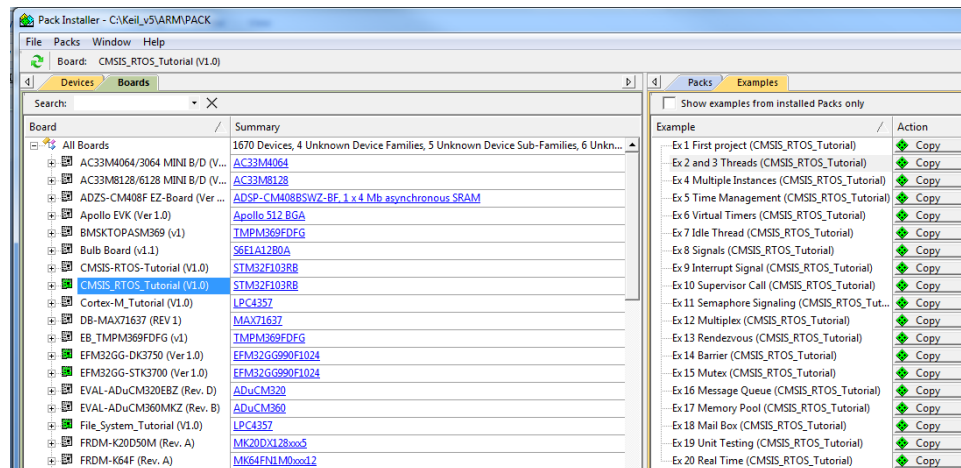
osThreadDef(thread2, osPriorityNormal, 1, 1024); //assign 1KB of stack to this thread
```

However, if you allocate a larger stack size to a thread then the additional memory must be allocated in the RTOS configuration file; again we will see this later.

Exercise creating and managing threads

In this project we will create and manage some additional threads. Each of the threads created will toggle a GPIO pin on GPIO port B to simulate flashing an LED. We can then view this activity in the simulator.

To access the exercise projects open the pack installer from within μ Vision.



Select the boards tab and select the CMSIS-RTOS_Tutorial

Now select the examples tab and all the example projects for this tutorial will be shown.

U!e b! f! ! ef!d d! ! f!h f! F b f!d ! fbef

A reference copy of the first exercise is included as Exercise 1

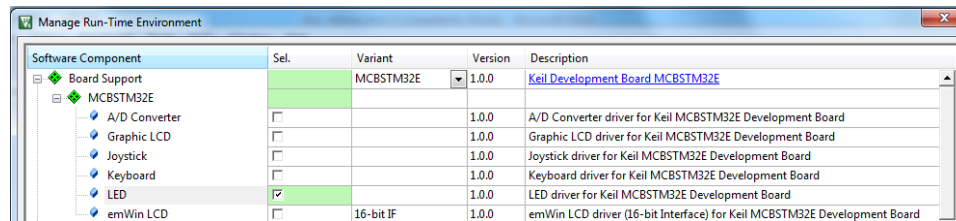
Tf fd! F !3!b e!4!U fbe !b e! f ! f!d !c

This will install the project to a directory of your choice and open the project in μ Vision.

Open the Run Time Environment Manager



In the board support section the MCBSTM32E:LED box is ticked. This adds support functions to control the state of a bank of LED's on the Microcontroller's GPIO port B.



When the RTOS starts main() runs as a thread and in addition we will create two additional threads. First we create handles for each of the threads and then define the parameters of each thread. These include the priority the thread will run at, the number of instances of each thread we will create and its stack size (the amount of memory allocated to it) zero indicates it will have the default stack size.

```
osThreadId main_ID,led_ID1,led_ID2;

osThreadDef(led_thread2, osPriorityNormal, 1, 0);

osThreadDef(led_thread1, osPriorityNormal, 1, 0);
```

Then in the main() function the two threads are created

```
led_ID2 = osThreadCreate(osThread(led_thread2), NULL);
led_ID1 = osThreadCreate(osThread(led_thread1), NULL);
```

When the thread is created we can pass it a parameter in place of the NULL define.

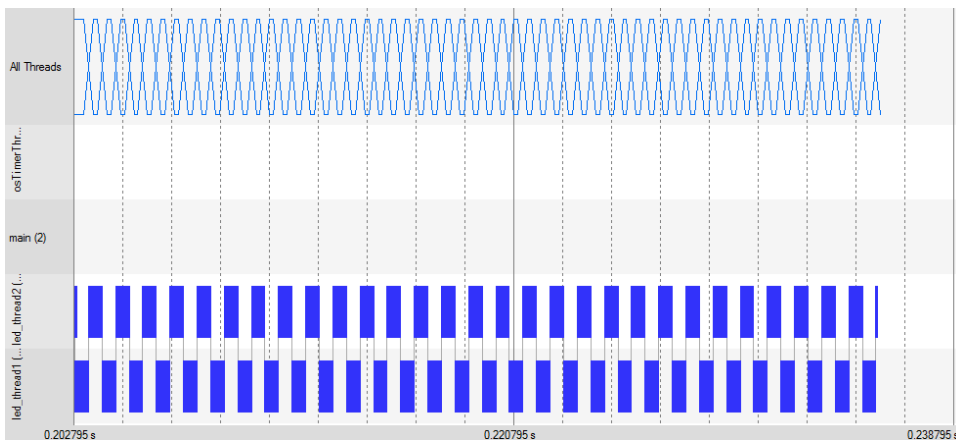
Build the project and start the debugger

Start the code running and open the Debug OS Support System and Thread Viewer

ID	Name	Priority	State	Delay	Event Value	Event Mask	Stack Usage
1	osTimerThread	High	Wait_MBX				32%
3	led_thread2	Normal	Running				0%
4	led_thread1	Normal	Ready				32%
255	os_idle_demon	None	Ready				

Now we have four active threads with one running and the others ready.

Open the Debug OS Support Event Viewer



The event viewer shows the execution of each thread as a trace against time. This allows you to visualize the activity of each thread and get a feel for amount of CPU time consumed by each thread.

Now open the Peripherals General Purpose IO GPIOB window

GPIOB		15	Bits	8	7	Bits	0
GPIOB_IDR:	0x00000300	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
GPIOB_ODR:	0x00000300	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
GPIOB_LCKR:	0x00000000	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Pins:	0x00000300	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Our two led threads are each toggling a GPIO port pin. Leave the code running and watch the pins toggle for a few seconds.

If you do not see the debug windows updating check the view\periodic window update option is ticked.

```
void led_thread2 (void const *argument) {  
  
    for (;;) {  
  
        LED_On(1);  
  
        delay(500);  
  
        LED_Off(1);  
  
        delay(500);  
  
    }  
}
```

Each thread calls functions to switch an LED on and off and uses a delay function between each on and off. Several important things are happening here. First the delay function can be safely called by each thread. Each thread keeps local variables in its stack so they cannot be corrupted by any other thread. Secondly none of the threads enter a descheduled waiting state, this means that each one runs for its full allocated time slice before switching to the next thread. As this is a simple thread most of its execution time will be spent in the delay loop effectively wasting cycles. Finally there is no synchronization between the threads. They are running as separate ‘programs’ on the CPU and as we can see from the GPIO debug window the toggled pins appear random.

Thread Management and Priority

When a thread is created it is assigned a priority level. The RTOS scheduler uses a thread’s priority to decide which thread should be scheduled to run. If a number of threads are ready to run, the thread with the highest priority will be placed in the run state. If a high priority thread becomes ready to run it will preempt a running thread of lower priority. Importantly a high priority thread running on the CPU will not stop running unless it blocks on an RTOS API call or is preempted by a higher priority thread. A thread’s priority is defined in the thread structure and the following priority definitions are available. The default priority is `osPriorityNormal`

CMSIS-RTOS Priority Levels
osPriorityIdle
osPriorityLow
osPriorityBelowNormal
osPriorityNormal
osPriorityAboveNormal
osPriorityHigh
osPriorityRealTime
osPriorityError

Once the threads are running, there are a small number of OS system calls which are used to manage the running threads. It is also then possible to elevate or lower a thread's priority either from another function or from within its own code.

```
osStatus osThreadSetPriority(threadID, priority);
```

```
osPriority osThreadGetPriority(threadID);
```

As well as creating threads, it is also possible for a thread to delete itself or another active thread from the RTOS. Again we use the thread ID rather than the function name of the thread.

```
osStatus = osThreadTerminate (threadID1);
```

Finally, there is a special case of thread switching where the running thread passes control to the next ready thread of the same priority. This is used to implement a third form of scheduling called co-operative thread switching.

```
osStatus osThreadYield();//switch to next ready to run thread
```

Exercise creating and managing threads II

In this exercise we will look at assigning different priorities to threads and also how to create and terminate threads dynamically.

Go back to the project **F !3! b e!4! U fbe** Change the priority of LED Thread 2 to Above Normal

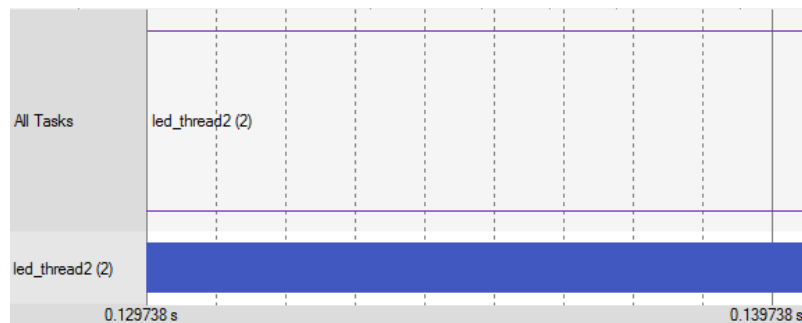
```
osThreadDef(led_thread2, osPriorityAboveNormal, 1, 0);
```

```
osThreadDef(led_thread1, osPriorityNormal, 1, 0);
```

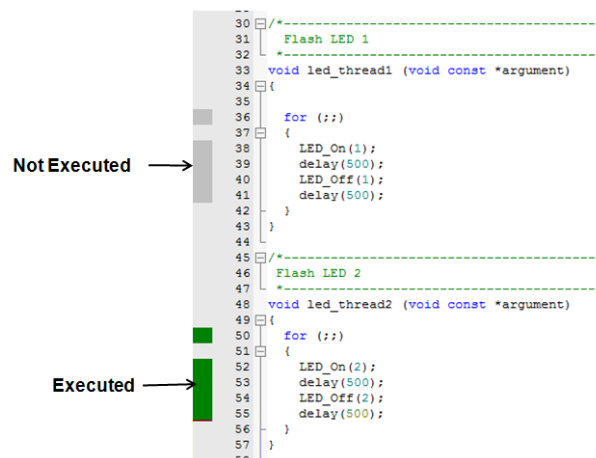
Build the project and start the debugger

Start the code running

Open the Debug OS Support Event Viewer window



Here we can see thread2 running but no sign of thread1. Looking at the coverage monitor for the two threads shows us that led_thread1 has not run.



Led_thread1 is running at normal priority and led_thread2 is running at a higher priority so has pre-empted led_thread1. To make it even worse led_thread2 never blocks so it will run forever preventing the lower priority thread from ever running.

Although this error may seem obvious in this example this kind of mistake is very common when designers first start to use an RTOS.

Multiple Instances

One of the interesting possibilities of an RTOS is that you can create multiple running instances of the same base thread code. So for example you could write a thread to control a UART and then create two running instances of the same thread code. Here each instance of the UART code could manage a different UART.

First we create the thread structure and set the number of thread instances to two;

```
osThreadDef(thread1, osPriorityNormal, 2, 0);
```

Then we can create two instances of the thread assigned to different thread handles. A parameter is also passed to allow each instance to identify which UART it is responsible for.

```
ThreadID_1_0 = osThreadCreate(osThread(thread1), UART1);  
ThreadID_1_1 = osThreadCreate(osThread(thread1), UART2);
```

Exercise Multiple thread instances

In this project we will look at creating one thread and then create multiple runtime instances of the same thread.

In the Pack Installer select Ex 4 Multiple Instances and copy it to your tutorial directory.

This project performs the same function as the previous LED flasher program. However we now have one led switcher function that uses an argument passed as a parameter to decide which LED to flash.

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



```
    for (;;) {  
        LED_On((uint32_t)argument);  
        delay(500);  
        LED_Off((uint32_t)argument);  
        delay(500);  
    }  
}
```

When we define the thread we adjust the instances parameter to two.

```
osThreadDef(ledSwitcher, osPriorityNormal, 2, 0);
```

Then in the main thread we create two threads which are different instances of the same base code. We pass a different parameter which corresponds to the led that will be toggled by the instance of the thread.

```
led_ID1 = osThreadCreate(osThread(ledSwitcher),(void *) 1UL);  
led_ID2 = osThreadCreate(osThread(ledSwitcher),(void *) 2UL);
```

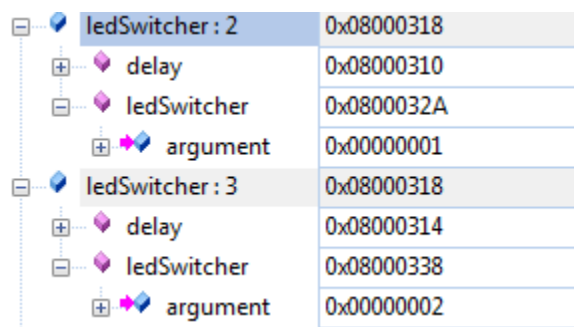
Build the code and start the debugger

Start the code running and open the RTX tasks and system window

ID	Name	Priority	State
255	os_idle_demon	0	Ready
3	ledSwitcher	4	Ready
2	ledSwitcher	4	Running

Here we can see both instances of the ledSwitcher task each with a different ID.

Examine the Call stack + locals window



ledSwitcher : 2	0x08000318
+ delay	0x08000310
- ledSwitcher	0x0800032A
+ argument	0x00000001
ledSwitcher : 3	0x08000318
+ delay	0x08000314
- ledSwitcher	0x08000338
+ argument	0x00000002

Here we can see both instances of the ledSwitcher threads and the state of their variables. A different argument has been passed to each instance of the thread.

Time Management

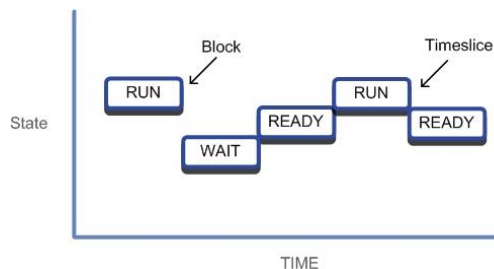
As well as running your application code as threads, the RTOS also provides some timing services which can be accessed through RTOS system calls.

Time Delay

The most basic of these timing services is a simple timer delay function. This is an easy way of providing timing delays within your application. Although the RTOS kernel size is quoted as 5k bytes, features such as delay loops and simple scheduling loops are often part of a non-RTOS application and would consume code bytes anyway, so the overhead of the RTOS can be less than it immediately appears.

void **osDelay** (uint32_t millisec)

This call will place the calling thread into the WAIT_DELAY state for the specified number of milliseconds. The scheduler will pass execution to the next thread in the READY state.



During their lifetime threads move through many states. Here a running thread is blocked by an `osDelay` call so it enters a wait state. When the delay expires, it moves to ready. The scheduler will place it in the run state. If its timeslice expires, it will move back to ready.

When the timer expires, the thread will leave the `wait_delay` state and move to the `READY` state. The thread will resume running when the scheduler moves it to the `RUNNING` state. If the thread then continues executing without any further blocking OS calls, it will be descheduled at the end of its time slice and be placed in the ready state, assuming another thread of the same priority is ready to run.

Waiting for an Event

In addition to a pure time delay it is possible to make a thread halt and enter the waiting state until the thread is triggered by another RTOS event. RTOS events can be a signal, message or mail event. The `osWait()` API call also has a timeout period defined in `millisec` that allows the thread to wake up and continue execution if no event occurs.

```
osStatus osWait (uint32_t millisec )
```

When the interval expires, the thread moves from the wait to the READY state and will be placed into the running state by the scheduler. `osWait` is an optional api call within the CMSIS RTOS specification. If you intend to use this function you must first check it is supported by the RTOS you are using. The `osWait` API call is not supported by the Keil RTX RTOS.

Exercise Time Management

In this exercise we will look at using the basic time delay function

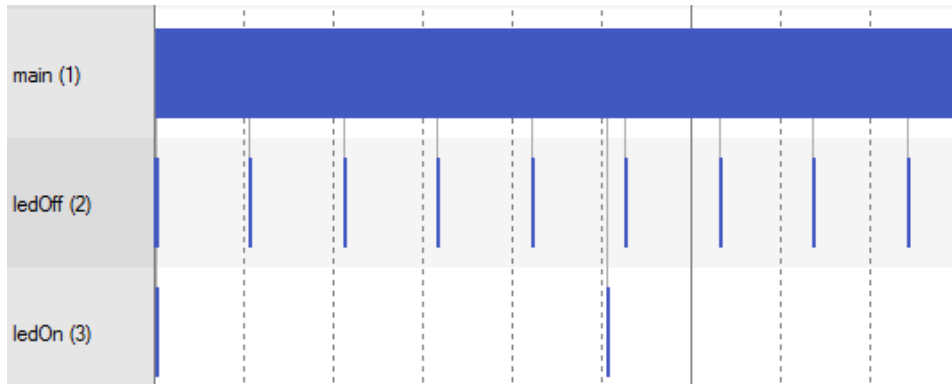
In the Pack Installer select `F !6!U f!N b bhf f` and copy it to your tutorial directory.

This is our original led flasher program but the simple delay function has been replaced by the `osDelay` API call. LED2 is toggled every 100mS and LED1 is toggled every 500mS

```
void ledOn (void const *argument) {  
  
    for (;;) {  
  
        LED_On(1);  
  
        osDelay(500);  
  
        LED_Off(1);  
  
        osDelay(500);  
  
    }  
}
```

Build the project and start the debugger

Start the code running and open the event viewer window



Now we can see that the activity of the code is very different. When each of the LED tasks reaches the `osDelay` API call it 'blocks' and moves to a waiting state. The main task will be in a ready state so the scheduler will start it running. When the delay period has timed out the led tasks will move to the ready state and will be placed into the running state by the scheduler. This gives us a multi threaded program where CPU runtime is efficiently shared between tasks.

Virtual Timers

The CMSIS-RTOS API can be used to define any number of virtual timers which act as count down timers. When they expire, they will run a user call-back function to perform a specific action. Each timer can be configured as a one shot or repeat timer. A virtual timer is created by first defining a timer structure.

```
osTimerDef(timer0,led_function);
```

This defines a name for the timer and the name of the call back function. The timer must then be instantiated in an RTOS thread.

```
osTimerId timer0_handle = osTimerCreate (timer(timer0), osTimerPeriodic, (void *)0);
```

This creates the timer and defines it as a periodic timer or a single shot timer (`osTimerOnce`). The final parameter passes an argument to the call back function when the timer expires.

```
osTimerStart ( timer0_handle,0x100);
```

The timer can then be started at any point in a thread the timer start function invokes the timer by its handle and defines a count period in milliseconds.

Exercise Virtual timer

In this exercise we will configure a number of virtual timers to trigger a callback function at various frequencies

In the Pack Installer select F!7W b!U f and copy it to your tutorial directory.

This is our original led flasher program and code has been added to create four virtual timers to trigger a callback function. Depending on which timer has expired, this function will toggle an additional LED.

The timers are defined at the start of the code

```
osTimerDef(timer0_handle, callback);
```

```
osTimerDef(timer1_handle, callback);
```

```
osTimerDef(timer2_handle, callback);
```

```
osTimerDef(timer3_handle, callback);
```

They are then initialized in the main function

```
osTimerId timer0 = osTimerCreate(osTimer(timer0_handle), osTimerPeriodic, (void *)0);
```

```
osTimerId timer1 = osTimerCreate(osTimer(timer1_handle), osTimerPeriodic, (void *)1);
```

```
osTimerId timer2 = osTimerCreate(osTimer(timer2_handle), osTimerPeriodic, (void *)2);
```

```
osTimerId timer3 = osTimerCreate(osTimer(timer3_handle), osTimerPeriodic, (void *)3);
```

Each timer has a different handle and ID and passed a different parameter to the common callback function

```
void callback(void const *param){
```

```
switch( (uint32_t) param){
```

```
case 0:
```

```
    GPIOB->ODR ^= 0x8;
```

```
break;

case 1:

    GPIOB->ODR ^= 0x4;

break;

case 2:

    GPIOB->ODR ^= 0x2;

break;

case 3:

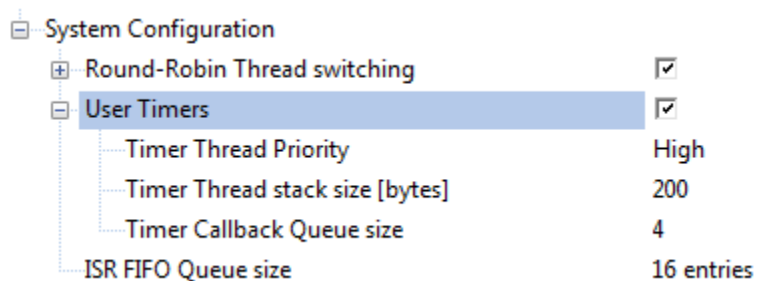
break;

}
```

When triggered, the callback function uses the passed parameter as an index to toggle the desired LED.

In addition to the configuring the virtual timers in the source code, the timer thread must be enabled in the RTX configuration file.

Open the RTX_Conf_CM.c file and press the configuration wizard tab



In the system configuration section make sure the User Timers box is ticked. If this thread is not created the timers will not work.

Build the project and start the debugger

Run the code and observe the activity of the GPIOB pins in the peripheral window


```
} while ((osKernelSysTick() - tick) < delayPeriod);
```

Idle Demon

The final timer service provided by the RTOS isn't really a timer, but this is probably the best place to discuss it. If during our RTOS program we have no thread running and no thread ready to run (e.g. they are all waiting on delay functions) then the RTOS will use the spare runtime to call an "Idle Demon" that is again located in the RTX_Conf_CM.c file. This idle code is in effect a low priority thread within the RTOS which only runs when nothing else is ready.

```
void os_idle_demon (void)
{
    for (;;) {
        /* HERE: include here optional user code to be executed when no thread runs. */
    }
    /* end of os_idle_demon */
}
```

You can add any code to this thread, but it has to obey the same rules as user threads. The simplest use of the idle demon is to place the microcontroller into a low-power mode when it is not doing anything.

```
void os_idle_demon (void) {
    __wfe();
}
```

What happens next depends on the power mode selected in the microcontroller. At a minimum the CPU will halt until an interrupt is generated by the SysTick timer and execution of the scheduler will resume. If there is a thread ready to run then execution of the application code will resume. Otherwise, the idle demon will be reentered and the system will go back to sleep.

Exercise Idle Thread

In the Pack Installer select F ! 8! e f and copy it to your tutorial directory.

This is a copy of the virtual timer project. **Open the RTX_Conf_CM.c file and click the text editor tab**

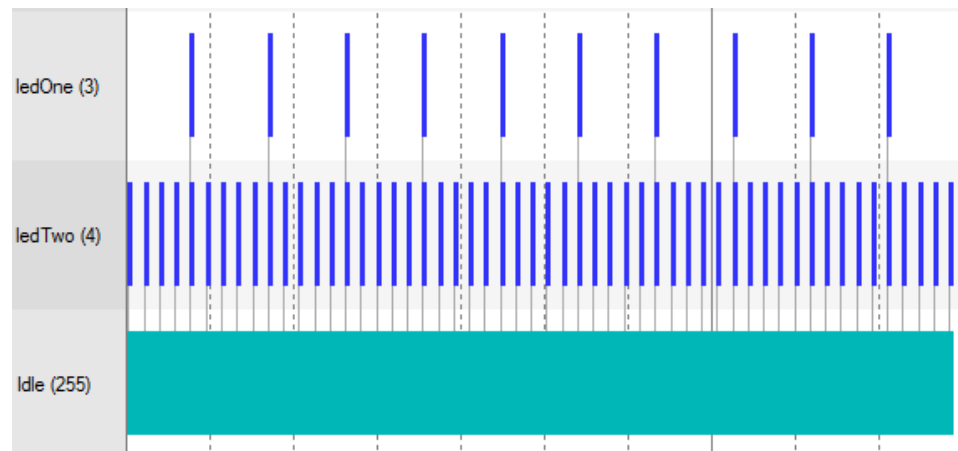
Locate the os_idle_demon thread

```
void os_idle_demon (void) {  
  
    int32_t i;  
  
    for (;;) {  
  
        //wfe();  
  
    }  
}
```

Build the code and start the debugger

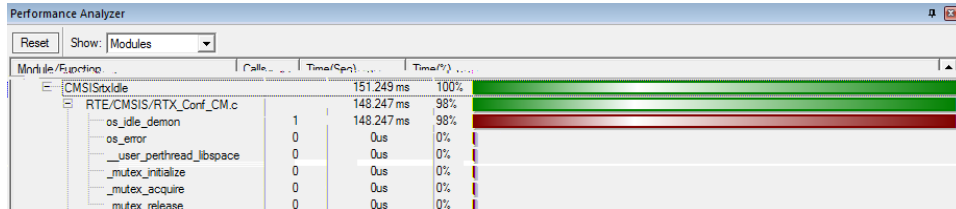
Run the code and observe the activity of the threads in the event Viewer.

This is a simple program which spend most of its time in the idle demon so this code will be run almost continuously



You can also see the activity of the idle demon in the event viewer. In a real project, the amount of time spent in the idle demon is an indication of spare CPU cycles.

Open the View Analysis Windows Performance Analyzer.



This window shows the cumulative run time for each function in the project. In this simple project the `os_idle_demon` is using most of the runtime because there is very little application code.

Exit the debugger

Remove the delay loop and the toggle instruction and add a `__wfe()` instruction in the for loop, so the code now looks like this.

```
void os_idle_demon (void) {

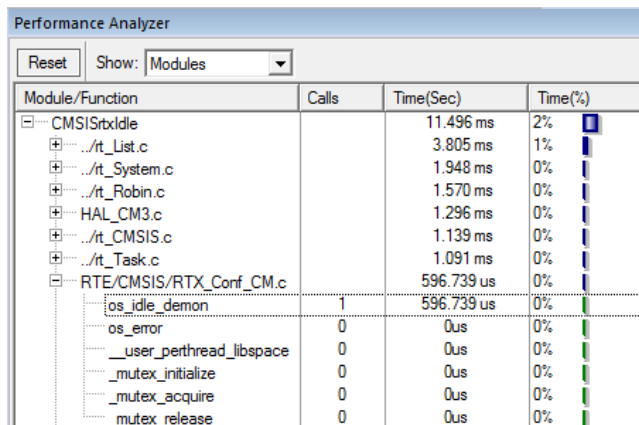
for (;;) {

__wfe();

}}
```

Rebuild the code, restart the debugger

Now when we enter the idle thread the `__wfe()` (wait for interrupt) instruction will halt the CPU until there is a peripheral or SysTick interrupt.



Performance analysis during hardware debugging

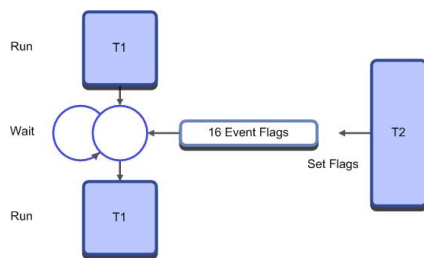
The code coverage and performance analysis tools are available when you are debugging on real hardware rather than simulation. However, to use these features you need two things: First, you need a microcontroller that has been fitted with the optional Embedded Trace Macrocell (ETM). Second, you need to use Keil ULINK*pro* debug adapter which supports instruction trace via the ETM.

Inter-Thread Communication

So far we have seen how application code can be defined as independent threads and how we can access the timing services provided by the RTOS. In a real application we need to be able to communicate between threads in order to make an application useful. To this end, a typical RTOS supports several different communication objects which can be used to link the threads together to form a meaningful program. The CMSIS-RTOS API supports inter-thread communication with signals, semaphores, mutexes, mailboxes and message queues. In the first section the key concept was concurrency. In this section the key concept is synchronizing the activity of multiple threads.

Signals

CMSIS-RTOS Keil RTX supports up to sixteen signal flags for each thread. These signals 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.



Each thread has 16 signal flags. 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 signal wait system calls will suspend execution of the thread and place it into the wait_evnt state. Execution of the thread will not start until all the flags set in the signal wait API call have been set. It is also possible to define a periodic timeout after which the waiting thread will move back to the ready state, so that it

can resume execution when selected by the scheduler. A value of 0xFFFF defines an infinite timeout period.

```
osEvent osSignalWait ( int32_t signals,uint32_t millisec);
```

If the signals variable is set to zero when osSignalWait is called then setting any flag will cause the thread to resume execution. You can see which flag was set by reading the osEvent.value.signals return value.

Any thread can set or clear a signal on any other thread.

```
int32_t osSignalSet ( osThreadId thread_id, int32_t signals);
```

```
int32_t osSignalClear ( osThreadId thread_id, int32_t signals);
```

Exercise Signals

In this exercise we will look at using signals to trigger activity between two threads. Whilst this is a simple program it introduces the concept of synchronizing the activity of threads together.

In the Pack Installer select F ! 9! Th b and copy it to your tutorial directory.

This is a modified version of the led flasher program one of the threads calls the same led function and uses osDelay() to pause the task. In addition it sets a signal flag to wake up the second led task.

```
void led_Thread2 (void const *argument) {
    for (;;) {
        LED_On(2);

        osSignalSet (T_led_ID1,0x01);

        osDelay(500);
```

```
LED_Off(2);

osSignalSet      (T_led_ID1,0x01);

osDelay(500);}}
```

The second led function waits for the signal flags to be set before calling the led functions.

```
void led_Thread1 (void const *argument) {

for (;;) {

osSignalWait (0x01,osWaitForever);

LED_On(1);

osSignalWait (0x01,osWaitForever);

LED_Off(1);

}}
```

Build the project and start the debugger

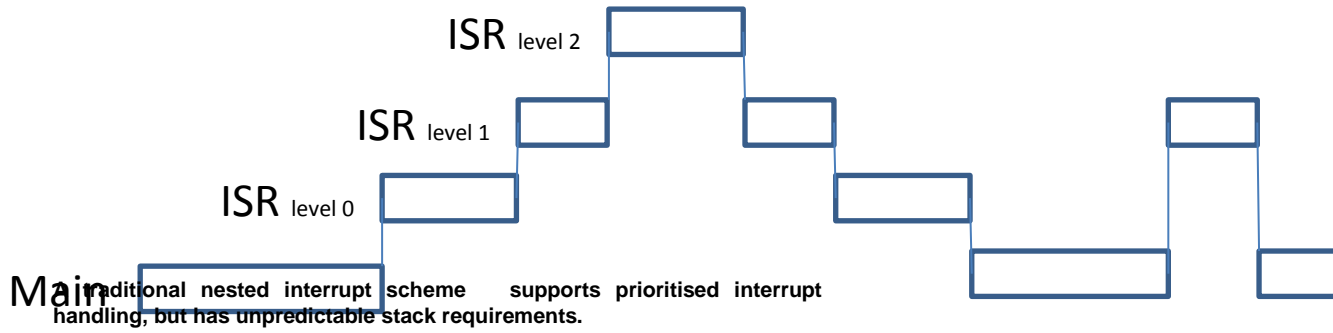
Open the GPIOB peripheral window and start the code running

Now the port pins will appear to be switching on and off together. Synchronizing the threads gives the illusion that both threads are running in parallel.

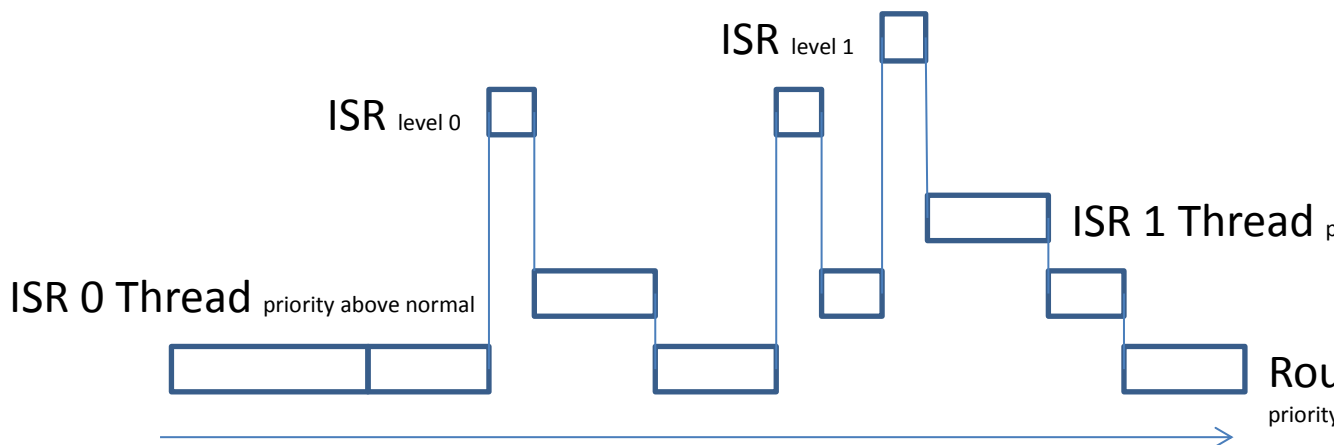
This is a simple exercise but it illustrates the key concept of synchronizing activity between threads in an RTOS based application.

RTOS Interrupt Handling

The use of signal flags is a simple and efficient method of triggering actions between threads running within the RTOS. Signal flags are also an important method of triggering RTOS threads from interrupt sources within the Cortex-M microcontroller. While it is possible to run C code in an interrupt service routine (ISR), this is not desirable within an RTOS if the interrupt code is going to run for more than a short period of time. This delays the timer tick and disrupts the RTOS kernel. The SysTick timer runs at the lowest priority within the NVIC so there is no overhead in reaching the interrupt routine.



With an RTOS application it is best to design the interrupt service code as a thread within the RTOS and assign it a high priority. The first line of code in the interrupt thread should make it wait for a signal flag. When an interrupt occurs, the ISR simply sets the signal flag and terminates. This schedules the interrupt thread which services the interrupt and then goes back to waiting for the next signal flag to be set.



A typical interrupt thread will have the following structure:

```
void Thread3 (void)
{
    while(1)
    {
        osSignalWait ( isrSignal,waitForever);           // Wait for the ISR to trigger an event
        .....                                           // Handle the interrupt
    }                                                    // Loop round and go back sleep
}
```

The actual interrupt source will contain a minimal amount of code.

```
void IRQ_Handler (void)
{
    osSignalSet (tsk3,isrSignal);                        // Signal Thread 3 with an event
}
```

Exercise Interrupt signal exercise

CMSIS-RTOS does not introduce any latency in serving interrupts generated by user peripherals. However operation of the RTOS may be disturbed if you lock out the SysTick interrupt for a long period of time. This exercise demonstrates a technique of signaling a thread from an interrupt and servicing the peripheral interrupt with a thread rather than a standard Interrupt service routine

In the Pack Installer select F !:! f !T h b and copy it to your tutorial directory.

In the main function we initialize the ADC and create an ADC thread which has a higher priority than all the other threads

```
osThreadDef(adc_Thread, osPriorityAboveNormal, 1, 0);

int main (void) {

    LED_Init ();

    init_ADC ();
```

```

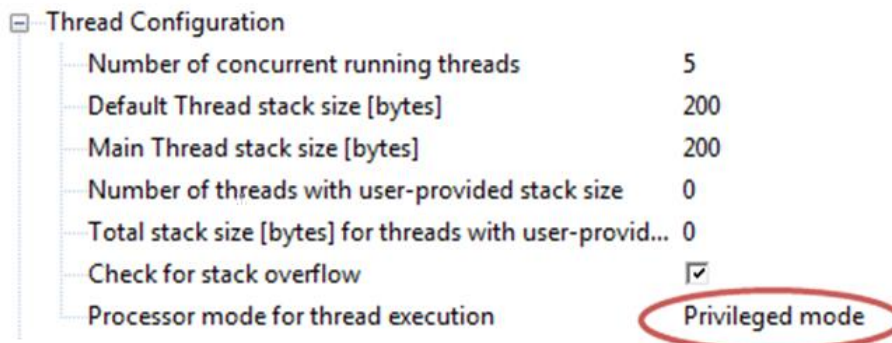
T_led_ID1 =      osThreadCreate(osThread(led_Thread1), NULL);

T_led_ID2 =      osThreadCreate(osThread(led_Thread2), NULL);

T_adc_ID =       osThreadCreate(osThread(adc_Thread), NULL);

```

However, there is a problem when we enter main: the RTOS may be configured to run the threads in unprivileged mode so we cannot access the NVIC registers without causing a fault exception. There are several ways round this. The simplest is to give the threads privileged access by changing the setting in the RTX_Conf_CM.c



Here, we have switched the thread execution mode to privileged which gives the threads full access to the Cortex-M processor. As we have added a thread, we also need to increase the number of concurrent running threads.

Build the code and start the debugger

Set breakpoints in led_Thread2, ADC_Thread and ADC1_2_IRQHandler

```

57 |      osDelay(500);
58 |      ADC1->CR2 |= (1UL << 22);
59 |      LED_Off(2);

```

And in adc_Thread()

```

35 |      osSignalWait ( 0x01,osWaitForever);
36 |      GPIOB->ODR = ADC1->DR;

```

And in ADC1_2_Handler


```

28 void ADC1_2_IRQHandler (void) {
29   osSignalSet ( T_adc_ID, 0x01);

```

Run the code

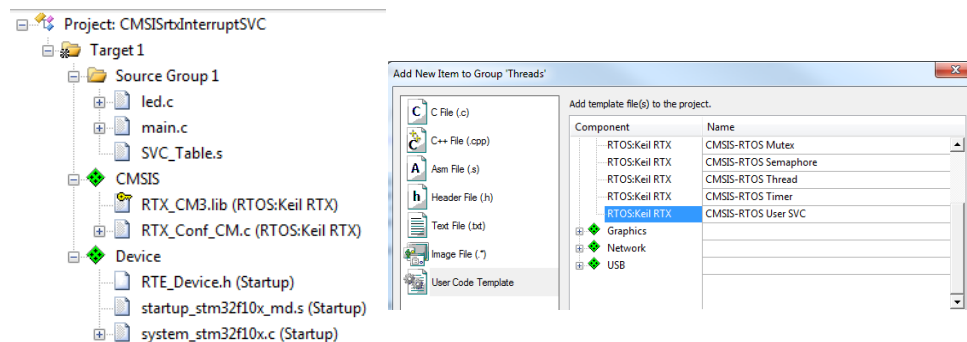
You should hit the first breakpoint which starts the ADC conversion, then run the code again and you should enter the ADC interrupt handler. The handler sets the `adc_thread` signal and quits. Setting the signal will cause the `adc` thread to preempt any other running task, run the ADC service code and then block waiting for the next signal.

Exercise Keil RTX and SVC exceptions

As we saw in the last example, when we are in a thread, it will be running in unprivileged mode. The simple solution is to allow threads to run in privileged mode but this allows the threads full access to the Cortex M processor potentially allowing runtime errors. In this exercise we will look at using the system call exception to enter privileged mode to run 'system level' code.

In the Pack b f ! f fd! F !21! f !Th b !b e!d ! ! ! ! tutorial directory.

In the project we have added a new file called `SVC_Table.s`. This file is available as a 'User Code Template' (CMSIS-RTOS User SVC) from the 'Add New Item' dialog:



This is the look up table for the SVC interrupts

; Import user SVC functions here.

```
IMPORT __SVC_1
```

```

EXPORT SVC_Table

SVC_Table

; Insert user SVC functions here. SVC 0 used by RTX Kernel.

DCD __SVC_1 ; user SVC function

```

In this file we need to add import name and table entry for each __SVC function we are going to use. In our example we only need __SVC_1

Now we can convert the ADC init function to a service call exception

```

void __svc(1) init_ADC (void);

void __SVC_1 (void){

```

Build the project and start the debugger

Step the code (F11) to the call to the init_ADC function and examine the operating mode in the register window.

Here we are in Thread mode, unprivileged and using the process stack

Internal	
Mode	Thread
Privilege	Unprivileged
Stack	PSP
States	4637
Sec	0.00008374

Now step into the function (F11) and step through the assembler until you reach the init_ADC C function

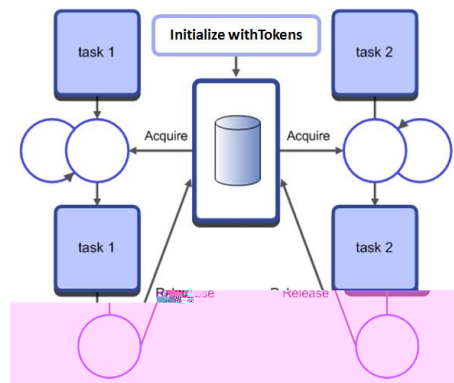
Internal	
Mode	Handler
Privilege	Privileged
Stack	MSP
States	4687
Sec	0.00008443

Now we are running in Handler mode with privileged access and are using the main stack pointer.

This allows us to setup the ADC and also access the NVIC.

Semaphores

Like signals, semaphores are a method of synchronizing activity between two or more threads. Put simply, a semaphore is a container that holds a number of tokens. As a thread executes, it will reach an RTOS call to acquire a semaphore token. If the semaphore contains one or more tokens, the thread will continue executing and the number of tokens in the semaphore will be decremented by one. If there are currently no tokens in the semaphore, the thread will be placed in a waiting state until a token becomes available. At any point in its execution, a thread may add a token to the semaphore causing its token count to increment by one.



Semaphores help to control access to program resources. Before a thread can access a resource, it must acquire a token. If none is available, it waits. When it is finished with the resource, it must return the token.

The diagram above illustrates the use of a semaphore to synchronize two threads. First, the semaphore must be created and initialized with an initial token count. In this case the semaphore is initialized with a single token. Both threads will run and reach a point in their code where they will attempt to acquire a token from the semaphore. The first thread to reach this point will acquire the token from the semaphore and continue execution. The second thread will also attempt to acquire a token, but as the semaphore is empty it will halt execution and be placed into a waiting state until a semaphore token is available.

Meanwhile, the executing thread can release a token back to the semaphore. When this happens, the waiting thread will acquire the token and leave the waiting state for the ready state. Once in the ready state the scheduler will place the thread into the run state so that thread execution can continue. While semaphores have a simple set of OS calls they can be one of the more difficult OS objects to fully understand. In this section we will first look at how to add

semaphores to an RTOS program and then go on to look at the most useful semaphore applications.

To use a semaphore in the CMSIS-RTOS you must first declare a semaphore container:

```
osSemaphoreId sem1;

osSemaphoreDef(sem1);
```

Then within a thread the semaphore container can be initialised with a number of tokens.

```
sem1 = osSemaphoreCreate(osSemaphore(sem1), SIX_TOKENS);
```

It is important to understand that semaphore tokens may also be created and destroyed as threads run. So for example you can initialise a semaphore with zero tokens and then use one thread to create tokens into the semaphore while another thread removes them. This allows you to design threads as producer and consumer threads.

Once the semaphore is initialized, tokens may be acquired and sent to the semaphore in a similar fashion to event flags. The `os_sem_wait` call is used to block a thread until a semaphore token is available, like the `os_evnt_wait` call. A timeout period may also be specified with `0xFFFF` being an infinite wait.

```
osStatus osSemaphoreWait(osSemaphoreId semaphore_id, uint32_t millisec);
```

Once the thread has finished using the semaphore resource, it can send a token to the semaphore container.

```
osStatus osSemaphoreRelease(osSemaphoreId semaphore_id);
```

Exercise Semaphore Signalling

In this exercise we will look at the configuration of a semaphore and use it to signal between two tasks.

In the Pack Installer select `F !22!` `f` `!Th b` and copy it to your tutorial directory.

First, the code creates a semaphore called `sem1` and initialises it with zero tokens.

```
osSemaphoreId sem1;  
osSemaphoreDef(sem1);  
int main (void) {  
    sem1 = osSemaphoreCreate(osSemaphore(sem1), 0);
```

The first task waits for a token to be sent to the semaphore.

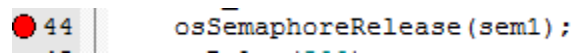
```
void led_Thread1 (void const *argument) {  
    for (;;) {  
        osSemaphoreWait(sem1, osWaitForever);  
        LED_On(1);  
        osDelay(500);  
        LED_Off(1);  
    }  
}
```

While the second task periodically sends a token to the semaphore.

```
void led_Thread2 (void const *argument) {  
    for (;;) {  
        LED_On(2);  
        osSemaphoreRelease(sem1);  
        osDelay(500);  
        LED_Off(2);  
        osDelay(500);  
    }  
}
```

Build the project and start the debugger

Set a breakpoint in the `led_Thread2` task



```
44 | osSemaphoreRelease(sem1);
```

Run the code and observe the state of the threads when the breakpoint is reached.

ID	Name	Priority	State
255	os_idle_demon	0	Ready
3	led_Thread1	5	Wait_SEM
2	led_Thread2	4	Running
1	main	4	Ready

Now `led_thread1` is blocked waiting to acquire a token from the semaphore. `led_Thread1` has been created with a higher priority than `led_thread2` so as soon as a token is placed in the semaphore it will move to the ready state and pre-empt the lower priority task and start running. When it reaches the `osSemaphoreWait()` call it will again block.

Now block step the code (F10) and observe the action of the threads and the semaphore.

Using Semaphores

Although semaphores have a simple set of OS calls, they have a wide range of synchronizing applications. This makes them perhaps the most challenging RTOS object to understand. In this section we will look at the most common uses of semaphores. These are taken from “The Little Book Of Semaphores” by Allen B. Downey. This book may be freely downloaded from the URL given in the bibliography at the end of this book.

Signaling

Synchronizing the execution of two threads is the simplest use of a semaphore:

```
osSemaphoreId sem1;

osSemaphoreDef(sem1);

void thread1 (void)
{
    sem1 = osSemaphoreCreate(osSemaphore(sem1), 0);

    while(1)
    {
        FuncA();
        osSemaphoreRelease(sem1)
    }
}

void task2 (void)
{
    while(1)
    {
        osSemaphoreWait(sem1,osWaitForever)
        FuncB();
    }
}
```

```
}  
}
```

In this case the semaphore is used to ensure that the code in FuncA() is executed before the code in FuncB().

Multiplex

A multiplex is used to limit the number of threads that can access a critical section of code. For example, this could be a routine that accesses memory resources and can only support a limited number of calls.

```
osSemaphoreId multiplex;  
  
osSemaphoreDef(multiplex);  
  
void thread1 (void)  
{  
    multiplex =osSemaphoreCreate(osSemaphore(multiplex), FIVE_TOKENS);  
  
    while(1) {  
        osSemaphoreWait(multiplex,osWaitForever)  
        ProcessBuffer();  
        osSemaphoreRelease(multiplex);  
    }  
}
```

In this example we initialise the multiplex semaphore with 5 tokens. Before a thread can call the ProcessBuffer() function, it must acquire a semaphore token. Once the function has completed, the token is sent back to the semaphore. If more than five threads are attempting to call ProcessBuffer(), the sixth must wait until a thread has finished with ProcessBuffer() and returns its token. Thus the multiplex semaphore ensures that a maximum of five threads can call the ProcessBuffer() function “simultaneously”.

Exercise Multiplex

In this exercise we will look at using a semaphore to control access to a function by creating a multiplex.

In the Pack Installer select F !23!N f and copy it to your tutorial directory.

The project creates a semaphore called semMultiplex which contains one token. Next, six instances of a thread containing a semaphore multiplex are created.

Build the code and start the debugger

Open the Peripherals General Purpose IO GPIOB window

Run the code and observe how the tasks set the port pins

As the code runs only one thread at a time can access the LED functions so only one port pin is set.

Exit the debugger and increase the number of tokens allocated to the semaphore when it is created

```
semMultiplex = osSemaphoreCreate(osSemaphore(semMultiplex), 3);
```

Build the code and start the debugger

Run the code and observe the GPIOB pins

Now three threads can access the led functions ‘concurrently’.

Rendezvous

A more generalised form of semaphore signalling is a rendezvous. A rendezvous ensures that two threads reach a certain point of execution. Neither may continue until both have reached the rendezvous point.

```
osSemaphoreId arrived1,arrived2;
```

```
osSemaphoreDef(arrived1);
```



```
osSemaphoreDef(arrived2);

void thread1 (void){
Arrived1 =osSemaphoreCreate(osSemaphore(arrived1),ZERO_TOKENS);
Arrived2 =osSemaphoreCreate(osSemaphore(arrived2),ZERO_TOKENS);
while(1){
FuncA1();
osSemaphoreRelease(Arrived1);
osSemaphoreWait(Arrived2,osWaitForever);
FuncA2();
}}

void thread2 (void) {
while(1){
FuncB1();
Os_sem_send(Arrived2);
os_sem_wait(Arrived1,osWaitForever);
FuncB2();
}}
```

In the above case the two semaphores will ensure that both threads will rendezvous and then proceed to execute FuncA2() and FuncB2().

Exercise Rendezvous

In this project we will create two tasks and make sure that they have reached a semaphore rendezvous before running the LED functions.

In the Pack Installer select F !24!S f e f and copy it to your tutorial directory.

Build the project and start the debugger.

Open the Peripherals\General Purpose IO\GPIOB window.

Run the code

Initially the semaphore code in each of the LED tasks is commented out. Since the threads are not synchronised the GPIO pins will toggle randomly.

Exit the debugger

Un-comment the semaphore code in the LED tasks.

Build the project and start the debugger.

Run the code and observe the activity of the pins in the GPIOB window.

Now the tasks are synchronised by the semaphore and run the LED functions ‘concurrently’.

Barrier Turnstile

Although a rendezvous is very useful for synchronising the execution of code, it only works for two functions. A barrier is a more generalised form of rendezvous which works to synchronise multiple threads.

```
osSemaphoreId count,barrier;

osSemaphoreDef(counter);

osSemaphoreDef(barrier);

unsigned int count;

void thread1 (void)
{
    count = osSemaphoreCreate(osSemaphore(counter),ONE_TOKEN);
    barrier = osSemaphoreCreate(osSemaphore(barrier),ZERO_TOKENS);

    while(1)
    {
        //Allow only one task at a time to run this code
        osSemaphoreWait(counter);
        count = count+1;
        if count == 5 os_sem_send(barrier, osWaitForever);
        osSemaphoreRelease(counter);

        //when all five tasks have arrived the barrier is opened
        os_sem_wait(barrier, osWaitForever);
        os_sem_send(barrier);

        critical_Function();
    }
}
```

In this code we use a global variable to count the number of threads which have arrived at the barrier. As each function arrives at the barrier it will wait until it can acquire a token from the counter semaphore. Once acquired, the count variable will be incremented by one. Once we have incremented the count

variable, a token is sent to the counter semaphore so that other waiting threads can proceed. Next, the barrier code reads the count variable. If this is equal to the number of threads which are waiting to arrive at the barrier, we send a token to the barrier semaphore.

In the example above we are synchronising five threads. The first four threads will increment the count variable and then wait at the barrier semaphore. The fifth and last thread to arrive will increment the count variable and send a token to the barrier semaphore. This will allow it to immediately acquire a barrier semaphore token and continue execution. After passing through the barrier it immediately sends another token to the barrier semaphore. This allows one of the other waiting threads to resume execution. This thread places another token in the barrier semaphore which triggers another waiting thread and so on. This final section of the barrier code is called a turnstile because it allows one thread at a time to pass the barrier. In our model of concurrent execution this means that each thread waits at the barrier until the last arrives then the all resume simultaneously. In the following exercise we create five instances of one thread containing barrier code. However the barrier could be used to synchronise five unique threads.

Exercise Semaphore Barrier

In this exercise we will use semaphores to create a barrier to synchronise multiple tasks.

In the Pack Installer select "Ex 14 Barrier" and copy it to your tutorial directory.

Build the project and start the debugger.

Open the Peripherals\General Purpose IO\GPIOB window.k

Run the code.

Initially, the semaphore code in each of the threads is commented out. Since the threads are not synchronised the GPIO pins will toggle randomly like in the rendezvous example.

Exit the debugger.

Remove the comments on lines 34, 45, 53 and 64 to enable the barrier code.

Built the project and start the debugger.

Run the code and observe the activity of the pins in the GPIOB window.

Now the tasks are synchronised by the semaphore and run the LED functions ‘concurrently’.

Semaphore Caveats

Semaphores are an extremely useful feature of any RTOS. However semaphores can be misused. You must always remember that the number of tokens in a semaphore is not fixed. During the runtime of a program semaphore tokens may be created and destroyed. Sometimes this is useful, but if your code depends on having a fixed number of tokens available to a semaphore you must be very careful to always return tokens back to it. You should also rule out the possibility of accidentally creating additional new tokens.

Mutex

Mutex stands for “Mutual Exclusion”. In reality, a mutex is a specialized version of semaphore. Like a semaphore, a mutex is a container for tokens. The difference is that a mutex can only contain one token which cannot be created or destroyed. The principle use of a mutex is to control access to a chip resource such as a peripheral. For this reason a mutex token is binary and bounded. Apart from this it really works in the same way as a semaphore. First of all we must declare the mutex container and initialize the mutex:

```
osMutexId uart_mutex;
```

```
osMutexDef (uart_mutex);
```

Once declared the mutex must be created in a thread.

```
uart_mutex = osMutexCreate(osMutex(uart_mutex));
```

Then any thread needing to access the peripheral must first acquire the mutex token:

```
osMutexWait(osMutexId mutex_id,uint32_t millisec;
```

Finally, when we are finished with the peripheral the mutex must be released:

```
osMutexRelease(osMutexId mutex_id);
```

Mutex use is much more rigid than semaphore use, but is a much safer mechanism when controlling absolute access to underlying chip registers.

Exercise Mutex

In this exercise our program writes streams of characters to the microcontroller UART from different threads. We will declare and use a mutex to guarantee that each thread has exclusive access to the UART until it has finished writing its block of characters.

In the Pack Installer select "Ex 15 Mutex" and copy it to your tutorial directory.

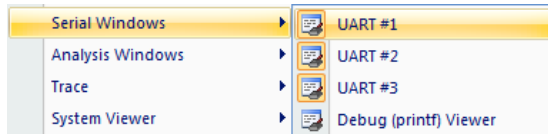
This project declares two threads which both write blocks of characters to the UART. Initially, the mutex is commented out.

```
void uart_Thread1 (void const *argument) {  
  
    uint32_t i;  
  
    for (;;) {  
  
        //osMutexWait(uart_mutex, osWaitForever);  
  
        for( i=0;i<10;i++)  SendChar('1');  
  
        SendChar('\n');  
  
        SendChar('\r');  
  
        //osMutexRelease(uart_mutex);  
  
    }  
}
```

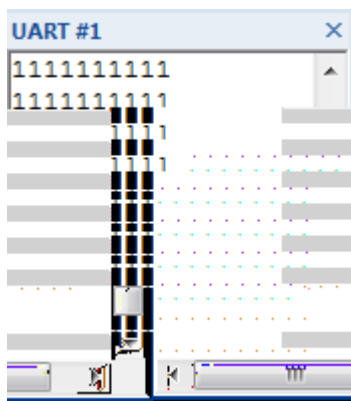
In each thread the code prints out the thread number. At the end of each block of characters it then prints the carriage return and new line characters.

Build the code and start the debugger.

Open the UART1 console window with View\Serial Windows\UART #1



Start the code running and observe the output in the console window.



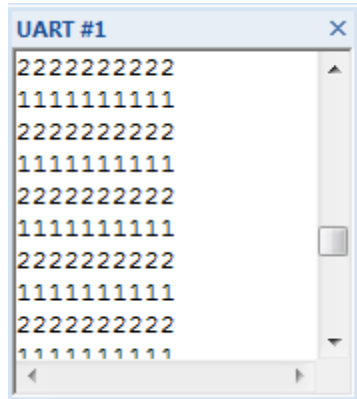
Here we can see that the output data stream is corrupted by each thread writing to the UART without any accessing control.

Exit the debugger.

Uncomment the mutex calls in each thread.

Build the code and start the debugger.

Observe the output of each task in the console window.



Now the mutex guarantees each task exclusive access to the UART while it writes each block of characters.

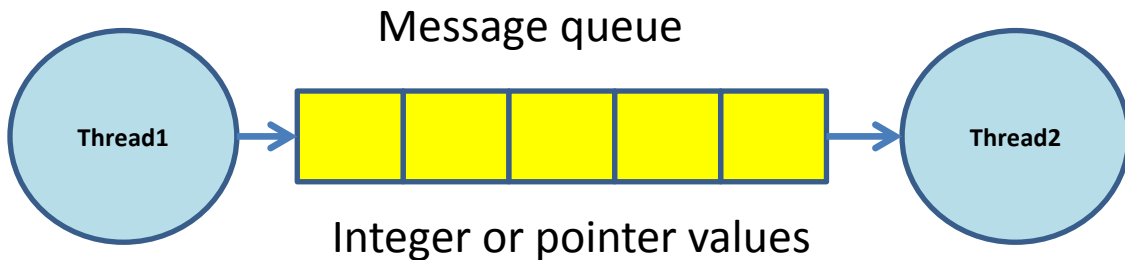
Mutex Caveats

Clearly you must take care to return the mutex token when you are finished with the chip resource, or you will have effectively prevented any other thread from accessing it. You must also be extremely careful about using the `osThreadTerminate()` call on functions which control a mutex token. Keil RTX is designed to be a small footprint RTOS so that it can run on even the very small Cortex-M microcontrollers. Consequently there is no thread deletion safety. This means that if you delete a thread which is controlling a mutex token, you will destroy the mutex token and prevent any further access to the guarded peripheral.

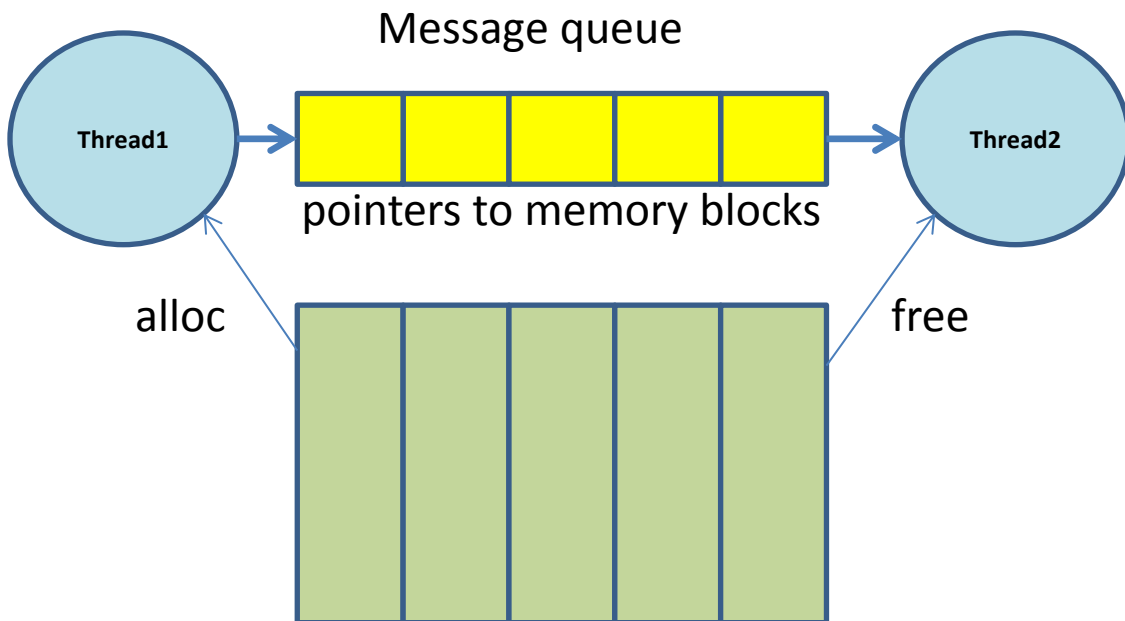
Data Exchange

So far all of the inter-thread communication methods have only been used to trigger execution of threads; they do not support the exchange of program data between threads. Clearly, in a real program we will need to move data between threads. This could be done by reading and writing to globally declared variables. In anything but a very simple program, trying to guarantee data integrity would be extremely difficult and prone to unforeseen errors. The exchange of data between threads needs a more formal asynchronous method of communication.

CMSIS-RTOS provides two methods of data transfer between threads. The first method is a message queue which creates a buffered data ‘pipe’ between two threads. The message queue is designed to transfer integer values.

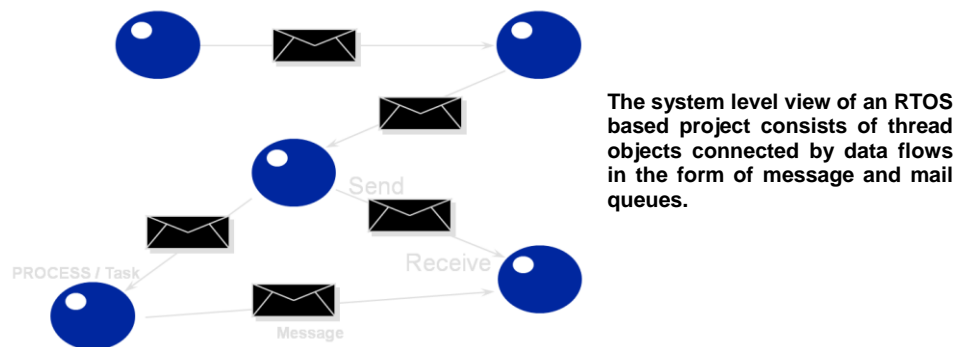


The second form of data transfer is a mail queue. This is very similar to a message queue except that it transfers blocks of data rather than a single integer.



Message and mail queues can provide a method for transferring data between threads. This allows you to view your design as a collection of objects (threads) interconnected by data flows. The data flow is implemented by message and mail queues. This provides both a buffered transfer of data and a well defined communication interface between threads. Starting with a system level design based on threads connected by mail and message queues allows you to code different subsystems of your project, especially useful if you are working in a

team. Also as each thread has well defined inputs and outputs it is easy to isolate for testing and code reuse.



Message Queue

To setup a message queue we first need to allocate the memory resources.

```
osMessageQId Q_LED;
```

```
osMessageQDef (Q_LED,16_Message_Slots,unsigned int);
```

This defines a message queue with sixteen storage elements. In this particular queue each element is defined as an unsigned int. Whilst we can post data directly into the message queue, it is also possible to post a pointer to a data object.

```
osEvent result;
```

We also need to define an osEvent variable which will be used to retrieve the queue data. The osEvent variable is a union that allows you to retrieve data from the message queue in a number of formats.

```
union{
```

```
uint32_t v
```

```
void *p
```

```
int32_t signals
```

```
}value
```

The osEvent union allows you to read the data posted to the message queue as an unsigned int or a void pointer. Once the memory resources are created we can declare the message queue in a thread.

```
Q_LED = osMessageCreate(osMessageQ(Q_LED),NULL);
```

Once the message queue has been created we can put data into the queue from one thread.

```
osMessagePut(Q_LED,0x0,osWaitForever);
```

and then read it from the queue in another.

```
result = osMessageGet(Q_LED,osWaitForever);
```

```
LED_data = result.value.v;
```

Exercise message queue

In this exercise we will look at defining a message queue between two threads and then use it to send process data.

! f! Qbd ! b f ! f fd! F !27 N f bhf! R f f !b e!d ! ! ! !
tutorial directory.

Open Main.c and view the message queue initialization code.

```
osMessageQId Q_LED;
```

```
osMessageQDef (Q_LED,0x16,unsigned char);
```

```
osEvent result;
```

```
int main (void) {
```

```
LED_Init ();
```

```
Q_LED = osMessageCreate(osMessageQ(Q_LED),NULL);
```

We define and create the message queue in the main thread along with the event structure.

```
osMessagePut(Q_LED,0x1,osWaitForever);
```

```
osDelay(100);
```

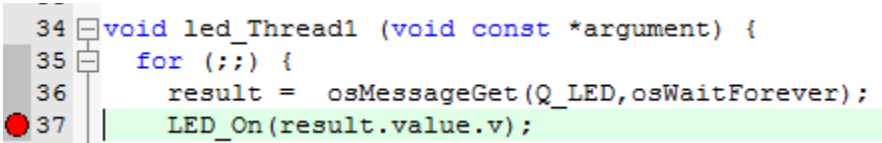
Then in one of the threads we can post data and receive it in the second.

```
result = osMessageGet(Q_LED,osWaitForever);
```

```
LED_On(result.value.v);
```

Build the project and start the debugger.

Set a breakpoint in led_thread1.



```
34 void led_Thread1 (void const *argument) {
35     for (;;) {
36         result = osMessageGet(Q_LED,osWaitForever);
37         LED_On(result.value.v);
```

Now run the code and observe the data as it arrives.

Memory pool

While it is possible to post simple data values into the message queue it is also possible to post a pointer to a more complex object. CMSIS-RTOS supports the dynamic allocation of memory in the form of a memory pool. Here we can declare a structure which combines a number of data elements.

```
typedef struct {
    uint8_t LED0;

    uint8_t LED1;

    uint8_t LED2;

    uint8_t LED3;

} memory_block_t;
```

Then we can create a pool of these objects as blocks of memory.

```
osPoolDef(led_pool,ten_blocks,memory_block_t);
```

```
osPoolId( led_pool);
```

Then we can create the memory pool by declaring it in a thread.

```
led_pool = osPoolCreate(osPool(led_pool));
```

Now we can allocate a memory pool within a thread.

```
memory_block_t *led_data;
```

```
*led_data = (memory_block_t *) osPoolAlloc(led_pool);
```

and then populate it with data;

```
led_data->LED0 = 0;
```

```
led_data->LED1 = 1;
```

```
led_data->LED2 = 2;
```

```
led_data->LED3 = 3;
```

It is then possible to place the pointer to the memory block in a message queue.

```
osMessagePut(Q_LED,(uint32_t)led_data,osWaitForever);
```

so the data can be accessed by another task.

```
osEvent event; memory_block_t * received;
```

```
event = osMessageGet(Q_LED,osWaitForever);
```

```
*received = (memory_block_t *)event.value.p;
```

```
led_on(received->LED0);
```

Once the data in the memory block has been used the block must be released back to the memory pool for reuse.

```
osPoolFree(led_pool,received);
```

Exercise Memorypool

This exercise demonstrates the configuration of a memorypool and message queue to transfer complex data between threads.

In the Pack Installer select `File -> Open` and copy it to your tutorial directory.

This exercise creates a memory pool and a message queue. A producer thread acquires a buffer from the memory pool and fills it with data. A pointer to the memory pool buffer is then placed in the message queue. A second thread reads the pointer from the message queue and then accesses the data stored in the memory pool buffer before freeing the buffer back to the memory pool. This allows large amounts of data to be moved from one thread to another in a safe synchronized way. This is called a 'zero copy' memory queue as only the pointer is moved through the message queue, the actual data does not move memory locations.

At the beginning of main.c the memory pool and message queue are defined.

```
typedef struct {  
  
    uint8_t canData[8];  
  
} message_t;  
  
osPoolDef(mpool, 16, message_t);  
  
osPoolId mpool;  
  
osMessageQDef(queue, 16, message_t);  
  
osMessageQId queue;
```

In the producer thread acquire a message buffer, fill it with data and post a testData++;

```
message = (message_t*)osPoolAlloc(mpool);  
  
for(index =0;index<8;index++){  
  
    message->canData[index] = testData+index;}  
  
osMessagePut(queue, (uint32_t)message, osWaitForever);
```

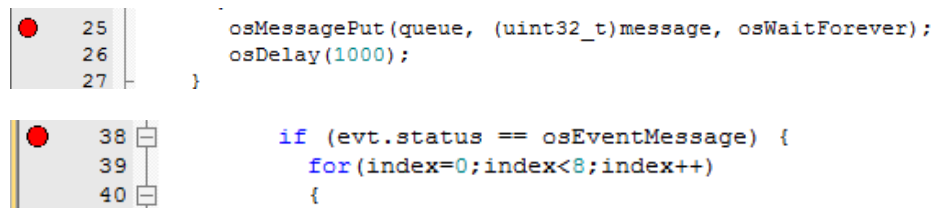
Then in the consumer thread we can read the message queue using the event.value.p pointer object and then access the memory pool buffer. Once we have used the data in the buffer it can be released back to the memory pool.

```
for(index=0;index<8;index++){  
  
    message_t *message = (message_t*)evt.value.p;  
  
    LED_On((uint32_t)message->canData[index]);}
```

```
osPoolFree(mpool, message);
```

Build the code and start the debugger.

Place breakpoints on the osMessagePut and osmessageGet functions.



Run the code and observe the data being transferred between the threads.

Mail Queue

While memory pools can be used as data buffers within a thread, CMSIS-RTOS also implements a mail queue which is a combination of memory pool and message queue. The Mail queue uses a memory pool to create formatted memory blocks and passes pointers to these blocks in a message queue. This allows the data to stay in an allocated memory block while we only move a pointer between the different threads. A simple mail queue API makes this easy to setup and use. First we need to declare a structure for the mail slot similar to the one we used for the memory pool.

```
typedef struct {
    uint8_t LED0;
    uint8_t LED1;
    uint8_t LED2;
    uint8_t LED3;
} mail_format;
```

This message structure is the format of the memory block that is allocated in the mail queue. Now we can create the mail queue and define the number of memory block ‘slots’ in the mail queue.

```
osMailQDef(mail_box, sixteen_mail_slots, mail_format);  
  
osMailQId mail_box;
```

Once the memory requirements have been allocated we can create the mail queue in a thread.

```
mail_box = osMailCreate(osMailQ(mail_box), NULL);
```

Once the mail queue has been instantiated we can post a message. This is different from the message queue in that we must first allocate a mail slot and populate it with data.

```
mail_format *LEDtx;  
  
LEDtx = (mail_format*)osMailAlloc(mail_box, osWaitForever);
```

First declare a pointer in the mail slot format and then allocate this to a mail slot. This locks the mail slot and prevents it being allocated to any other thread. If all of the mail slots are in use the thread will block and wait for a mail slot to become free. You can define a timeout in milliseconds which will allow the task to continue if a mail slot has not become free.

Once a mail slot has been allocated it can be populated with data and then posted to the mail queue.

```
LEDtx->LED0 = led0[index];  
  
LEDtx->LED1 = led1[index];  
  
LEDtx->LED2 = led2[index];  
  
LEDtx->LED3 = led3[index];  
  
osMailPut(mail_box, LEDtx);
```

The receiving thread must declare a pointer in the mail slot format and an osEvent structure.

```
osEvent evt;  
  
mail_format *LEDrx;
```

Then in the thread loop we can wait for a mail message to arrive.

```
evt = osMailGet(mail_box, osWaitForever);
```

We can then check the event structure to see if it is indeed a mail message and extract the data.

```
if (evt.status == osEventMail) {
    LEDrx = (mail_format*)evt.value.p;
```

Once the data in the mail message has been used the mail slot must be released so it can be reused.

```
osMailFree(mail_box, LEDrx);
```

Exercise Mailbox

This exercise demonstrates configuration a mailbox and using it to post messages between tasks.

In the Pack Installer select F ! 28! N b c and copy it to your tutorial directory.

The project creates a 16 slot mailbox to send LED data between threads.

```
typedef struct {
    uint8_t LED0;
    uint8_t LED1;
    uint8_t LED2;
    uint8_t LED3;
} mail_format;

osMailQDef(mail_box, 16, mail_format);
osMailQId mail_box;

int main (void) {
    LED_Init();
    mail_box = osMailCreate(osMailQ(mail_box), NULL);
```

A producer task then allocates a mail slot fills it with data and posts it to the mail queue.

```
LEDtx = (mail_format*)osMailAlloc(mail_box, osWaitForever);

LEDtx->LED0 = led0[index];
```



```
LEDtx->LED1 = led1[index];

LEDtx->LED2 = led2[index];

LEDtx->LED3 = led3[index];

osMailPut(mail_box, LEDtx);
```

The receiving task waits for a mail message to arrive then reads the data. Once the data has been used the mail slot is released.

```
evt = osMailGet(mail_box, osWaitForever);

if(evt.status == osEventMail){

    LEDrx = (mail_format*)evt.value.p;

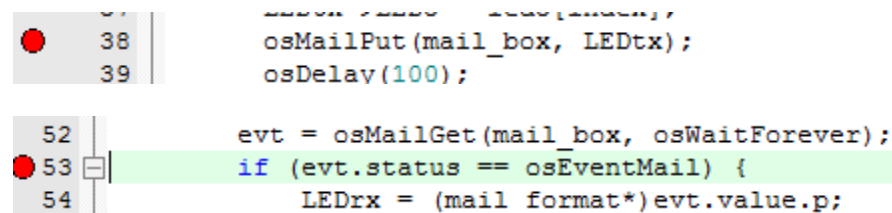
    LED_Out((LEDrx->LED0|LEDrx->LED1|LEDrx->LED2|LEDrx->LED3)<<8);

    osMailFree(mail_box, LEDrx);

}
```

Build the code and start the debugger

Set a breakpoint in the consumer and producer threads and run the code.



```
38  osMailPut(mail_box, LEDtx);
39  osDelay(100);

52  evt = osMailGet(mail_box, osWaitForever);
53  if (evt.status == osEventMail) {
54  LEDrx = (mail_format*)evt.value.p;
```

Observe the mailbox messages arriving at the consumer thread.

Configuration

So far we have looked at the CMSIS-RTOS API. This includes thread management functions, time management and inter-thread communication. Now that we have a clear idea of exactly what the RTOS kernel is capable of, we can take a more detailed look at the configuration file. There is one configuration file for all of the Cortex-M based microcontrollers.

[-] Thread Configuration	
Number of concurrent running user threads	6
Default Thread stack size [bytes]	200
Main Thread stack size [bytes]	200
Number of threads with user-provided stack ...	0
Total stack size [bytes] for threads with user-...	0
Stack overflow checking	<input checked="" type="checkbox"/>
Stack usage watermark	<input checked="" type="checkbox"/>
Processor mode for thread execution	Privileged mode
[-] RTX Kernel Timer Tick Configuration	
Use Cortex-M SysTick timer as RTX Kernel Ti...	<input checked="" type="checkbox"/>
RTOS Kernel Timer input clock frequency [Hz]	12000000
RTX Timer tick interval value [us]	1000
[-] System Configuration	
+ Round-Robin Thread switching	<input checked="" type="checkbox"/>
+ User Timers	<input checked="" type="checkbox"/>
ISR FIFO Queue size	16 entries

Like the other configuration files, the RTX_Conf_CM.c file is a template file which presents all the necessary configurations as a set of menu options.

Thread Definition

In the thread definition section we define the basic resources which will be required by the CMSIS-RTOS threads. For each thread we allocate a defined stack space (in the above example this is 200 bytes.) We also define the maximum number of concurrently running threads. Thus, the amount of RAM required for the above example can easily be computed as 200×6 or 1200 bytes. If some of our threads need a larger stack space, then a larger stack can be allocated when the task is created. In addition the total custom stack size must be allocated in the configuration file along with the total number of threads with custom stack size. Again, the RAM requirement is easily calculated.

Kernel Debug support

During development, CMSIS-RTOS can trap stack overflows. When this option is enabled, an overflow of a thread stack space will cause the RTOS kernel to call the `os_error` function which is located in the `RTX_Conf_CM.c` file. This function gets an error code and then sits in an infinite loop. The stack checking option is intended for use during debugging and should be disabled on the final application to minimize the kernel overhead. However, it is possible to modify the `os_error()` function if enhanced error protection is required in the final release.

```
#define OS_ERROR_STACK_OVF    1

#define OS_ERROR_FIFO_OVF     2

#define OS_ERROR_MBX_OVF     3

extern osThreadId svcThreadGetId (void);

void os_error (uint32_t error_code) {

    switch (error_code) {

        case OS_ERROR_STACK_OVF:

            /* Stack overflow detected for the currently running task. */

            /* Thread can be identified by calling svcThreadGetId(). */

            break;

        case OS_ERROR_FIFO_OVF:

            /* ISR FIFO Queue buffer overflow detected. */

            break;

        case OS_ERROR_MBX_OVF:

            /* Mailbox overflow detected. */

            break;

    }

    for (;;)

}
```

It is also possible to monitor the maximum stack memory usage during run time. If you check the ‘Stack Usage Watermark’ option, a pattern (0xCC) is written into each stack space. During runtime this watermark is used to calculate the maximum memory usage. This figure is reported in the threads section of the ‘System and Event Viewer’ window.

ID	Name	Priority	State	Delay	Event Value	Event Mask	Stack Usage
1	osTimerThread	High	Wait_MBX				cur: 32%, max: 32% [64/200]
3	Thread	Normal	Running				cur: 32%, max: 32% [64/200]
255	os_idle_demon	None	Ready				

The final option in the thread definition section allows you to define the number of user timers. It is a common mistake to leave this set at zero. If you do not set this value to match the number of virtual timers in use by your application, the `os_timer()` API calls will fail to work. The thread definition section also allows us to select whether the threads are running in privileged or unprivileged mode.

System Timer Configuration

The default timer for use with CMSIS-RTOS is the Cortex-M SysTick timer which is present on nearly all Cortex-M processors. The input to the SysTick timer will generally be the CPU clock. It is possible to use a different timer by unchecking the ‘Use SysTick’ option. If you do this there are two function stubs in the `RTX_Conf_CM.c` file that allow you to initialize the alternative timer and acknowledge its interrupt.

```
int os_tick_init (void) {

    return (-1); /* Return IRQ number of timer (0..239) */

}

void os_tick_irqack (void) {

    /* ... */

}
```

Whichever timer you use you must next setup its input clock value. Next we must define our timer tick rate. This is the rate at which timer interrupts are generated. On each timer tick the RTOS kernel will run the scheduler to determine if it is necessary to perform a context switch and replace the running

thread. The timer tick value will depend on your application, but the default starting value is set to 1msec.

Timeslice configuration

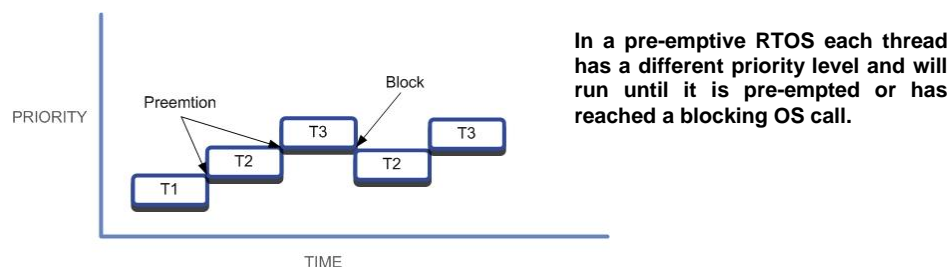
The final configuration setting allows you to enable round robin scheduling and define the timeslice period. This is a multiple of the timer tick rate so in the above example each thread will run for five ticks or 5msec before it will pass execution to another thread of the same priority that is ready to run. If no thread of the same priority is ready to run, it will continue execution. The system configuration options also allow you to enable and configure the virtual timer thread. If you are going to use the virtual timers this option must be configured or the timers will not work. Then lastly if you are going to trigger a thread from an interrupt routine using event flags then it is possible to define a FIFO queue for triggered signals. This buffers signal triggers in the event of bursts of interrupt activity.

Scheduling Options

CMSIS-RTOS allows you to build an application with three different kernel scheduling options. These are round robin scheduling, pre-emptive scheduling and co-operative multi-tasking. A summary of these options are as follows:

Pre-emptive scheduling

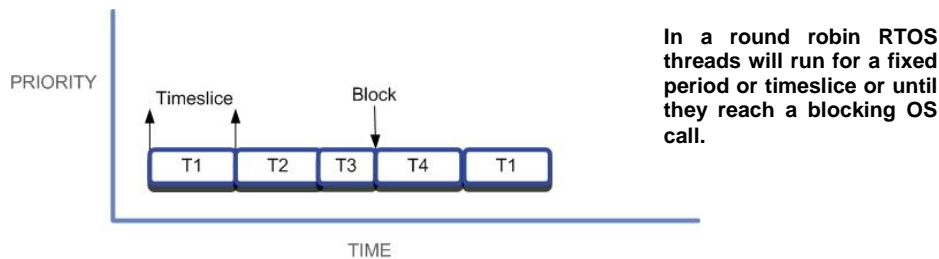
If the round robin option is disabled in the RTX_Config_CM.c file, each thread must be declared with a different priority. When the RTOS is started and the threads are created, the thread with the highest priority will run.



This thread will run until it blocks, i.e. it is forced to wait for an event flag, semaphore or other object. When it blocks, the next ready thread with the highest priority will be scheduled and will run until it blocks, or a higher priority thread becomes ready to run. So with pre-emptive scheduling we build a hierarchy of thread execution, with each thread consuming variable amounts of run time.

Round-Robin Scheduling

A round-robin based scheduling scheme can be created by enabling the round-robin option in the RTX_Conf_CM.c file and declaring each thread with the same priority.



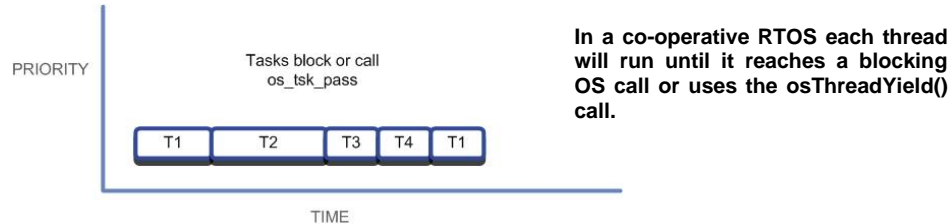
In this scheme, each thread will be allotted a fixed amount of run time before execution is passed to the next ready thread. If a thread blocks before its timeslice has expired, execution will be passed to the next ready thread.

Round-Robin Pre-emptive Scheduling

As discussed at the beginning of this tutorial, the default scheduling option for the Keil RTX is round-robin pre-emptive. For most applications this is the most useful option and you should use this scheduling scheme unless there is a strong reason to do otherwise.

Co-operative Multitasking

A final scheduling option is co-operative multitasking. In this scheme, round-robin scheduling is disabled and each thread has the same priority. This means that the first thread to run will run forever unless it blocks. Then execution will pass to the next ready thread.



Threads can block on any of the standard OS objects, but there is also an additional OS call, `os_task_pass`, that schedules a thread to the ready state and passes execution to the next ready thread.

RTX Source Code

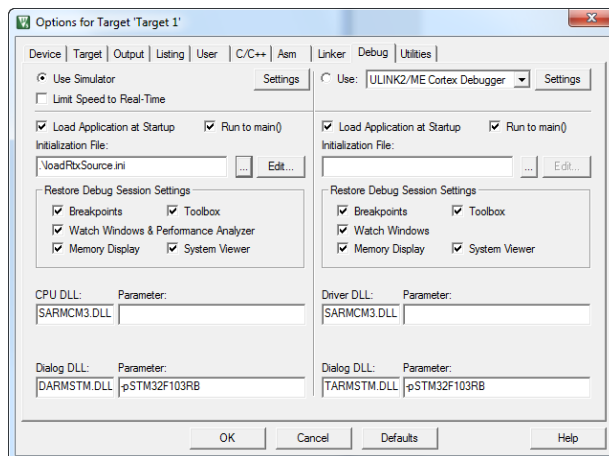
CMSIS-RTOS Keil RTX is included with all versions of the MDK-ARM toolchain. The source code can be found in the following directory of the toolchain.

C:\Keil\ARM\Pack\ARM\CMSIS\<version>\CMSIS\RTOS\RTX

If you want to perform source level debugging of the RTOS code create a text file containing the following command line where the path is the RTX source directory.

SET SRC = <path>

Now add this file to the initialization box in the debugger menu.



Now when you start the debug session the RTX source will be loaded.

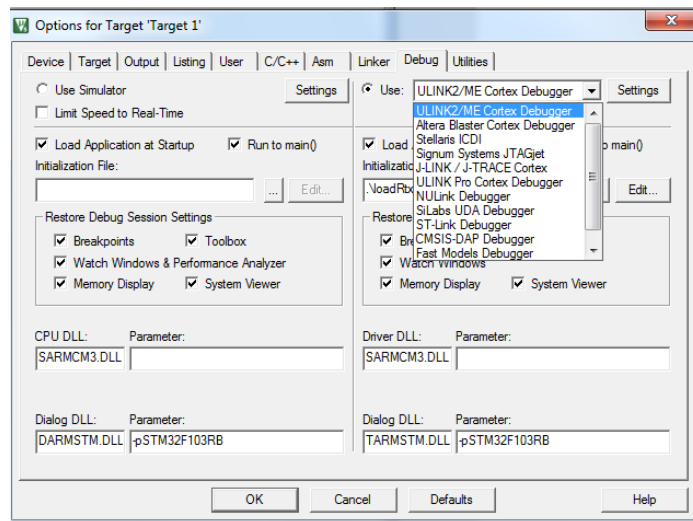
RTX License

CMSIS-RTOS Keil RTX is provided under a three clause BSD license and may be used freely without cost for commercial and non commercial projects. RTX will also compile using the IAR and GCC tools. For more information use the URL below.

<https://www.keil.com/demo/eval/rtx.htm>

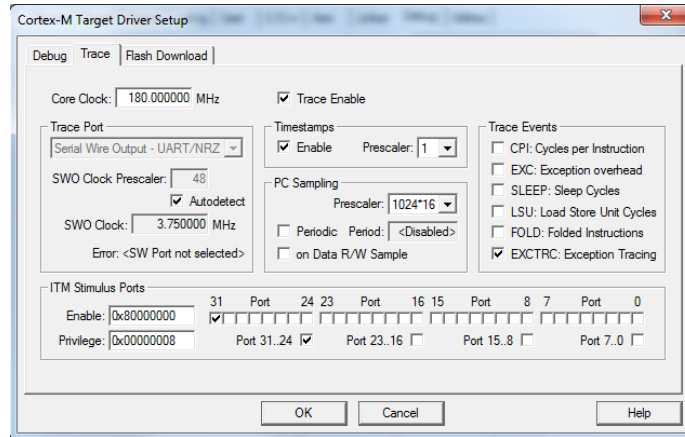
Hardware Debug

During this tutorial we have used the simulator within the μ Vision debugger. To debug real hardware you need to select the hardware interface you are using and select the radio button to enable hardware debug.



If your hardware debugger supports the CoreSight Instrumentation Trace Macrocell (ITM), you will be able to get the same debug information in the 'System and Thread Viewer' and 'Event Viewer'. However, to make these windows active you must enable and configure the ITM. In the debug dialog press the settings button next to the hardware debug interface.

CMSIS-RTOS Tutorial



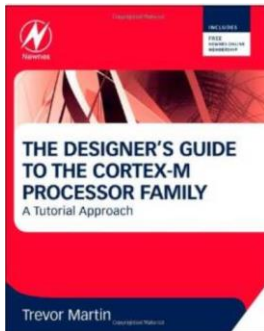
In this menu you must set the Core clock to the CPU frequency of your microcontroller.

Next tick the trace enable box.

Finally enable port 31 of the ITM stimulus ports

This will now receive the additional debug information sent by the RTX kernel.

Further Reading



This tutorial is an excerpt from the Designers Guide to the Cortex-M Processor family by Trevor Martin.

Table of contents

- Introduction to the Cortex-M Processor Family
- Developing Software for the Cortex-M Processors
- Cortex-M Architecture
- Cortex Microcontroller Software Interface Standard
- Advanced Architecture Features
- Developing with CMSIS-RTOS
- Practical DSP for the Cortex-M4
- Debugging with CoreSight

For More details please see the Elsevier Store

Print book ISBN 978-0080982960

<http://store.elsevier.com/product.jsp?isbn=9780080982960&pagename=search>

E Book ISBN 978-0080982991

<http://store.elsevier.com/product.jsp?isbn=9780080982991&pagename=search>

Reference Material

Little Book Of Semaphores Allen B downey

<http://www.greenteapress.com/semaphores/>

Training Courses

In Depth Training courses for the Cortex-M processors are available from Hitex in Germany and the UK.

Training courses in Germany

<http://www.hitex.com/index.php?id=training&L=2>

Training courses in the UK

<http://www.hitex.co.uk/index.php?id=3431>