CHAPTER 6

Developing with CMSIS RTOS

Introduction

This chapter is an introduction to using a small footprint RTOS on a Cortex-M microcontroller. If you are used to writing procedural-based C code on small 8/16-bit microcontrollers, you may be doubtful about the need for such an OS. If you are not familiar with using an RTOS in real-time embedded systems, you should read this chapter before dismissing the idea. The use of an RTOS represents a more sophisticated design approach, inherently fostering structured code development that is enforced by the RTOS API.

The RTOS structure allows you to take a more object-orientated design approach, while still programming in C. The RTOS also provides you with multithreaded support on a small microcontroller. These two features actually create quite a shift in design philosophy, moving us away from thinking about procedural C code and flowcharts. Instead we consider the fundamental program threads and the flow of data between them. The use of an RTOS also has several additional benefits that may not be immediately obvious. Since an RTOS-based project is composed of well-defined threads it helps to improve project management, code reuse, and software testing.

The trade-off for this is that an RTOS has additional memory requirements and increased interrupt latency. Typically, the Keil RTX RTOS will require 500 bytes of RAM and 5 KB of code, but remember that some of the RTOS code would be replicated in your program anyway. We now have a generation of small low-cost microcontrollers that have enough on-chip memory and processing power to support the use of an RTOS. Developing using this approach is therefore much more accessible.

Getting Started

In this chapter 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 file.

Setting Up a Project

The first exercise in the examples accompanying this book provides a PDF document giving a detailed step-by-step guide for setting up an RTOS project. Here we will look at the main differences between a standard C program and an RTOS-based program. First, our µVision project is defined in the normal way. This means that we start a new project and select a microcontroller from the component database. This will add the startup code and configure the compiler, linker, simulation model, and JTAG programming algorithms. Next, we add an empty C module and save it as main.c to start a C-based application. This will give us a project structure similar to that shown below.

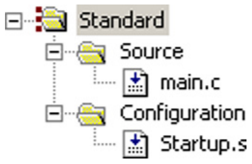


Figure 6.1
A minimal application program consists of an assembler file for the startup code and a C module.

To make this into an RTOS project we must add an RTX configuration file and the RTOS library:

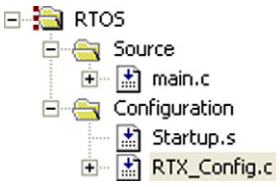


Figure 6.2
The RTOS configuration is held in the file RTX_Config_CM.c which must be added to your project.

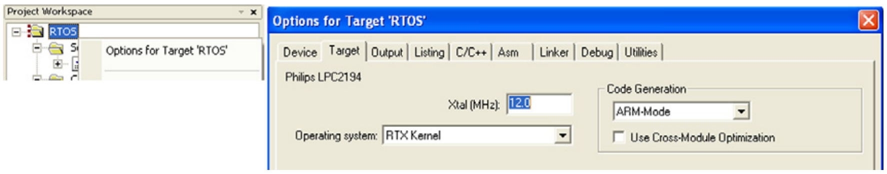
As its name implies, this file holds the configuration settings for the RTOS. The default version of this file can be found in C:\Keil\ARM\Startup\, when you are using the default installation path.

We will examine this file in more detail later, after we have looked more closely at the RTOS and understand what needs to be configured.

To enable our C code to access the CMSIS RTOS API, we need to add an include file to all our application files that use RTOS functions. To do this you must add the following include file in main.c:

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

We must let the debugger know that we are using the RTOS so it can provide additional debug support. This is done by selecting RTX Kernel in the Options for Target\Target menu.



Options for Target 'RTOS'

Philips LPC2194

Xtal (MHz): 12.0

Operating system: RTX Kernel

Code Generation: ARM-Mode

Use Cross-Module Optimization

Figure 6.3
Additional debug support is added to the project by enabling the OS support in the Options for Target menu.

First Steps with CMSIS RTOS

The RTOS itself consists of a scheduler, which supports round-robin, preemptive and cooperative multitasking of program threads, as well as time and memory management services. Interthread 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 that are scheduled by the RTOS kernel.

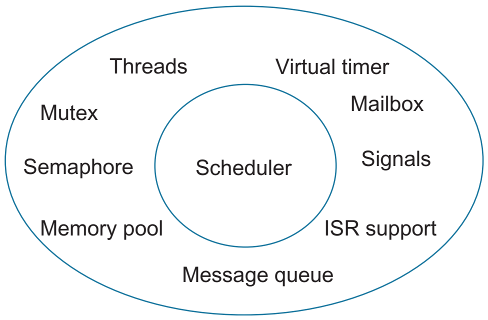


Figure 6.4
The RTOS kernel contains a scheduler that runs program code as threads. Communication between threads is accomplished by RTOS objects such as events, semaphores, mutexes, and mailboxes. Additional RTOS services include time and memory management and interrupt support.

Threads

The building blocks of a typical C program are functions that we call to perform a specific procedure and that 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)        void thread (void)
{                                      {
  ......                                 while(1)
  return(ch);                              {
}                                          ......
                                       }
                                       }
```

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 100 ms then be descheduled to allow thread2 to run for a similar period; thread2 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 that 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.

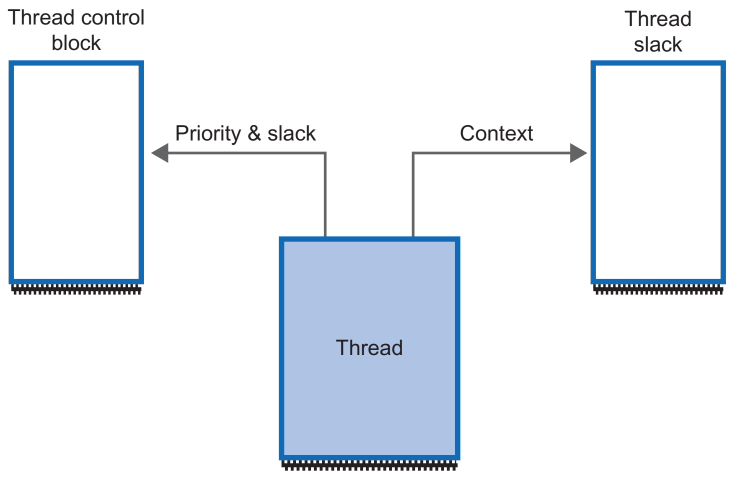


Figure 6.5
Each thread has its own stack for saving its data during a context switch. The thread control block is used by the kernel to manage the active thread.

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 will be in the running state and all the others will be suspended but ready to run or in a wait state, waiting to be triggered by an OS event. The RTOS has various methods of inter-thread communication (i.e., signals, semaphores, messages, etc.). 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 the running state by the RTOS scheduler.

Table 6.1: At Any Given Moment a Single Thread May Be Running

Running	The currently running thread
Ready	Threads ready to run
Wait	Blocked threads waiting for an OS event

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 start running when main() is entered and the main() function becomes the first active thread. Once in main() we can create further threads. It is also possible to configure CMSIS RTOS not to start automatically. The cmsis_os.h include file contains a define:

```
#define osFeature_MainThread 1
```

If this is changed to 0 then the main() function does not become a thread and the RTOS must be started explicitly. You can run any initializing code you want, before starting the RTOS.

```
void main (void)
{
  IODIR1 = 0x00FF0000;    / Do any C code you want
  osKernelStart(osThreadDef(Thread1),NULL);    //Start the RTOS
}
```

The osKernelStart function launches the RTOS but only starts the first thread running. After the OS has been initialized, control will be passed to this thread. When the first thread is created it is also assigned a priority. If there are a number of threads ready to run and they all have the same priority, they will be allotted runtime in a round-robin fashion. However, if a thread with a higher priority becomes ready to run, the RTOS scheduler will deschedule the currently running thread and start the high-priority thread running. This is called preemptive priority-based scheduling. When assigning priorities you have to be careful because the high-priority thread will continue to run until it enters a waiting state or until a thread of equal or higher priority is ready to run.

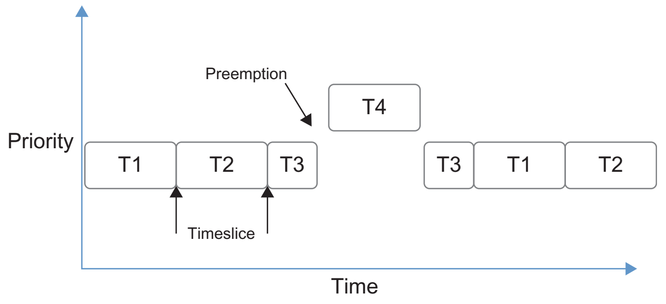


Figure 6.6
Threads of equal priority will be scheduled in a round-robin fashion. High-priority threads will preempt low-priority threads and enter the running state "on demand."

It is also possible to detect if the RTOS is running with osKernelRunning().

```
void main (void)
{
  if(!osKernelRunning()){
```

```
  osKernelStart(osThreadDef(Thread1),NULL);  //Start the RTOS
  }else{
while(1){
  main_thread();  //main is a thread so do stuff here
      }
    }
}
```

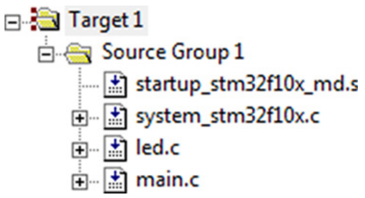
This allows us to enter main() and detect if the kernel has started automatically or needs to be started explicitly.

Exercise: A First CMSIS RTOS Project

This project will take you through the steps necessary to convert a standard C application to a CMSIS RTOS application.

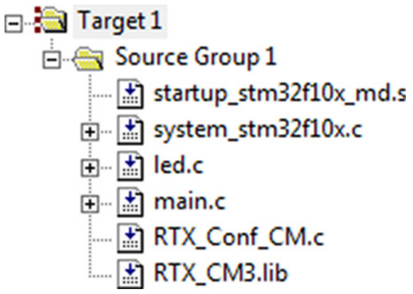
Open the project in C:\exercises\CMSIS RTOS first project.

```
#include "stm32f10x.h"
int main (void) {
  for (;;);
}
```


Target 1
Source Group 1
startup_stm32f10x_md.s
system_stm32f10x.c
led.c
main.c

Here we have a simple program that runs to main and sits in a loop forever.

Right click on the Source Group 1 folder and add the RTX_Conf_CM.c file and the RTX library. You will need to change the "types of file" filter to see the library file.

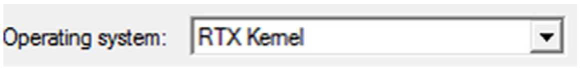

Target 1
Source Group 1
startup_stm32f10x_md.s
system_stm32f10x.c
led.c
main.c
RTX_Conf_CM.c
RTX_CM3.lib

Now your project should look like this.

In main.c add the CMSIS RTOS header file.

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

In the Options for Target\Target tab set RTX Kernel as the OS.

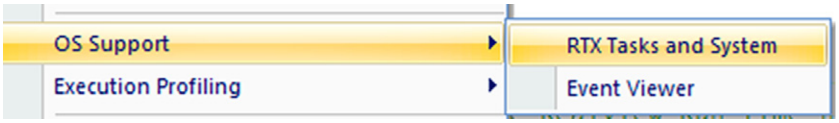


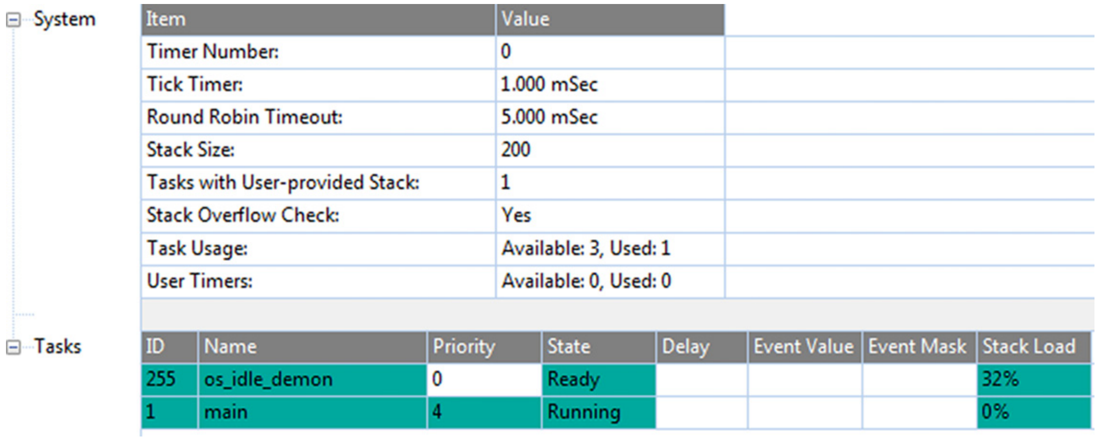
Operating system: RTX Kernel

Build the project and start the debugger.

The code will reach the for loop in main() and halt.

Open the Debug\OS Support\RTX Tasks and System window.





System

Item	Value
Timer Number:	0
Tick Timer:	1.000 mSec
Round Robin Timeout:	5.000 mSec
Stack Size:	200
Tasks with User-provided Stack:	1
Stack Overflow Check:	Yes
Task Usage:	Available: 3, Used: 1
User Timers:	Available: 0, Used: 0

Tasks

ID	Name	Priority	State	Delay	Event Value	Event Mask	Stack Load
255	os_idle_demon	0	Ready				32%
1	main	4	Running				0%

This is a new debug window that shows us the status of the RTOS and status of the running threads.

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. When the first thread is launched it is not assigned a thread ID. The first RTOS function we must therefore call is osThreadGetId() which returns the thread ID number. This is then stored in its ID handle "thread1." 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)
  {
    .........
  }
}
```

Once we have obtained the thread number, we can use the main thread to create further active threads. This is done in two stages. First a thread structure is defined; this allows us to define the thread operating parameters. Then the thread is created from within the main thread.

```
osThreadId main_id;  //create the thread handles
osThreadId thread1_id;
void thread1 (void const *argument);  //function prototype for thread1
osThreadDef(thread1, osPriorityNormal, 1, 0);  //thread definition structure
void main (void)
{
/* Read the Thread-ID of the main thread */
    main_id = osThreadGetId ();
/* Create the second thread and assign its priority */
    Thread1_id = osThreadCreate(osThread(thread1), NULL);
    while(1)
    {
      .........
    }
}
```

The thread structure requires us to define the start address 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. This creates the thread and starts it running. It is also possible to pass a parameter to the thread when it starts.

```
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 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. If necessary a thread can be given additional memory resources by defining a bigger stack size in the thread structure.

```
osThreadDef(thread1, osPriorityNormal, 1, 1024);  //thread definition structure
```

Exercise: Creating and Managing Threads

In this project, we will create and manage some additional threads.

Open the project in c:\exercises\CMSIS RTOS Threads.

When the RTOS starts main() runs as a thread and in addition we will create two additional threads. First, we will 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, osPriorityAboveNormal, 1, 0);
osThreadDef(led_thread1, osPriorityNormal, 1, 0);
```

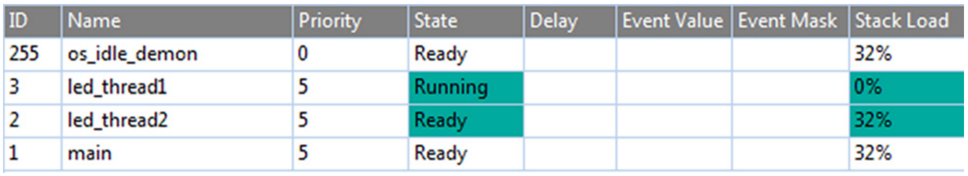
Then in the main() function the two threads are created.

```
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.

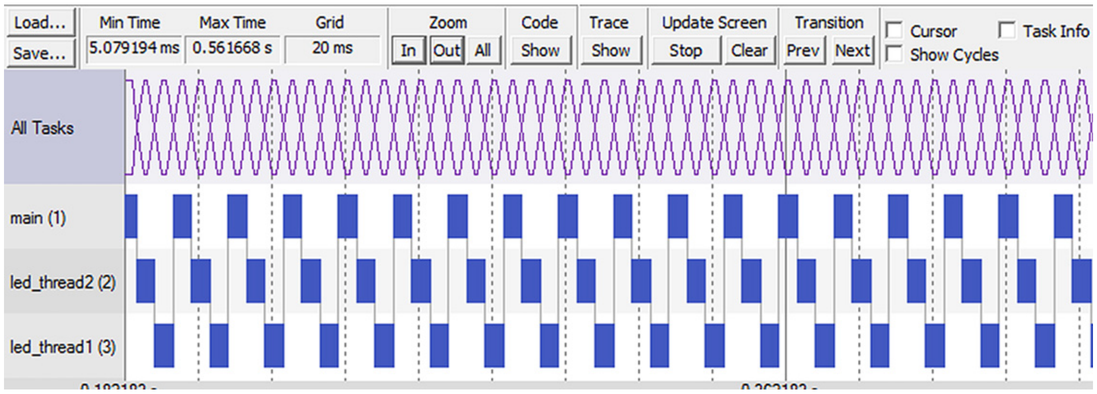
Start running the code and open the Debug\OS Support\RTX Tasks and System window.



ID	Name	Priority	State	Delay	Event Value	Event Mask	Stack Load
255	os_idle_demon	0	Ready				32%
3	led_thread1	5	Running				0%
2	led_thread2	5	Ready				32%
1	main	5	Ready				32%

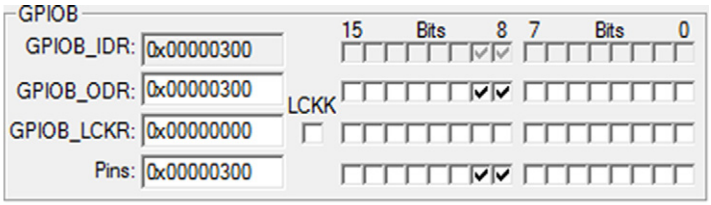
Now we have four active threads with one running and the others ready.

Open the Debug\OS Support\Event Viewer window.



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 the amount of CPU time consumed by each thread.

Now open the Peripherals\General Purpose IO\GPIOB window.



GPIOB_IDR: 0x00000300
GPIOB_ODR: 0x00000300
GPIOB_LCKR: 0x00000000
Pins: 0x00000300

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

```
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. Second, none of the threads blocks; each one runs for its full allocated timeslice, mostly sitting in the delay loop 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.

Table 6.2: CMSIS Priority Levels

osPriorityIdle
osPriorityLow
osPriorityBelowNormal
osPriorityNormal
osPriorityAboveNormal
osPriorityHigh
osPriorityRealTime
osPriorityError

Threads with the same priority level will use round-robin scheduling. Higher priority threads will preempt lower priority threads.

Once the threads are running, there are a small number of OS system calls that 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 cooperative thread switching.

```
osStatus osThreadYeild();//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.

Open the project in c:\exercises\CMSIS RTOS threads.

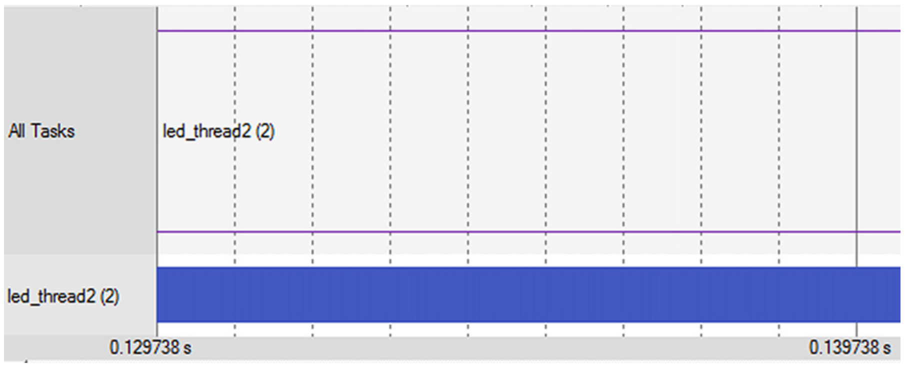
Change the priority of each LED thread to AboveNormal.

```
osThreadDef(led_task2, osPriorityAboveNormal, 1, 0);
osThreadDef(led_task1, osPriorityAboveNormal, 1, 0);
```

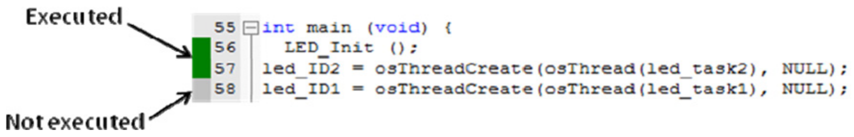
Build the project and start the debugger.

Start running the code.

Open the Debug\OS support\Event Viewer window.



Here we can see thread2 running but no sign of thread1. Looking at the main() function and its coverage monitor shows us what has gone wrong.



```
int main (void) {
  LED_Init ();
led_ID2 = osThreadCreate(osThread(led_task2), NULL);
led_ID1 = osThreadCreate(osThread(led_task1), NULL);
```

Main is running at normal priority and created an LED thread at a higher priority. This thread immediately preempted main and stopped the creation of the second LED thread. To make it even worse, the LED thread never blocks so it will run forever preventing the main thread from ever running.

Exit the debugger and add the following code in the main() function before the osThreadCreate API calls.

```
main_ID = osThreadGetId ();
osThreadSetPriority(main_ID,osPriorityAboveNormal);
```

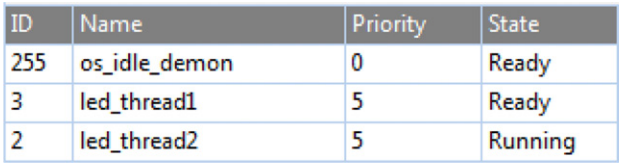
First, we set the main thread handle and then raise the priority of main to the same level as the LED threads.

Since we are only using main to create the LED threads we can stop running it by adding the following line of code after the osThreadCreate() functions.

```
osThreadTerminate(main_ID);
```

Build the code and start the debugger.

Open the RTX Tasks and System window.



ID	Name	Priority	State
255	os_idle_demon	0	Ready
3	led_thread1	5	Ready
2	led_thread2	5	Running

Now we have three threads running with no main thread. Each LED thread also has a higher priority. If you also open the GPIOB window the update rate of the LED pins would have changed because we are no longer using any runtime to sit in the main thread.

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 can 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.

Open the project in c:\exercises\CMSIS RTOS multiple instance.

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 2.

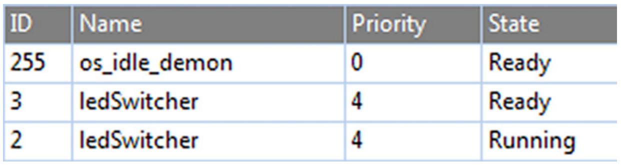
```
osThreadDef(ledSwitcher, osPriorityNormal, 2, 0);
```

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

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

Build the Code and Start the Debugger

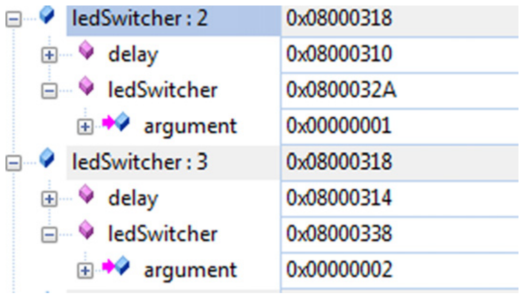
Start running the code and open the RTX Tasks and System window.



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() thread 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 that 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 5 KB, 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.

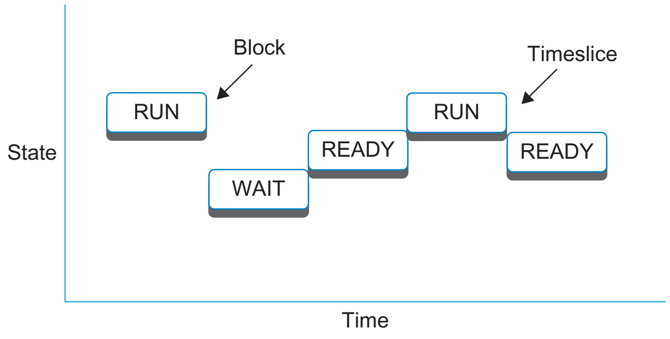


Figure 6.7
During their lifetime thread move through many states. Here a running thread is blocked by an osDelay() call so it enters a wait state. When the delay expires, it moves to ready. The scheduler will place it in the run state. If its timeslice expires, it will move back to ready.

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 of OS calls, it will be descheduled at the end of its timeslice 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 milliseconds 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. We will use this function later when we look at the RTOS thread intercommunication methods.

Exercise: Time Management

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

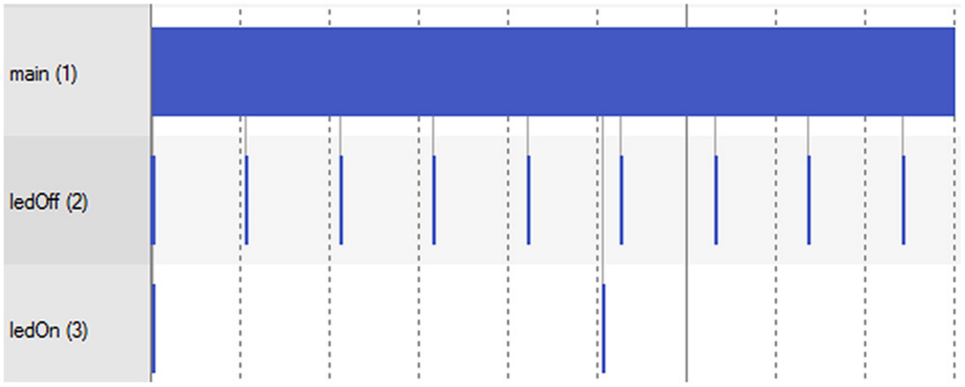
Open the project in C:\exercises\CMSIS RTOS time management.

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

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

Build the project and start the debugger.

Start running the code and open the event viewer window.



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

Virtual Timers

The CMSIS RTOS API can be used to define any number of virtual timers, which act as countdown timers. When they expire, they will run a user callback 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 callback function. The timer must then be instantiated in an RTOS thread.

```
osTimerId timer0_handle = ocTimerCreate (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 callback 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.

Open the project in c:\exercises\CMSIS RTOS timer.

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);
```

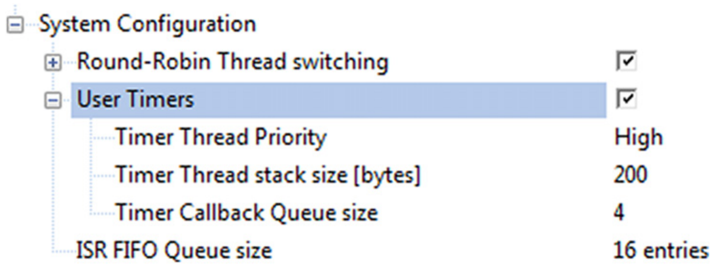
Each timer has a different handle and ID and passes a different parameter to the common callback function.

```
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 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.

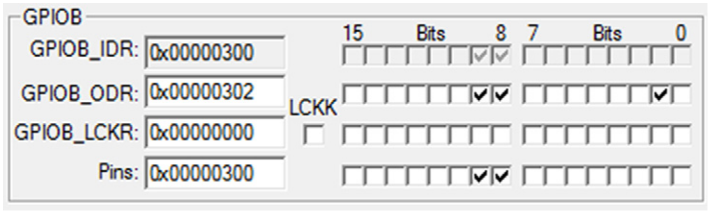


System Configuration	
Round-Robin Thread switching	✓
User Timers	✓
Timer Thread Priority	High
Timer Thread stack size [bytes]	200
Timer Callback Queue size	4
ISR FIFO Queue size	16 entries

In the System Configuration section, make sure the User Timers box is ticked. If this thread is not created the timers will not work.

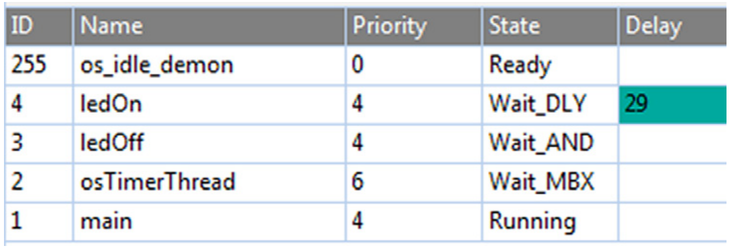
Build the project and start the debugger.

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



GPIOB
GPIOB_IDR: 0x00000300
GPIOB_ODR: 0x00000302
GPIOB_LCKR: 0x00000000
Pins: 0x00000300

There will also be an additional thread running in the RTX threads and System window.



ID	Name	Priority	State	Delay
255	os_idle_demon	0	Ready	
4	ledOn	4	Wait_DLY	29
3	ledOff	4	Wait_AND	
2	osTimerThread	6	Wait_MBX	
1	main	4	Running	

The osDelay() function provides a relative delay from the point at which the delay is started. The virtual timers provide an absolute delay that allows you to schedule code to run at fixed intervals.

Idle Demon

The final timer service provided by the RTOS is not 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_Config.c file. This idle code is in effect a low-priority thread within the RTOS that 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 idle demon is an ideal place to add power management routines.

Exercise Idle Thread

Open the project in c:\exercises\CMSIS RTOS idle.

This is a copy of the virtual timer project. Some code has been added to the idle thread that will execute when our process thread are descheduled.

Open the RTX_Conf_CM.c file and click the text editor tab.

Locate the os_idle_demon thread.

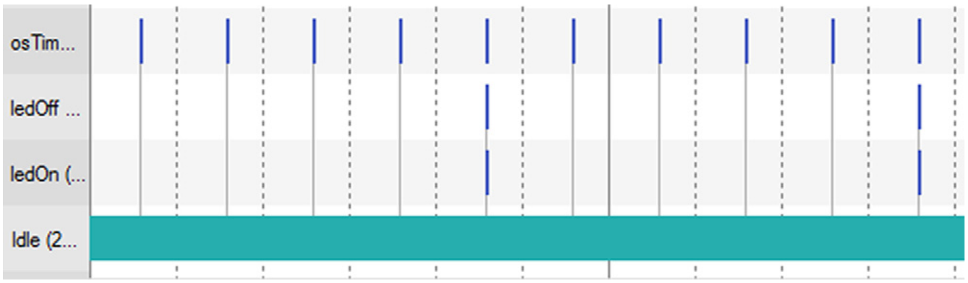
```
void os_idle_demon (void) {
int32_t i;
for (;;) {
GPIOB->ODR ^= 0x1;
for(i=0;i<0x500;i++);
}}
```

When we enter the idle demon it will toggle the least significant port bit.

Build the code and start the debugger.

Run the code and observe the state of the port pins in the GPIOB peripheral window.

This is a simple program that spends 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.

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) {
__wfi();
}}
```

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.

Interthread 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 that can be used to link the threads together to form a meaningful program. The CMSIS RTOS API supports interthread communication with signals, semaphores, mutexes, mailboxes, and message queues.

Signals

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

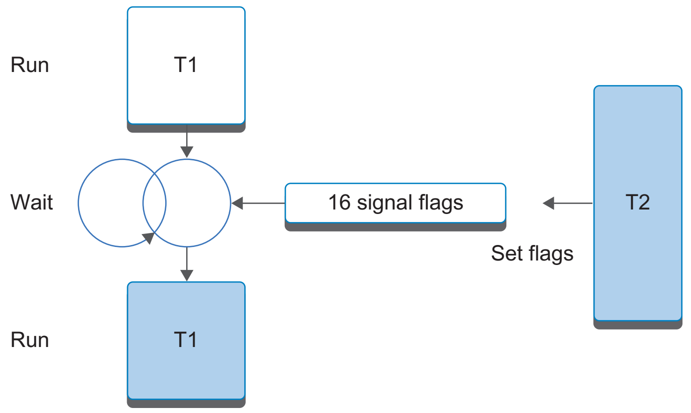


Figure 6.8
Each thread has eight signal flags. A thread may be placed into a waiting state until a pattern of flags is set by another thread. When this happens it will return to the ready state and wait to be scheduled by the kernel.

The signal wait system calls will suspend execution of the thread and place it into the wait_event 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. 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);
```

It is also possible for threads to read the state of their own or other threads' signal flags.

```
Int32_t osSignalGet (osThreadId threaded);
```

While the number of signals available to each thread defaults to eight, it is possible to enable up to 32 flags per thread by changing the signals defined in the CMSIS_os.h header file.

```
#define osFeature_Signals 16
```

Exercise: Signals

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

Open the project in c:\exercises\CMSIS RTOS signals.

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 thread. In addition it sets a signal flag to wake up the second LED thread.

```
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 running the code.

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 microcontroller. While it is possible to run C code in an 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.

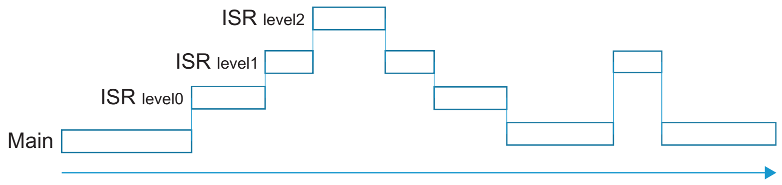


Figure 6.9
A traditional nested interrupt scheme supports prioritized interrupt handling, but has unpredictable stack requirements.

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.

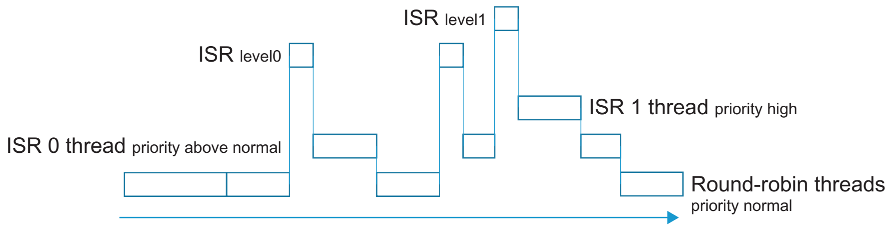

ISR level1
ISR level0
ISR 1 thread priority high
ISR 0 thread priority above normal
Round-robin threads
priority normal

Figure 6.10
Within the RTOS, interrupt code is run as threads. The interrupt handlers signal the threads when an interrupt occurs. The thread priority level defines which thread gets scheduled by the kernel.

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 to 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

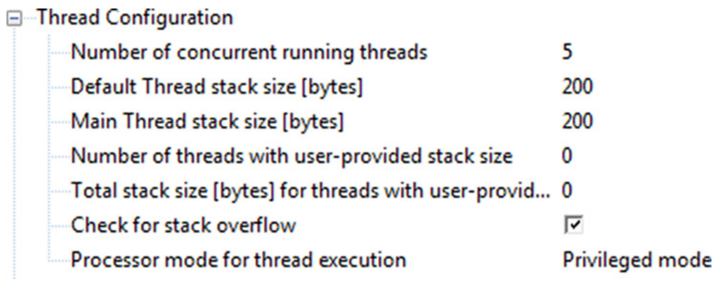
CMSIS RTOS does not introduce any latency in serving interrupts generated by user peripherals. However, operation of the RTOS may be disturbed if you lock out the systick interrupt for a long period of time. This exercise demonstrates a technique of signaling a thread from an interrupt and servicing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

Open the project in c:\exercises\CMSIS RTOS interrupt signal.

In the main() function we initialize the ADC and create an ADC thread that 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 code is running in unprivileged mode. This means that 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 RTX_Conf_CM.c.



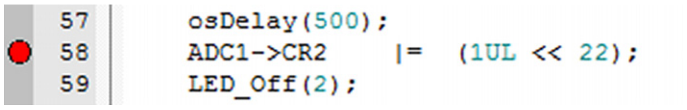
Thread Configuration

Number of concurrent running threads	5
Default Thread stack size [bytes]	200
Main Thread stack size [bytes]	200
Number of threads with user-provided stack size	0
Total stack size [bytes] for threads with user-provid...	0
Check for stack overflow	✓
Processor mode for thread execution	Privileged mode

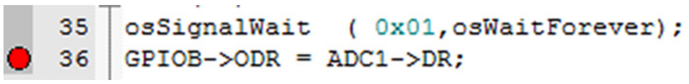
Here we have switched the thread execution mode to privileged, which gives the threads full access to the Cortex-M processor. As we have added a thread, we also need to increase the number of concurrent running threads.

Build the code and start the debugger.

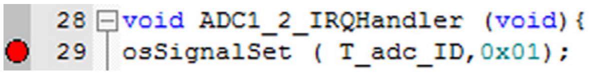
Set breakpoints in LED_Thread2, ADC_Thread, and ADC1_2_IRQHandler.



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



```
osSignalWait  ( 0x01,osWaitForever);
GPIOB->ODR = ADC1->DR;
```



```
void ADC1_2_IRQHandler (void){
osSignalSet ( T_adc_ID,0x01);
```

Run the code.

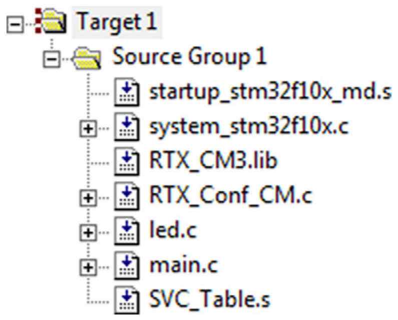
You should hit the first breakpoint, which starts the ADC conversion. Then run the code again and you should enter the ADC interrupt handler. The handler sets the ADC_thread signal and quits. Setting the signal will cause the ADC thread to preempt any other running thread run the ADC service code, and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

Exercise: CMSIS 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.

Open the project in c:\exercises\CMSIS RTOS interrupt_SVC.

In the project we have added a new file called SVC_Tables.c. This is located in c:\Keil\ARM\RL\RTX\SRC\CM.


Target 1
Source Group 1
startup_stm32f10x_md.s
system_stm32f10x.c
RTX_CM3.lib
RTX_Conf_CM.c
led.c
main.c
SVC_Table.s

This is the lookup 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
```

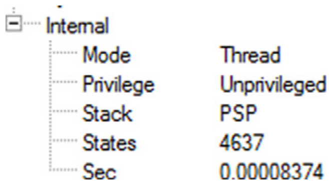
In this file we need to add import name and table entry for each __SVC function that we are going to use. In our example, we only need __SVC_1.

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

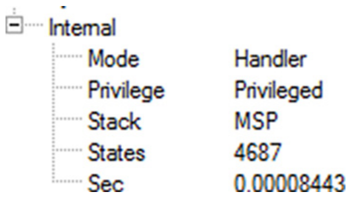
```
void __svc(1) init_ADC (void);
void __SVC_1 (void){
```

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 MSP.

This allows us the set up the ADC and also access the NVIC.

Semaphores

Like events, semaphores are a method of synchronizing activity between two and 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.

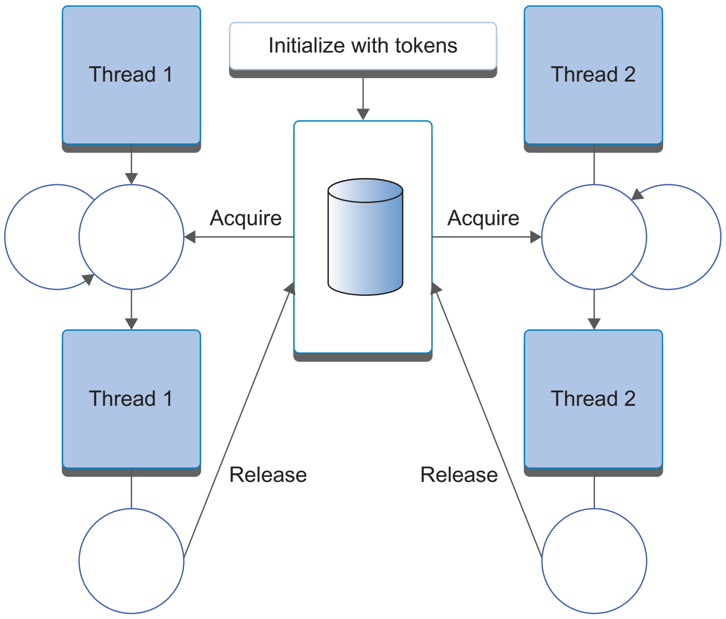


Figure 6.11
Semaphores are used to control access to program resources. Before a thread can access a resource, it must acquire a token. If none is available, it waits. When it is finished with the resource, it must return the token.

Figure 6.11 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 initialized 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 initialize a semaphore with zero tokens and then use one thread to create tokens for 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 osSemaphoreWait() call is used to block a thread until a semaphore token is available, like the osSignalWait() 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 Signaling

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

Open the exercise in C:\exercises\CMSIS RTOS semaphore.

The code first creates a semaphore called sem1 and initializes it with zero tokens.

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

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

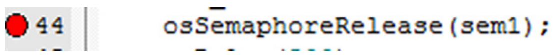
```
void led_Thread1 (void const *argument) {
  for (;;) {
    osSemaphoreWait(sem1, osWaitForever);
    LED_On(1);
    osDelay(500);
      LED_Off(1);
  }
}
```

The second thread 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);
      }}
```

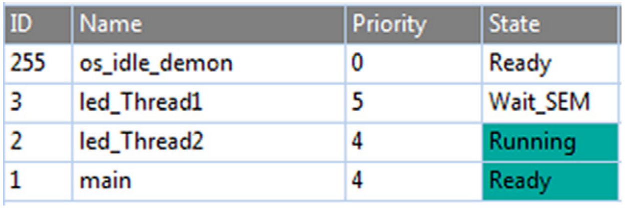
Build the project and start the debugger.

Set a breakpoint in the LED_thread2 thread.



```
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 preempt the lower priority thread 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. Downy. 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 initialize the multiplex semaphore with five tokens. Before a thread can call the ProcessBuffer() function, it must acquire a semaphore token. Once the function has been 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.

Open the project in c:\exercises\CMSIS RTOS semaphore multiplex.

The project creates a semaphore called semMultiplex that 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 threads 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 generalized form of semaphore signaling 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();
osSemaphoreRelease(semArrived2);
     osSemaphoreWait
(semArrived1,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 threads and have to reach a semaphore rendezvous before running the LED functions.

Open the exercise in c:\exercises\CMSIS RTOS semaphore rendezvous.

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 threads is commented out. Since the threads are not synchronized the GPIO pins will toggle randomly.

Exit the debugger.

Uncomment the semaphore code in the LED threads.

Build the project and start the debugger.

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

Now the threads are synchronized by the semaphore and run the LED functions "concurrently."

Barrier Turnstile

Although a rendezvous is very useful for synchronizing the execution of code, it only works for two functions. A barrier is a more generalized form of rendezvous that works to synchronize multiple threads.

```
osSemaphoreId count,barrier;
osSemaphoreDef(counter);
osSemaphoreDef(barrier);
```

```
Unsigned int count;
Turnstile    =   osSemaphoreCreate(osSemaphore(Turnstile), 0);
Turnstile2   =   osSemaphoreCreate(osSemaphore(Turnstile2), 1);
Mutex        =   osSemaphoreCreate(osSemaphore(Mutex), 1);
while(1)
{
        osSemaphoreWait(Mutex,0xffff);   //Allow only one task at a time to run this code
        count = count + 1;                  // Increment count
        if(count = =5)        //When last section of code reaches this point run his code
{
            osSemaphoreWait (Turnstile2,0xffff); //Lock the exit turnstile
            osSemaphoreRelease(Turnstile); //Unlock the first turnstile
}
osSemaphoreRelease(Mutex);                  //Allow other tasks to access the turnstile
osSemaphoreWait(Turnstile,0xFFFF);         //Turnstile Gate
osSemaphoeRelease(Turnstile);
criticalCode();
}
```

In this code we use a global variable to count the number of threads that 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 that are waiting to arrive at the barrier, we send a token to the barrier semaphore.

In the example above, we are synchronizing 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.

Exercise: Semaphore Barrier

In this exercise we will use semaphores to create a barrier to synchronize multiple threads.

Open the project in c:\exercises\CMSIS RTOS semaphore barrier.

Open the exercise in c:\exercises\semaphore rendezvous.

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 threads is commented out. Since the threads are not synchronized the GPIO pins will toggle randomly like in the rendezvous example.

Exit the debugger.

Uncomment the semaphore code in the threads.

Build the project and start the debugger.

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

Now the threads are synchronized 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 accidently creating additional new tokens.

Mutex

Mutex stands for "mutual exclusion." In reality a mutex is a specialized version of a 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 principal 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.

Open the project in c:\exercises\mutex.

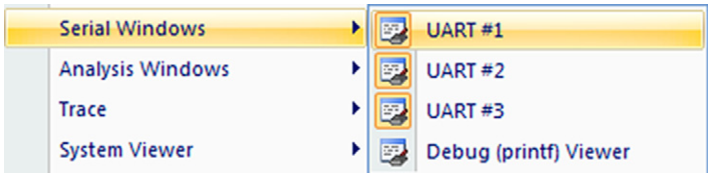
This project declares two threads and both write blocks of characters to the UART. Initially, the mutex is commented out.

```
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);
    }}
```

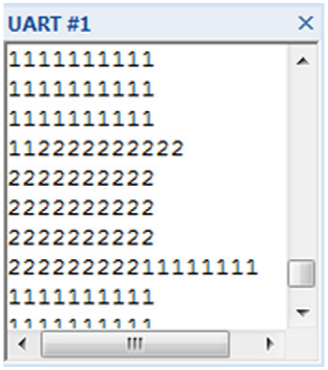
In each thread, the code prints out the thread number at the end of each block of characters; it then prints the carriage return and new line characters.

Build the code and start the debugger.

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



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



```
1111111111
1111111111
1111111111
112222222222
2222222222
2222222222
2222222222
22222222211111111
1111111111
```

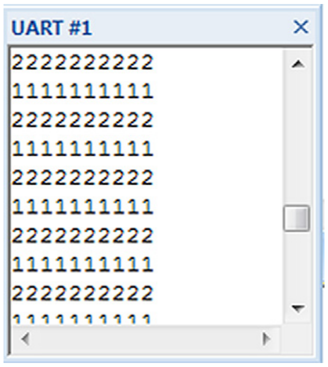
Here we can see that the output data stream is corrupted by each thread writing to the UART without any accessing control.

Exit the debugger.

Uncomment the mutex calls in each thread.

Build the code and start the debugger.

Observe the output of each thread in the console window.



```
2222222222
1111111111
2222222222
1111111111
2222222222
1111111111
2222222222
1111111111
2222222222
```

Now the mutex guarantees each thread 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 that control a mutex token. The Keil RTX RTOS is designed to be a small footprint RTOS so that it can run on even the very small Cortex microcontrollers. Consequently, there is no thread deletion safety. This means that if you delete a thread that 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 interthread 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.

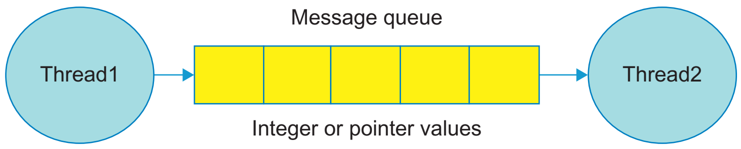


Figure 6.12
CMSIS RTOS supports a message queue that allows simple memory objects to be passed between threads.

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.

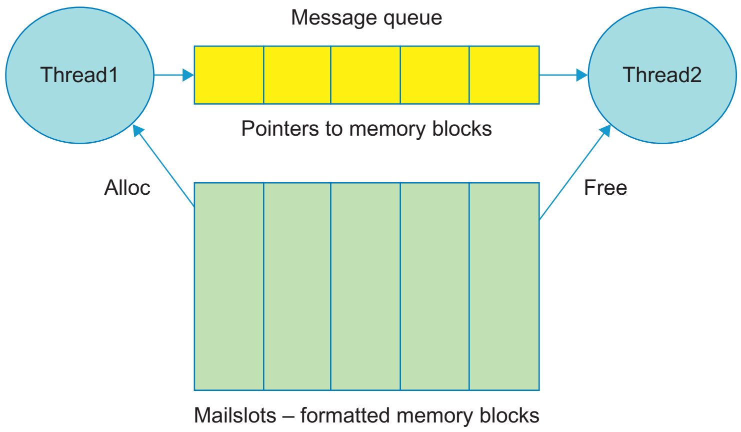

Message queue
Thread1
Thread2
Pointers to memory blocks
Alloc
Free
Mailslots – formatted memory blocks

Figure 6.13
CMSIS RTOS also defines a mailbox system that allows complex memory objects to be transferred between threads. The mailbox is a combination of a memory pool and message queue.

Message Queue

To set up 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 16 storage elements. In this particular queue, each element is defined as an unsigned integer. While 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 integer 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);
```

Then we can read 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.

Open the project in c:\exercises\message queue.

```
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.

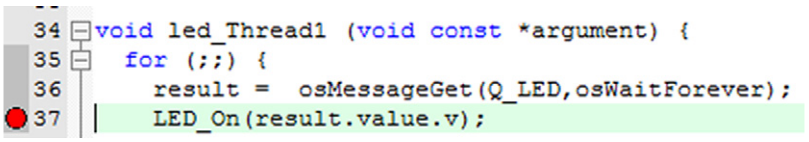
```
osMessagePut(Q_LED,0x1,osWaitForever);
osDelay(100);
```

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.



```
void led_Thread1 (void const *argument) {
  for (;;) {
    result =  osMessageGet(Q_LED,osWaitForever);
    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 dynamically allocatable memory in the form of a memory pool. Here we can declare a structure that combines a number of data elements.

```
typedef struct {
uint8_t LED0;
uint8_t LED1;
uint8_t LED2;
uint8_tLED3;
} 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);
```

Next 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 as follows:

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

Then we can 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 so that the data can be accessed by another thread.

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

The data is then received in another thread as follows.

```
osEvent event; memory_block_t * received;
event = osMessageGet(Q_LED,osWatiForever);
*received = (memory_block *)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);
```

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 a 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 set up and use. First, we need to declare a structure for the mailslot 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 mailslot and populate it with data.

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

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

Once a mailslot 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 mailslot format and in 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 mailslot must be released so it can be reused.

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

Exercise: Mailbox

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

Open the project in c:\exercises\CMSIS RTOS mailbox.

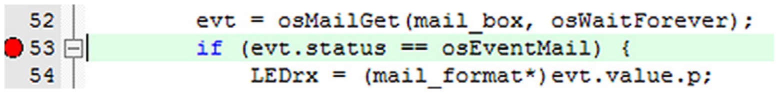
The project uses the LED structure used in the description above to define a 16-slot mailbox.

```
osMailQDef(mail_box, 16, mail_format);
osMailQId mail_box;
int main (void) {
  LED_Init();
  mail_box = osMailCreate(osMailQ(mail_box), NULL);
```

A producer thread then allocates a mailslot, fills it with data, and posts it to the mail queue. The receiving thread waits for a mail message to arrive then reads the data. Once the data has been used the mailslot is released.

Build the code and start the debugger.

Set a breakpoint in the consumer thread and run the code.

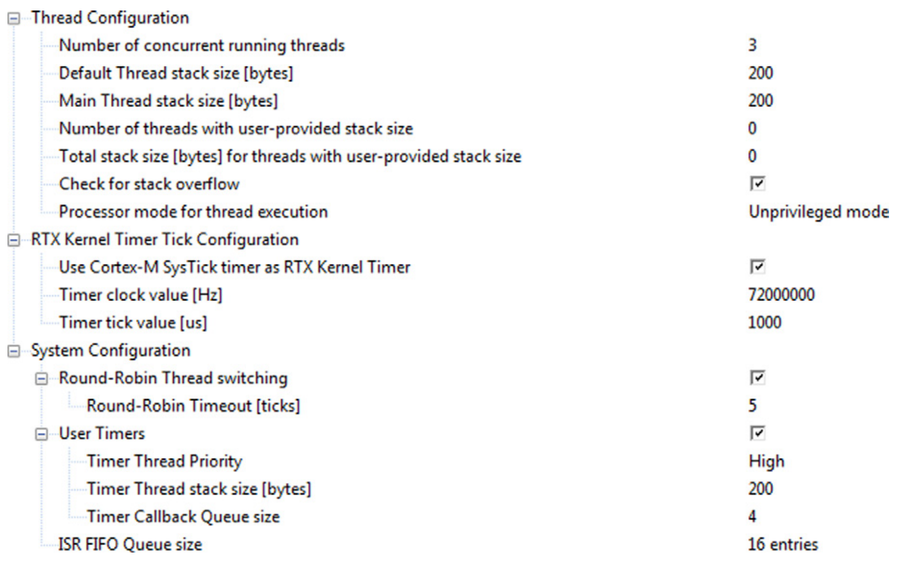


```
        evt = osMailGet(mail_box, osWaitForever);
        if (evt.status == osEventMail) {
            LEDrx = (mail_format*)evt.value.p;
```

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 interthread 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 processors and microcontrollers.



Thread Configuration	
Number of concurrent running threads	3
Default Thread stack size [bytes]	200
Main Thread stack size [bytes]	200
Number of threads with user-provided stack size	0
Total stack size [bytes] for threads with user-provided stack size	0
Check for stack overflow	✓
Processor mode for thread execution	Unprivileged mode
RTX Kernel Timer Tick Configuration	
Use Cortex-M SysTick timer as RTX Kernel Timer	✓
Timer clock value [Hz]	72000000
Timer tick value [us]	1000
System Configuration	
Round-Robin Thread switching	✓
Round-Robin Timeout [ticks]	5
User Timers	✓
Timer Thread Priority	High
Timer Thread stack size [bytes]	200
Timer Callback Queue size	4
ISR FIFO Queue size	16 entries

Figure 6.14
The CMSIS RTOS configuration values are held in the RTX_Config.c file. All of the RTOS parameters can be configured through a microvision wizard.

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

Thread Definition

In the thread definition section, we define the basic resources that 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 thread is created. If we are defining custom stack sizes, we must define the number of threads with custom stacks. Again, the RAM requirement is easily calculated.

During development, the 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. 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 user 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 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_Config_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 set up 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 10 ms.

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 50 ms 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 buffer signal triggers in the event of bursts of interrupt activity.

Scheduling Options

The CMSIS RTOS allows you to build an application with three different kernel scheduling options. These are round-robin scheduling, preemptive scheduling, and cooperative multitasking. A summary of these options follows.

Preemptive 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.

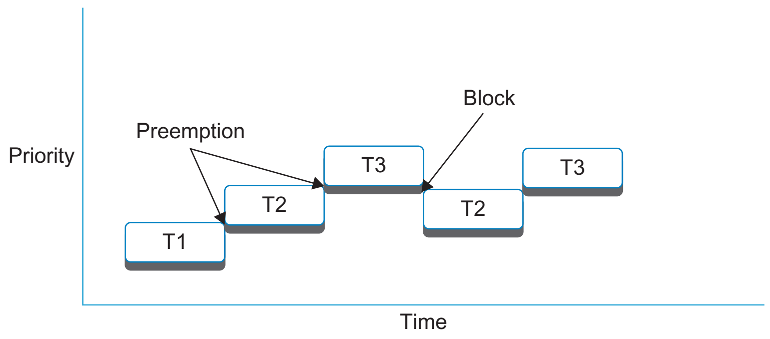


Figure 6.15
In a preemptive RTOS, each thread has a different priority level and will run until it is preempted or has reached a blocking OS call.

This thread will run until it blocks, that is, 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 preemptive scheduling, we build a hierarchy of thread execution, with each thread consuming variable amounts of runtime.

Round-Robin Scheduling

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

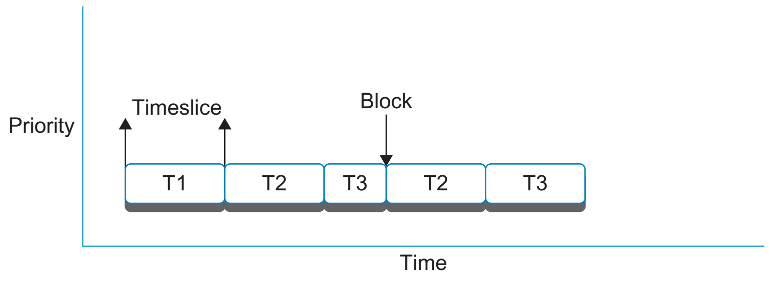


Figure 6.16
In a round-robin RTOS, threads will run for a fixed period or timeslice or until they reach a blocking OS call.

In this scheme, each thread will be allotted a fixed amount of runtime 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 Preemptive Scheduling

As discussed at the beginning of this chapter, the default scheduling option for the Keil RTX RTOS is round-robin preemptive. 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.

Cooperative Multitasking

A final scheduling option is cooperative 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.

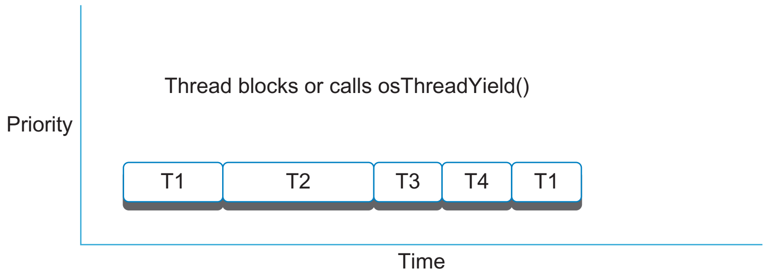


Figure 6.17
In a cooperative RTOS, each thread will run until it reaches a blocking OS call or uses the osThreadYield() function.

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

Priority Inversion

Finally, no discussion of RTOS scheduling would be complete without mentioning priority inversion.

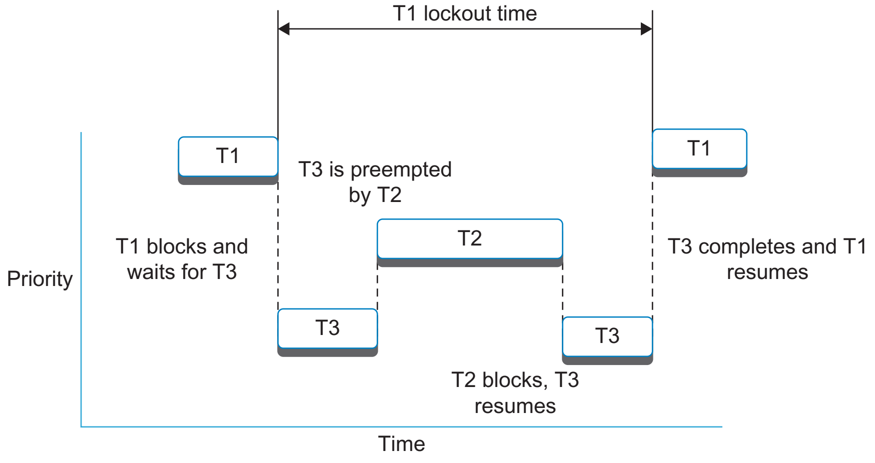


Figure 6.18
A priority inversion is a common RTOS design error. Here a high-priority thread may become delayed or permanently blocked by a medium-priority thread.

In a preemptive scheduling system, it is possible for a high-priority thread T1 to block while it calls a low-priority thread T3 to perform a critical function before T1 continues. However, the low-priority thread T3 could be preempted by a medium priority thread T2. Now T2 is free to run until it blocks (assuming it does) before allowing T3 to resume completing its operation and allowing T1 to resume execution. The upshot is the high-priority thread T1 is blocked and becomes dependent on T2 to complete before it can resume execution.

```
osThreadSetPriority(t_phaseD, osPriorityHigh);  //raise the priority of thread phaseD
  osSignalSet(t_phaseD,0x001);  //trigger it to run//  Call task four to write to the LCD
  osSignalWait(0x01,osWaitForever);  //wait for thread phase D to complete
osThreadSetPriority(t_phaseD,osPriorityBelowNormal);  //lower its priority
```

The answer to this problem is priority elevation. Before T1 calls T3, it must raise the priority of T3 to its level. Once T3 has completed its priority can be lowered back to its initial state.

Exercise: Priority Inversion

In this exercise we will create four threads: two are running at normal priority, one is running at high priority, and one is running at below normal priority. A priority inversion is caused by the high-priority thread waiting for a signal from the below normal task. The normal priority tasks toggle GPIOB port pin 2, while the high and below normal threads toggle GPIOB port pin 1.

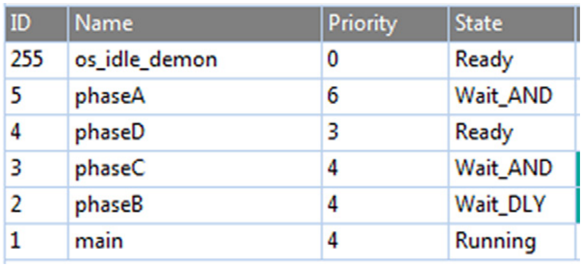
Open the project in c:\exercises\CMSIS RTOS priority inversion.

Build the code and start the debugger.

Open the Peripherals\General Purpose IO\GPIOB window.

Run the code.

The main(), phaseA(), and phaseB() tasks are running at normal priority. PhaseA() is waiting for a signal from phaseD(), but phaseD() cannot run because it is at a low priority and will always be preempted by main(), phaseA(), and phaseB().



ID	Name	Priority	State
255	os_idle_demon	0	Ready
5	phaseA	6	Wait_AND
4	phaseD	3	Ready
3	phaseC	4	Wait_AND
2	phaseB	4	Wait_DLY
1	main	4	Running

Examine the code and understand the problem.

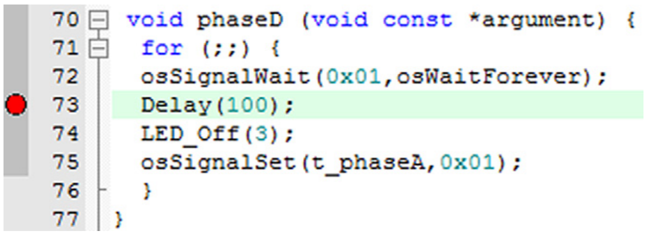
Exit the debugger.

Uncomment the priority control code in the phaseA thread.

```
osThreadSetPriority(t_phaseD, osPriorityHigh);
```

Build the project and start the debugger.

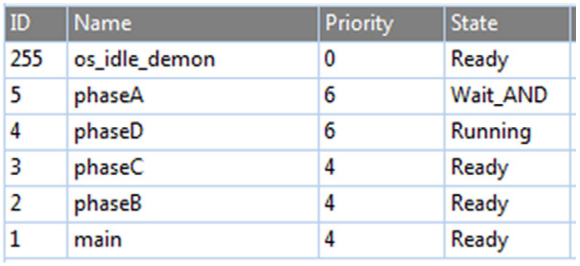
Set a breakpoint in the phaseD thread.



```
void phaseD (void const *argument) {
 for (;;) {
 osSignalWait(0x01,osWaitForever);
 Delay(100);
 LED_Off(3);
 osSignalSet(t_phaseA,0x01);
 }
}
```

Run the code.

Examine the state of the threads when the code halts.



ID	Name	Priority	State
255	os_idle_demon	0	Ready
5	phaseA	6	Wait_AND
4	phaseD	6	Running
3	phaseC	4	Ready
2	phaseB	4	Ready
1	main	4	Ready

The phaseD thread now has the same priority level as phaseA and a higher priority than phaseB or phaseC.

Remove the breakpoint and run the code.

Both GPIO pins will now toggle as expected.