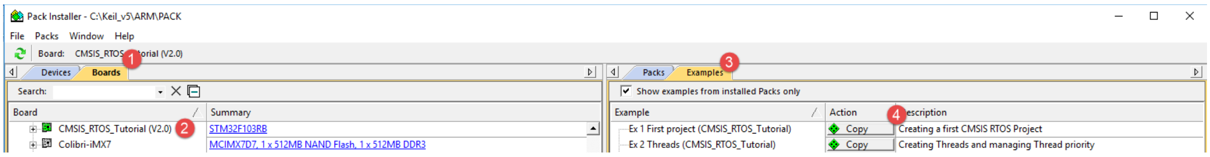
Exercise 1 A first CMSIS-RTOS2 project

This project will take you through the steps necessary to create and debug a CMSIS-RTOS2 based project.

First start the pack installer

This can be done from within microvision from the main toolbar



Pack Installer - C:\Keil_v5\ARM\PACK

Board: CMSIS_RTOS_Tutorial (V2.0)

Board	Summary
CMSIS_RTOS_Tutorial (V2.0)	STM32F103RB
Colibri-iMX7	MCIMX7D7, 1 x 512MB NAND Flash, 1 x 512MB DDR3

☑ Show examples from installed Packs only

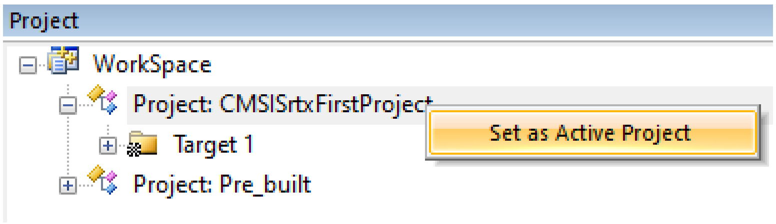
Example	Action	Description
Ex 1 First project (CMSIS_RTOS_Tutorial)	Copy	Creating a first CMSIS RTOS Project
Ex 2 Threads (CMSIS_RTOS_Tutorial)	Copy	Creating Threads and managing Thread priority

In the pack installer select the boards tab, then select the CMSIS-RTOS Tutorial

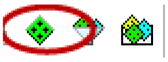
Next select the Examples tab and open the first example by pressing the copy button

This will open a project shell which has been setup for the STM32F103B. This is a basic Cortex-M3 microcontroller. In microvision there is a legacy simulator which has a full model for the STM32F103. This allows us to experiment with CMSIS-RTOS2 without the need for a specific hardware board.

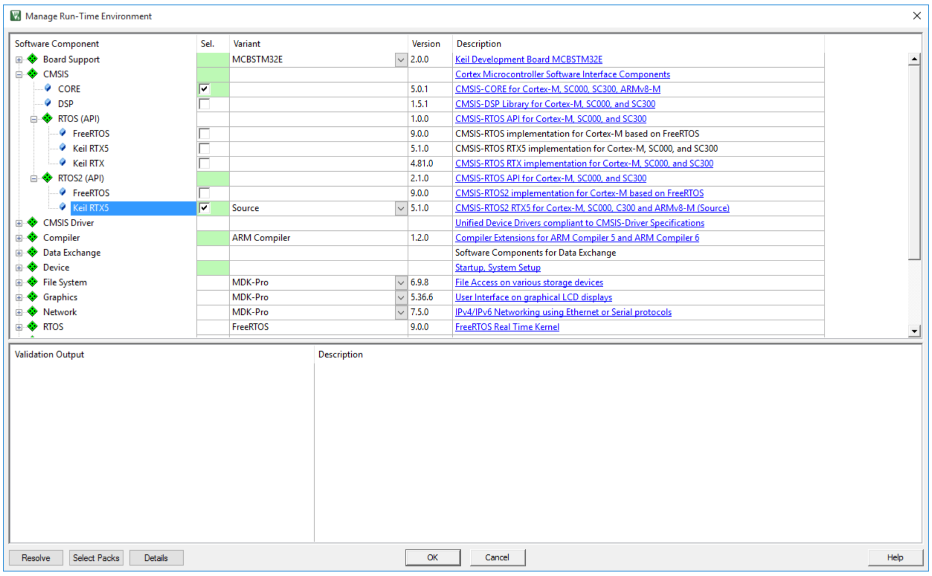
This first project is a multi project workspace. The shell project is set as the active project. A pre built working project is included as a reference. If you want to build this project highlight the project, right click and select "Set as active project". Any compile and debug actions will work on the active project.



Open the Run Time Environment (RTE) by selecting the green diamond on the toolbar



The RTE 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.



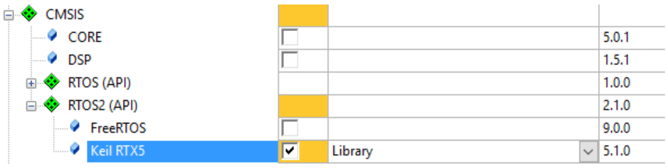
Software Component	Sel.	Variant	Version	Description
Board Support		MCBSTM32E	2.0.0	Keil Development Board MCBSTM32E
CMSIS				Cortex Microcontroller Software Interface Components
CORE	☑		5.0.1	CMSIS-CORE for Cortex-M, SC000, SC300, ARMv8-M
DSP	☐		1.5.1	CMSIS-DSP Library for Cortex-M, SC000, and SC300
RTOS (API)			1.0.0	CMSIS-RTOS API for Cortex-M, SC000, and SC300
FreeRTOS	☐		9.0.0	CMSIS-RTOS implementation for Cortex-M based on FreeRTOS
Keil RTX5	☐		5.1.0	CMSIS-RTOS RTX5 implementation for Cortex-M, SC000, and SC300
Keil RTX	☐		4.81.0	CMSIS-RTOS RTX implementation for Cortex-M, SC000, and SC300
RTOS2 (API)			2.1.0	CMSIS-RTOS API for Cortex-M, SC000, and SC300
FreeRTOS	☐		9.0.0	CMSIS-RTOS2 implementation for Cortex-M based on FreeRTOS
Keil RTX5	☑	Source	5.1.0	CMSIS-RTOS2 RTX5 for Cortex-M, SC000, C300 and ARMv8-M (Source)
CMSIS Driver				Unified Device Drivers compliant to CMSIS-Driver Specifications
Compiler		ARM Compiler	1.2.0	Compiler Extensions for ARM Compiler 5 and ARM Compiler 6
Data Exchange				Software Components for Data Exchange
Device				Startup, System Setup
File System		MDK-Pro	6.9.8	File Access on various storage devices
Graphics		MDK-Pro	5.36.6	User Interface on graphical LCD displays
Network		MDK-Pro	7.5.0	IPv4/IPv6 Networking using Ethernet or Serial protocols
RTOS		FreeRTOS	9.0.0	FreeRTOS Real Time Kernel

Validation Output	Description

Fig 6 Add the RTOS

To configure the project for use with the CMSIS-RTOS2 Keil RTX, simply tick the CMSIS::RTOS2 (API):Keil RTX5 box.

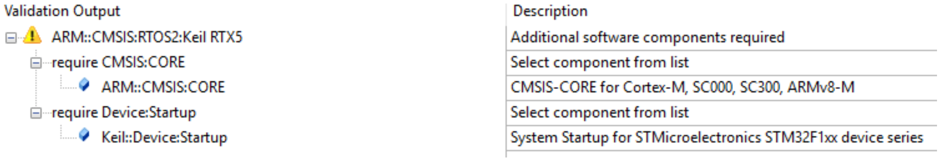
Switch the Keil RTX5 dropdown variant box from 'Source' to 'Library'.



CMSIS			
CORE	☐		5.0.1
DSP	☐		1.5.1
RTOS (API)			1.0.0
RTOS2 (API)			2.1.0
FreeRTOS	☐		9.0.0
Keil RTX5	☑	Library	5.1.0

Fig 7 If the Sel column elements turn Orange then the RTOS requires other components to be added

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.

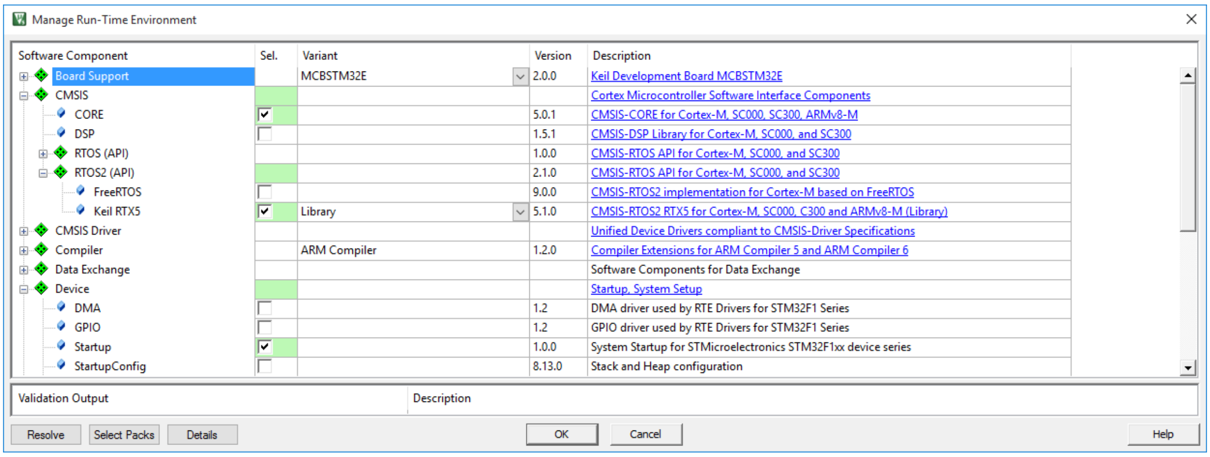


Validation Output	Description
ARM::CMSIS:RTOS2:Keil RTX5	Additional software components required
require CMSIS:CORE	Select component from list
ARM::CMSIS:CORE	CMSIS-CORE for Cortex-M, SC000, SC300, ARMv8-M
require Device:Startup	Select component from list
Keil::Device:Startup	System Startup for STMicroelectronics STM32F1xx device series

Fig 8 The validation box lists the missing components

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.



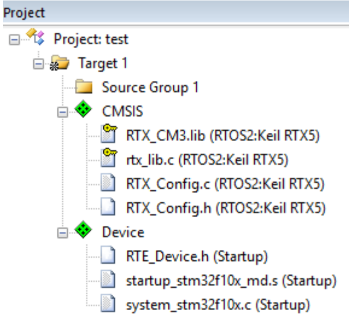
Software Component	Sel.	Variant	Version	Description
Board Support		MCBSTM32E	2.0.0	Keil Development Board MCBSTM32E
CMSIS				Cortex Microcontroller Software Interface Components
CORE	☑		5.0.1	CMSIS-CORE for Cortex-M, SC000, SC300, ARMv8-M
DSP	☐		1.5.1	CMSIS-DSP Library for Cortex-M, SC000, and SC300
RTOS (API)			1.0.0	CMSIS-RTOS API for Cortex-M, SC000, and SC300
RTOS2 (API)			2.1.0	CMSIS-RTOS API for Cortex-M, SC000, and SC300
FreeRTOS	☐		9.0.0	CMSIS-RTOS2 implementation for Cortex-M based on FreeRTOS
Keil RTX5	☑	Library	5.1.0	CMSIS-RTOS2 RTX5 for Cortex-M, SC000, C300 and ARMv8-M (Library)
CMSIS Driver				Unified Device Drivers compliant to CMSIS-Driver Specifications
Compiler		ARM Compiler	1.2.0	Compiler Extensions for ARM Compiler 5 and ARM Compiler 6
Data Exchange				Software Components for Data Exchange
Device				Startup, System Setup
DMA	☐		1.2	DMA driver used by RTE Drivers for STM32F1 Series
GPIO	☐		1.2	GPIO driver used by RTE Drivers for STM32F1 Series
Startup	☑		1.0.0	System Startup for STMicroelectronics STM32F1xx device series
StartupConfig	☐		8.13.0	Stack and Heap configuration

Fig 9 pressing the resolve button adds the missing components and the Sel. Column turns green

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

The other RTOS options will be discussed towards the end of this tutorial.

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



- Project: test
  - Target 1
    - Source Group 1
    - CMSIS
      - RTX_CM3.lib (RTOS2:Keil RTX5)
      - rtx_lib.c (RTOS2:Keil RTX5)
      - RTX_Config.c (RTOS2:Keil RTX5)
      - RTX_Config.h (RTOS2:Keil RTX5)
    - Device
      - RTE_Device.h (Startup)
      - startup_stm32f10x_md.s (Startup)
      - system_stm32f10x.c (Startup)

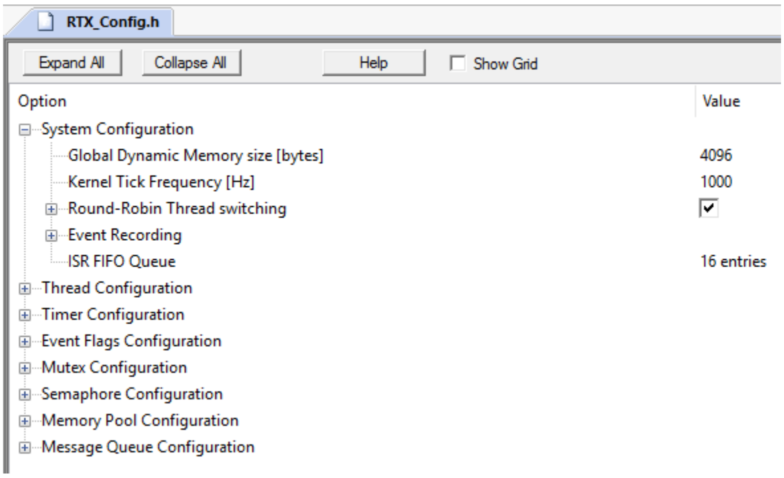
Fig 9 The configured project platform

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 tool chain 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_Config.h file and at the bottom of the editor window select the 'Configuration Wizard' tab.

Fig 10 Selecting the configuration wizard

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



Option	Value
System Configuration	
Global Dynamic Memory size [bytes]	4096
Kernel Tick Frequency [Hz]	1000
Round-Robin Thread switching	☑
Event Recording	
ISR FIFO Queue	16 entries
Thread Configuration	
Timer Configuration	
Event Flags Configuration	
Mutex Configuration	
Semaphore Configuration	
Memory Pool Configuration	
Message Queue Configuration	

Fig 11 The RTX configuration options

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

File name	Description
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_Config.h	Configures Keil RTX

Table 2 Project configuration files

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-click the 'Source Group 1' folder and select 'Add new item to Source Group 1'

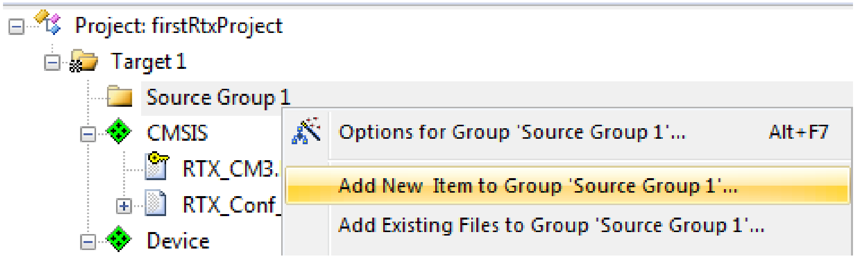
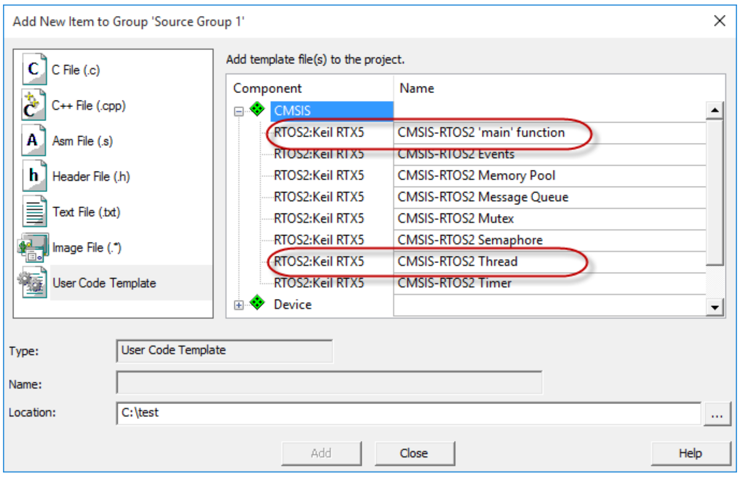


Fig 12 Adding a source module

In the Add new Item dialog select the 'User code template' Icon and in the CMSIS section select the 'CMSIS-RTOS 'main' function' and click Add



Component	Name
CMSIS	
RTOS2:Keil RTX5	CMSIS-RTOS2 'main' function
RTOS2:Keil RTX5	CMSIS-RTOS2 Events
RTOS2:Keil RTX5	CMSIS-RTOS2 Memory Pool
RTOS2:Keil RTX5	CMSIS-RTOS2 Message Queue
RTOS2:Keil RTX5	CMSIS-RTOS2 Mutex
RTOS2:Keil RTX5	CMSIS-RTOS2 Semaphore
RTOS2:Keil RTX5	CMSIS-RTOS2 Thread
RTOS2:Keil RTX5	CMSIS-RTOS2 Timer
Device	

Type: User Code Template

Name:

Location: C:\test

Fig 13 selecting a CMSIS RTOS template

Repeat this but this time select 'CMSIS-RTOS2 Thread'.

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

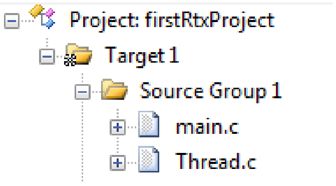


Fig 14 The project with main and Thread code

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.

```
extern int Init_Thread (void);

/*----------------------------------------------------------------------------

* Application main thread

*---------------------------------------------------------------------------*/

void app_main (void *argument) {

        Init_Thread ();

       for (;;) {}

}
```

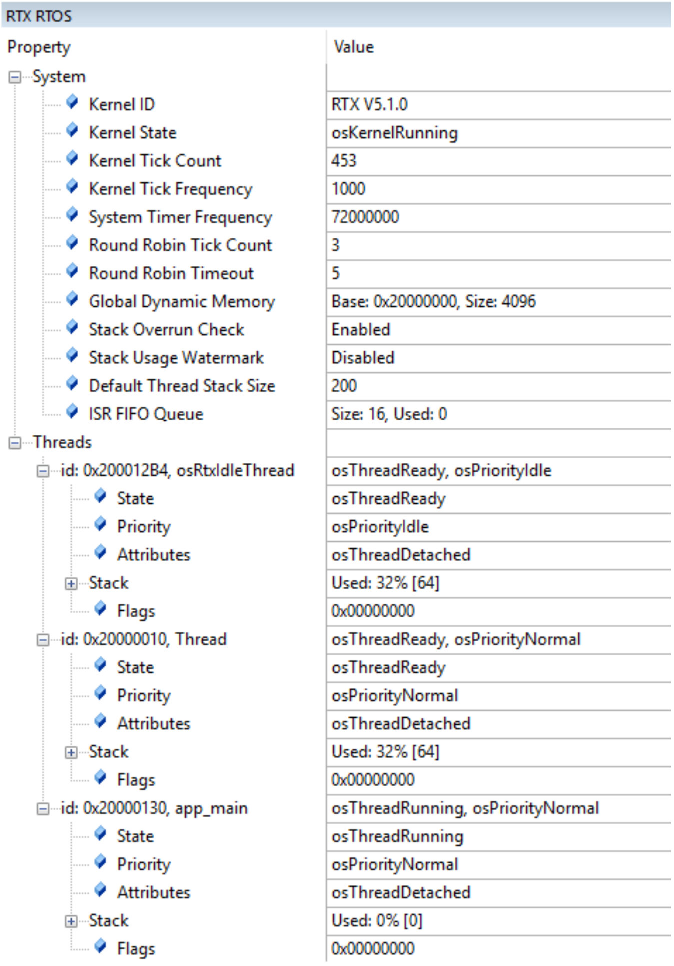
Build the project (F7)

Start the debugger (Ctrl+F5)

This will run the code up to main

Open the Debug → View → Watch Windows →RTX RTOS

Start the code running (F5)



RTX RTOS

Property	Value
System	
Kernel ID	RTX V5.1.0
Kernel State	osKernelRunning
Kernel Tick Count	453
Kernel Tick Frequency	1000
System Timer Frequency	72000000
Round Robin Tick Count	3
Round Robin Timeout	5
Global Dynamic Memory	Base: 0x20000000, Size: 4096
Stack Overrun Check	Enabled
Stack Usage Watermark	Disabled
Default Thread Stack Size	200
ISR FIFO Queue	Size: 16, Used: 0
Threads	
id: 0x200012B4, osRtxIdleThread	osThreadReady, osPriorityIdle
State	osThreadReady
Priority	osPriorityIdle
Attributes	osThreadDetached
Stack	Used: 32% [64]
Flags	0x00000000
id: 0x20000010, Thread	osThreadReady, osPriorityNormal
State	osThreadReady
Priority	osPriorityNormal
Attributes	osThreadDetached
Stack	Used: 32% [64]
Flags	0x00000000
id: 0x20000130, app_main	osThreadRunning, osPriorityNormal
State	osThreadRunning
Priority	osPriorityNormal
Attributes	osThreadDetached
Stack	Used: 0% [0]
Flags	0x00000000

Fig 16 The RTX5 component viewer

This debug view shows all the running threads and their current state. At the moment we have three threads which are app_main, osRtxIdleThread and Thread.

This window is a component view which shows key variables in a software library (component). It is generated by an XML file. It is possible to create such a view for key variables in your application code. This is very useful if you have a long term project or code that you are going to give to a third party.

Exit the debugger

While this project does not actually do anything it demonstrates the few steps necessary to start using CMSIS-RTOS2.

Exercise 2 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.

Open the Pack Installer.

Select the Boards::Designers Guide Tutorial.

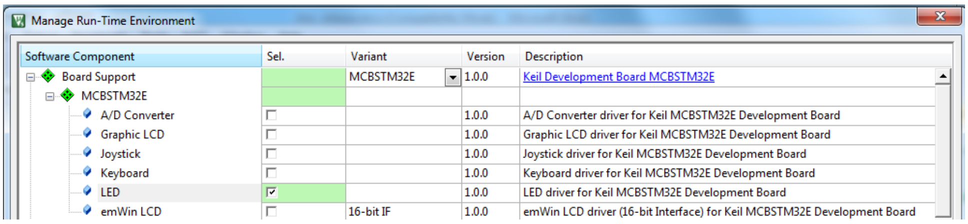
Select the example tab and Copy “EX 9.2 and 9.3 CMSIS-RTOS2 Threads”.

A reference copy of the first exercise is included as Exercise 9.1

This will install the project to a directory of your choice and open the project in **µVision.**

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.



Software Component	Sel.	Variant	Version	Description
Board Support		MCBSTM32E	1.0.0	Keil Development Board MCBSTM32E
MCBSTM32E				
A/D Converter	☐		1.0.0	A/D Converter driver for Keil MCBSTM32E Development Board
Graphic LCD	☐		1.0.0	Graphic LCD driver for Keil MCBSTM32E Development Board
Joystick	☐		1.0.0	Joystick driver for Keil MCBSTM32E Development Board
Keyboard	☐		1.0.0	Keyboard driver for Keil MCBSTM32E Development Board
LED	☑		1.0.0	LED driver for Keil MCBSTM32E Development Board
emWin LCD	☐	16-bit IF	1.0.0	emWin LCD driver (16-bit Interface) for Keil MCBSTM32E Development Board

Fig 19 selecting the board support components

As in the first example main() creates app_main() and starts the RTOS. Inside app_main() we create two additional threads. First we create handles for each of the threads and then define the structures for each thread. The structures are defined in two different ways, for app_main we define the full structure and use NULL to inherit the default values.

```
static const osThreadAttr_t threadAttr_app_main = {
    "app_main",
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    osPriorityNormal,
    NULL,
    NULL
};
```

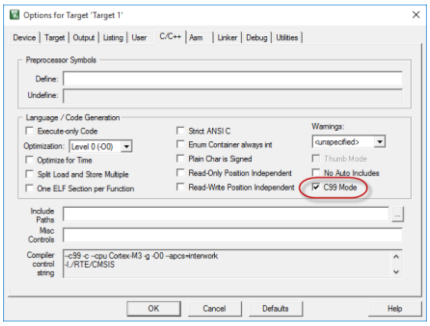
For the thread LED1 a truncated syntax is used as shown below;

```
static const osThreadAttr_t ThreadAttr_LED2 = {
```

```
    .name = "LED_Thread_2",
};
```

In order to use this syntax the compiler options must be changed to allow C99 declarations

Project → Options for Target→C/C++



Define:

Undefine:

☐ Execute-only Code
Optimization: Level 0 (-O0)
☐ Optimize for Time
☐ Split Load and Store Multiple
☐ One ELF Section per Function
☐ Strict ANSI C
☐ Enum Container always int
☐ Plain Char is Signed
☐ Read-Only Position Independent
☐ Read-Write Position Independent
Warnings: <unspecified>
☐ Thumb Mode
☐ No Auto Includes
☑ C99 Mode

Include Paths:

Misc Controls:

Compiler control string:

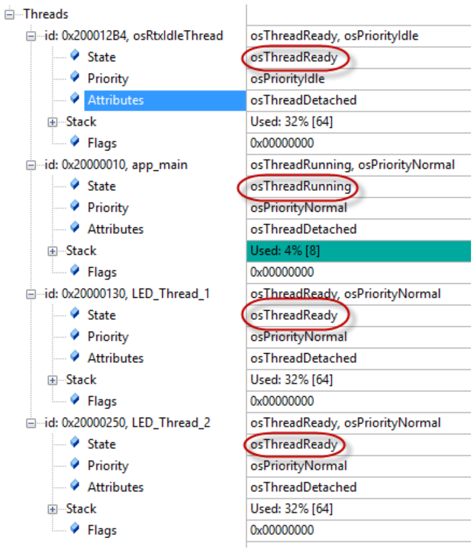
```
--c99 -c --cpu Cortex-M3 -g -O0 --apcs=interwork
-I./RTE/CMSIS
```

Now app_main() is used to first initialise the bank of LED's and then create the two threads. Finally app_main() is terminated with the osThreadExit() api call.

```
void app_main (void *argument) {

        LED_Initialize ();

        led_ID1 = osThreadNew(led_thread1, NULL, &threadAttr_LED1);

        led_ID1 = osThreadNew(led_thread2, NULL, &threadAttr_LED2);

        osThreadExit();

}
```

Build the project and start the debugger

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

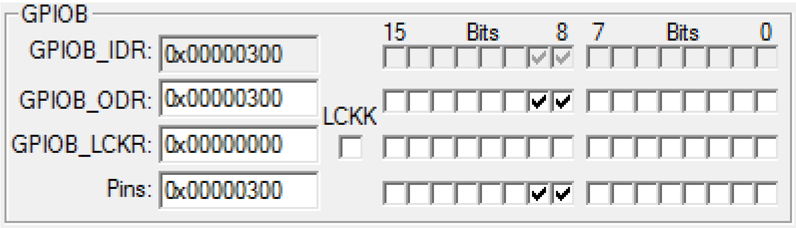


Threads	
id: 0x200012B4, osRtxIdleThread	osThreadReady, osPriorityIdle
State	osThreadReady
Priority	osPriorityIdle
Attributes	osThreadDetached
Stack	Used: 32% [64]
Flags	0x00000000
id: 0x20000010, app_main	osThreadRunning, osPriorityNormal
State	osThreadRunning
Priority	osPriorityNormal
Attributes	osThreadDetached
Stack	Used: 4% [8]
Flags	0x00000000
id: 0x20000130, LED_Thread_1	osThreadReady, osPriorityNormal
State	osThreadReady
Priority	osPriorityNormal
Attributes	osThreadDetached
Stack	Used: 32% [64]
Flags	0x00000000
id: 0x20000250, LED_Thread_2	osThreadReady, osPriorityNormal
State	osThreadReady
Priority	osPriorityNormal
Attributes	osThreadDetached
Stack	Used: 32% [64]
Flags	0x00000000

Fig 20 The running Threads

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

Now open the Peripherals → General Purpose IO → GPIOB window



GPIOB_IDR: 0x00000300

GPIOB_ODR: 0x00000300

GPIOB_LCKR: 0x00000000

Pins: 0x00000300

Fig 22 the peripheral window shows the LED pin activity

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.

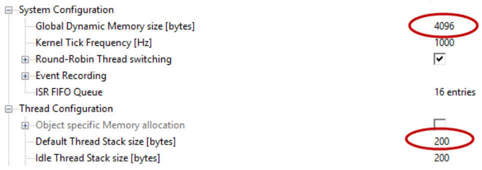
Exercise 3 Memory Model

In this exercise we will create a thread with a custom memory allocation and also create a thread with a static memory allocation.

In the Pack Installer select “Ex memory model” and copy it to your tutorial directory.

This exercise uses the same two LED flasher threads as the previous exercise.

Open cmsis::rtx_config.c



System Configuration	
Global Dynamic Memory size [bytes]	4096
Kernel Tick Frequency [Hz]	1000
Round-Robin Thread switching	☑
Event Recording	
ISR FIFO Queue	16 entries
Thread Configuration	
Object specific Memory allocation	☐
Default Thread Stack size [bytes]	200
Idle Thread Stack size [bytes]	200

The threads are allocated memory from the global dynamic memory pool and by default each thread is allocated 200 bytes

When we create led-thread1 we pass the attribute structure which has been modified to create the thread with a custom stack size of 1025 bytes

```
static const osThreadAttr_t ThreadAttr_LED1 = {
        "LED_Thread_1",
        NULL,   //attributes
        NULL,   //cb memory
        NULL,   //cb size
        NULL,   //stack memory
        1024,   //stack size        This memory is allocated from the global memory pool
        osPriorityNormal,
        NULL,   //trust zone id
        NULL    //reserved
};
```

The second thread is created with a statically defined thread control block and a statically defined stack space. First we need to define an array of memory for the stack space;

```
static uint64_t LED2_thread_stk[64];
```

Followed by a custom RTX thread control block;

```
static osRtxThread_t LED2_thread_tcb;
```

The custom type osRtxThread is defined in rtx_os.h

Now we can create a thread attribute which statically allocates the both the stack and the task control block;

```
static const osThreadAttr_t ThreadAttr_LED2 = {

        "LED_Thread_2",

        NULL,                           //attributes

        &LED2_thread_tcb,               //cb memory

        sizeof(LED2_thread_tcb),        //cb size

        &LED2_thread_stk[0],             //stack memory      Here the control block and user stack space are statically allocated

        sizeof(LED2_thread_stk),        //stack size

        osPriorityNormal,

        NULL,                   //trust zone id

        NULL                    //reserved

};
```

Build the code.

Start the debugger and check it runs

The statically allocated thread will not appear in the RTOS component viewer as the custom memory allocation is not detected

Exit the debugger

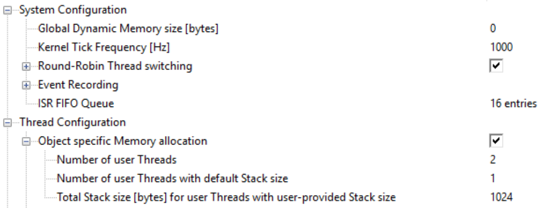
In the CMSIS:RTX_Conf.c file we can change the memory model to use "Object Specific" memory allocation.

Set the Global Dynamic memory size to zero

In thread configuration enable the Object specific memory model

Set the number of threads to two

Number of user threads wit default stack size to 1 and total stack size for threads with use provided stack to 1024.



System Configuration	
Global Dynamic Memory size [bytes]	0
Kernel Tick Frequency [Hz]	1000
Round-Robin Thread switching	☑
Event Recording	
ISR FIFO Queue	16 entries
Thread Configuration	
Object specific Memory allocation	☑
Number of user Threads	2
Number of user Threads with default Stack size	1
Total Stack size [bytes] for user Threads with user-provided Stack size	1024

In total we have three user threads but one has statically allocated memory so our thread object pool only needs to accommodate two. One of those threads (Led_thread1) has a custom stack size of 1024 bytes. We need to provide this information to the RTOS so it can work out the total amount of memory to allocate for thread use.

Enable the MUTEX object

Set the number of mutex objects to 5

Mutex Configuration
Object specific Memory allocation ☑
Number of Mutex objects 5

We will use mutexes later but they are concerned with protecting access to resources. The RTOS creates a number to protect access to the run time ‘C’ library from different threads.

Build the code

Start the debugger

Run the code

Now we have one thread using statically located memory and object using object specific memory.

Exercise 4 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);
    }
}
```

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.

First we can create two different thread definitions with different debug names

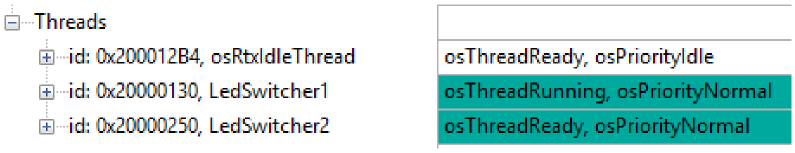
```
static const osThreadAttr_t ThreadAttr_LedSwitcher1 = {
        .name =   "LedSwitcher1",
    };

static const osThreadAttr_t ThreadAttr_LedSwitcher2 = {
    .name =       "LedSwitcher2",
    };
```

Next we can create two instances of the same thread code

```
led_ID1 = osThreadNew(ledSwitcher,(void *) 1UL,   &ThreadAttr_LedSwitcher1);
led_ID2 = osThreadNew(ledSwitcher,(void *) 2UL,   &ThreadAttr_LedSwitcher2);
```

Build the code and start the debugger

Start the code running and open the RTX5tasks and system window

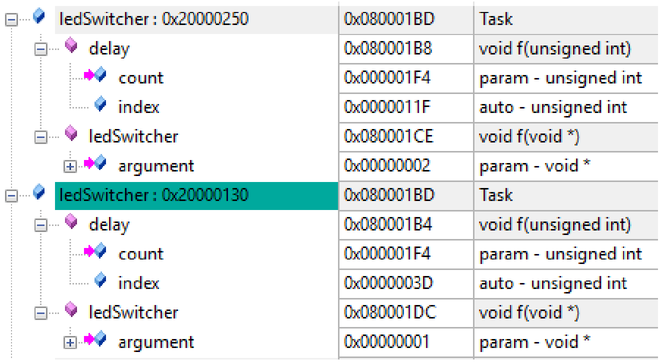


Threads
id: 0x200012B4, osRtxIdleThread | osThreadReady, osPriorityIdle
id: 0x20000130, LedSwitcher1 | osThreadRunning, osPriorityNormal
id: 0x20000250, LedSwitcher2 | osThreadReady, osPriorityNormal

Fig 25 Multiple instances of thread running

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

Examine the Call stack + locals window



ledSwitcher : 0x20000250	0x080001BD	Task
delay	0x080001B8	void f(unsigned int)
count	0x000001F4	param - unsigned int
index	0x0000011F	auto - unsigned int
ledSwitcher	0x080001CE	void f(void *)
argument	0x00000002	param - void *
ledSwitcher : 0x20000130	0x080001BD	Task
delay	0x080001B4	void f(unsigned int)
count	0x000001F4	param - unsigned int
index	0x0000003D	auto - unsigned int
ledSwitcher	0x080001DC	void f(void *)
argument	0x00000001	param - void *

Fig 26 The watch window is thread aware

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.

Exercise 5 Joinable threads

In this exercise we will create a thread which in turn spawns two joinable threads. The initial thread will then call osThreadJoin() to wait until each of the joinable threads has terminated.

In the Pack Installer select "Ex 4 Join " and copy it to your tutorial directory.

Open main.c

In main.c we create a thread called worker_Thread and define it as joinable in the thread attribute structure.

When the RTOS starts we create the led_thread() as normal.

```
__NO_RETURN void led_thread1 (void *argument) {
for (;;) {
        worker_ID1   = osThreadNew(worker_thread,(void *) LED1_ON, &ThreadAttr_worker);
        LED_On(2);
        osThreadJoin(worker_ID1);
        ……………………….
```

In this thread we create an instance of the worker thread and then call osJoin() to join it. At this point the led_thread enters a waiting state and the worker thread runs.

```
void worker_thread (void *argument) {
if((uint32_t)argument == LED1_ON) {
LED_On(1);
}
else if ((uint32_t)argument == LED1_OFF){
LED_Off(1);
}
delay(500);
osThreadExit();
}
```

When the worker thread runs it flashes the led but instead of having an infinite loop it calls osExit(); to terminate its runtime which will cause led_thread1 to leave the waiting state and enter the ready state and in this example then enter the run state.

Build the code

Start the debugger

Open the View\watch\RTOS window

Run the code and watch the behavior of the threads

Exercise 6 Time Management

In this exercise we will look at using the basic time osDelay() and delayUntil() functions

In the Pack Installer select “Ex 6 Time Management” 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 and osDelayUntil() API calls. LED2 is toggled every 100mS and LED1 is toggled every 500mS

```
        void ledOn (void  *argument) {

         for (;;) {

                LED_On(1);

                osDelay(500);

                LED_Off(1);

                osDelay(500);

        }}
```

In the Led2 thread we use the osDelayUntil() function to create a 1000 tick delay

```
__NO_RETURN void led2 (void  *argument) {

        for (;;) {

                ticks = osKernelGetTickCount();

                   LED_On(2);

                osDelayUntil((ticks + 1000));          //Toggle LED 2 with an absolute delay

                LED_Off(2);

                osDelayUntil((ticks+2000));

        }

}
```

Build the project and start the debugger

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 appMain thread 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 threads.

Exercise 7 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 “Ex 7 Virtual Timers” 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

```
osTimerId_t timer0,timer1,timer2,timer3;

static const  osTimerAttr_t timerAttr_timer0 = {

        .name = "timer_0",

};

static const  osTimerAttr_t timerAttr_timer1 = {

        .name = "timer_1",

};

static const  osTimerAttr_t timerAttr_timer2 = {

        .name = "timer_2",

};

static const  osTimerAttr_t timerAttr_timer3 = {

        .name = "timer_3",

};
```

They are then initialized in the main function;

```
timer0 = osTimerNew(&callback, osTimerPeriodic,(void *)0, &timerAttr_timer0);

timer1 = osTimerNew(&callback, osTimerPeriodic,(void *)1, &timerAttr_timer1);

timer2 = osTimerNew(&callback2, osTimerPeriodic,(void *)2, &timerAttr_timer2);

timer3 = osTimerNew(&callback2, osTimerPeriodic,(void *)3, &timerAttr_timer3);
```

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

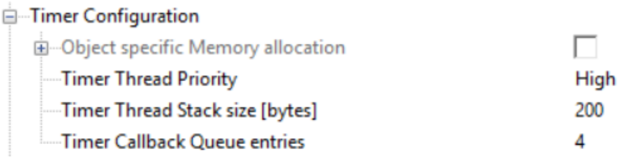
```
void callback(void const *param){

switch( (uint32_t) param){
```

```
          case 0:
                    GPIOB->ODR ^= 0x8;
          break;
          case 1:
                    GPIOB->ODR ^= 0x4;
          break;
          case 2:
                    GPIOB->ODR ^= 0x2;
          break;
}}
```

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 RTX5 configuration file.

Open the RTX_Config.h file and press the configuration wizard tab



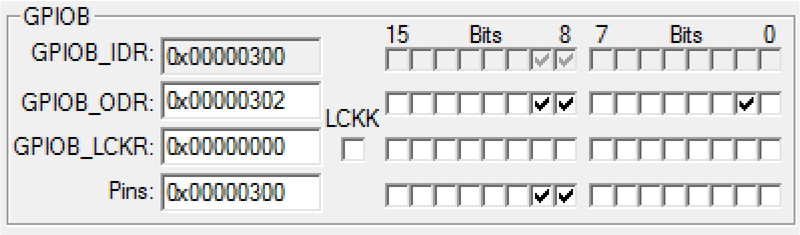
Timer Configuration
Object specific Memory allocation ☐
Timer Thread Priority High
Timer Thread Stack size [bytes] 200
Timer Callback Queue entries 4

Fig 29 configuring the virtual timers

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

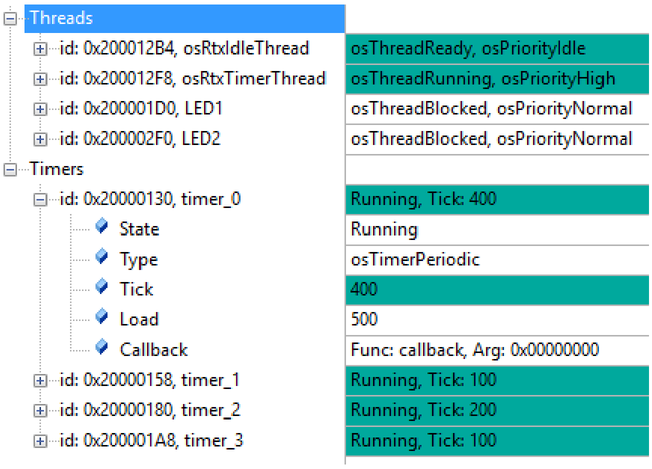
Run the code and observe the activity of the GPIOB pins in the peripheral window



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

Fig 30 The user timers toggle additional LED pins

There will also be an additional thread running in the System and Thread Viewer window



Threads	
id: 0x200012B4, osRtxIdleThread	osThreadReady, osPriorityIdle
id: 0x200012F8, osRtxTimerThread	osThreadRunning, osPriorityHigh
id: 0x200001D0, LED1	osThreadBlocked, osPriorityNormal
id: 0x200002F0, LED2	osThreadBlocked, osPriorityNormal
Timers	
id: 0x20000130, timer_0	Running, Tick: 400
State	Running
Type	osTimerPeriodic
Tick	400
Load	500
Callback	Func: callback, Arg: 0x00000000
id: 0x20000158, timer_1	Running, Tick: 100
id: 0x20000180, timer_2	Running, Tick: 200
id: 0x200001A8, timer_3	Running, Tick: 100

Fig 31 The user timers create an additional osTimerThread

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

Exercise 8 Idle Thread

In the Pack Installer select “Ex 8 Idle” and copy it to your tutorial directory.

This is a copy of the virtual timer project.

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

Locate the idle thread

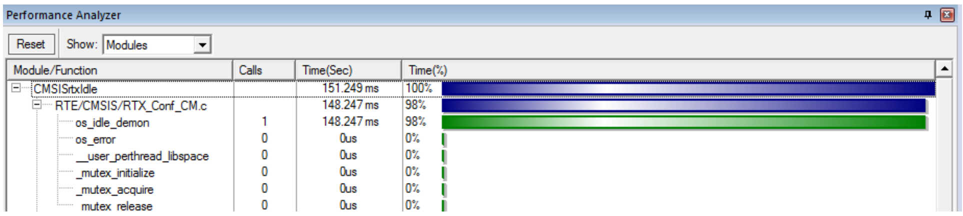
```
__NO_RETURN void osRtxIdleThread (void *argument){

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

Open the View → Analysis Windows → Performance Analyzer.



Module/Function	Calls	Time(Sec)	Time(%)
CMSISrtxIdle		151.249 ms	100%
RTE/CMSIS/RTX_Conf_CM.c		148.247 ms	98%
os_idle_demon	1	148.247 ms	98%
os_error	0	0us	0%
__user_perthread_libspace	0	0us	0%
_mutex_initialize	0	0us	0%
_mutex_acquire	0	0us	0%
mutex release	0	0us	0%

Fig 33 The performance analyser shows that most of the run time is being spent in the idle loop

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

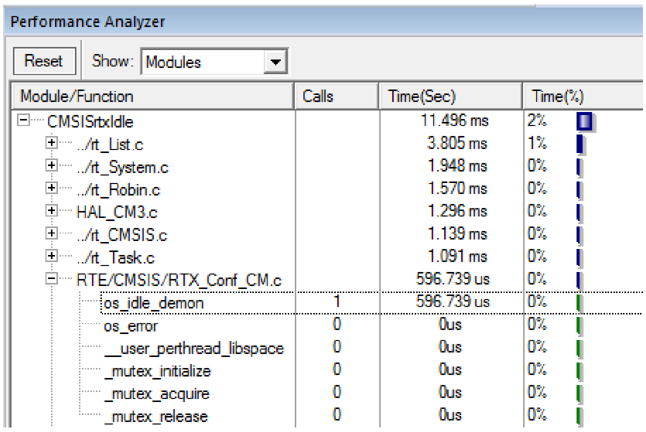
Exit the debugger

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

```
__NO_RETURN void osRtxIdleThread (void *argument){

for (;;) {

        __wfe();

}}
```

Rebuild the code, restart the debugger

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



Module/Function	Calls	Time(Sec)	Time(%)
CMSISrtxIdle		11.496 ms	2%
../rt_List.c		3.805 ms	1%
../rt_System.c		1.948 ms	0%
../rt_Robin.c		1.570 ms	0%
HAL_CM3.c		1.296 ms	0%
../rt_CMSIS.c		1.139 ms	0%
../rt_Task.c		1.091 ms	0%
RTE/CMSIS/RTX_Conf_CM.c		596.739 us	0%
os_idle_demon	1	596.739 us	0%
os_error	0	0us	0%
__user_perthread_libspace	0	0us	0%
_mutex_initialize	0	0us	0%
_mutex_acquire	0	0us	0%
_mutex_release	0	0us	0%

Fig 34 The __wfe() intrinsic halts the CPU when it enters the idle loop. Saving cycles and runtime energy

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.

Exercise 9 Thread Flags

In this exercise we will look at using thread flags 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 "Ex 9 Thread Flags" 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 thread flag to wake up the second led task.

```
void led_Thread2 (void *argument) {
 for (;;) {
         LED_On(2);
        oThreadFlagSet (T_led_ID1,0x01);
        osDelay(500);
        LED_Off(2);
        osThreadFlagSet (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 *argument) {
for (;;) {
        osThreadFlagsWait (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.

Exercise 10 Event Flags

In this exercise we will look at the configuration of an event Flag object and use it to synchronise the activity of several threads

In the Pack Installer select "Ex 10 Event Flags" and copy it to your tutorial directory.

Open main.c

The code in main.c creates and event flag object and instantiates it in appMain().

```
static const osEventFlagsAttr_t EventFlagAttr_LED = {

        .name = "LED_Events",

};

void app_main (void *argument)

{

        LED_Initialize();

        EventFlag_LED = osEventFlagsNew(&EventFlagAttr_LED);
```

The code then creates three threads. Two of the threads wait for an event flag to be set ;

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

for (;;) {

osEventFlagsWait (EventFlag_LED,0x01,osFlagsWaitAny,osWaitForever);

LED_On(1);


 _NO_RETURN void led_Thread2 (void *argument) {

for (;;)     {

        osEventFlagsWait (EventFlag_LED,0x01,osFlagsWaitAny,osWaitForever);

        LED_On(2);
```

The remaining thread is used to set the flag;

```
__NO_RETURN void led_Thread3 (void *argument) {

for (;;) {

        osEventFlagsSet      (EventFlag_LED,0x01);

        LED_On(3);
```

Build the code

Start the debugger and run the code

Observe the activity of the LED's

Why does the code not run as expected?

When the event flag is set one of the waiting threads will wake up and clear the flag. The second waiting thread is not triggered. Each thread should be waiting on a separate Event flag within the event flag object.

Change the code so that the waiting threads are waiting on separate flags and the remaining thread sets both flags

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

for (;;) {

osEventFlagsWait (EventFlag_LED,0x01,osFlagsWaitAny,osWaitForever);

LED_On(1);



 _NO_RETURN void led_Thread2 (void *argument) {

for (;;)      {

        osEventFlagsWait (EventFlag_LED,0x02,osFlagsWaitAny,osWaitForever);

        LED_On(2);
```

The remaining thread is used to set both flags;

```
__NO_RETURN void led_Thread3 (void *argument) {

for (;;) {

        osEventFlagsSet        (EventFlag_LED,0x03);

        LED_On(3);

    }

}
```

Exercise 11 Semaphore Signalling

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

In the Pack Installer select “Ex 11 Interrupt Signals” and copy it to your tutorial directory.

First, the code creates a semaphore called sem1 and initialises it with zero tokens and a maximum count o five tokens.

```
osSemaphoreId_t sem1;

static const osSemaphoreAttr_t semAttr_SEM1 = {
.name = "SEM1",
};

void app_main (void *argument) {

      sem1 = osSemaphoreNew(5, 0, &semAttr_SEM1 );
```

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

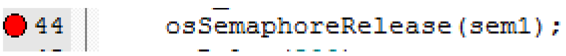
```
__NO_RETURN void led_Thread1 (void  *argument) {

for (;;) {
      osSemaphoreAcquire(sem1, osWaitForever);
      LED_On(1);
      osSemaphoreAcquire(sem1, osWaitForever);
      LED_Off(1);
}
```

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

```
__NO_RETURN void led_Thread2 (void *argument) {

for (;;) {
      osSemaphoreRelease(sem1);
      LED_On(2);
      osDelay(500);
      osSemaphoreRelease(sem1);
      LED_Off(2);
      osDelay(500);
}}
```

Build the project and start the debugger

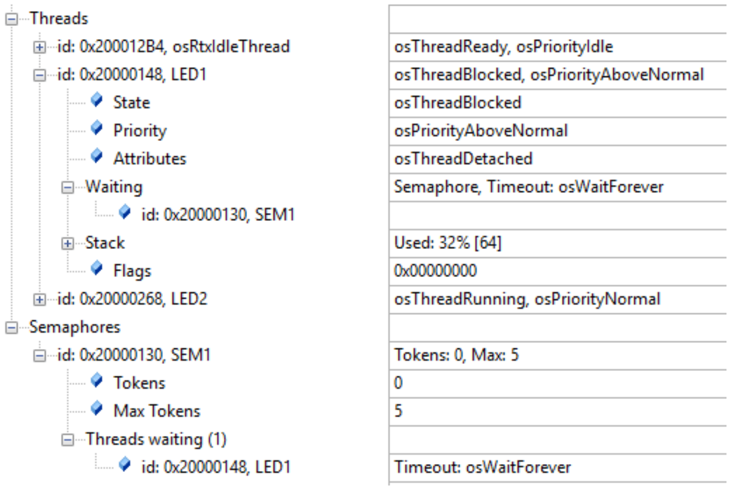
Set a breakpoint in the led_Thread2 task



```
osSemaphoreRelease(sem1);
```

Fig 46 Breakpoint on the semaphore release call in led_Thread2

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



Threads	
id: 0x200012B4, osRtxIdleThread	osThreadReady, osPriorityIdle
id: 0x20000148, LED1	osThreadBlocked, osPriorityAboveNormal
State	osThreadBlocked
Priority	osPriorityAboveNormal
Attributes	osThreadDetached
Waiting	Semaphore, Timeout: osWaitForever
id: 0x20000130, SEM1	
Stack	Used: 32% [64]
Flags	0x00000000
id: 0x20000268, LED2	osThreadRunning, osPriorityNormal
Semaphores	
id: 0x20000130, SEM1	Tokens: 0, Max: 5
Tokens	0
Max Tokens	5
Threads waiting (1)	
id: 0x20000148, LED1	Timeout: osWaitForever

Fig 47 Led_Thread1 is waiting to acquire a semaphore

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 osSemaphoreAcquire() call it will again block.

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

Exercise 12 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 "Ex 12 Multiplex" 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 = osSemaphoreNew(5, 3,&semAttr_Multiplex);
```

Build the code and start the debugger

Run the code and observe the GPIOB pins

Now three threads can access the led functions 'concurrently'.

Exercise 13 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 “Ex 13 Rendezvous” 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.

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’.

Exercise 14 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 62, 75, 80 and 93 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’.

Exercise 15 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 *argument) { uint32_t

i;

 for (;;) {

            //osMutexAcquire(uart_mutex, osWaitForever);

for( i=0;i<10;i++)    SendChar('1');

            SendChar('\n');

            SendChar('\r');

            //osMutexRelease(uart_mutex);

  }}
```

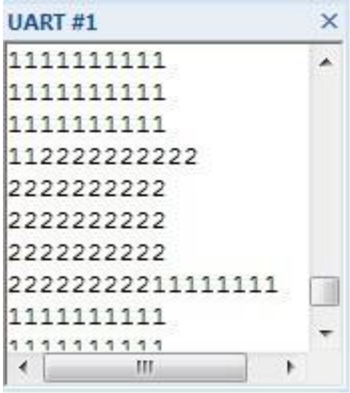
In each thread the code prints out the thread number. At the end of each block of characters it then prints the carriage return and new line characters.

Build the code and start the debugger.

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

Fig 48 Open the UART console window

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



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

Fig 50 The mis-ordered serial output

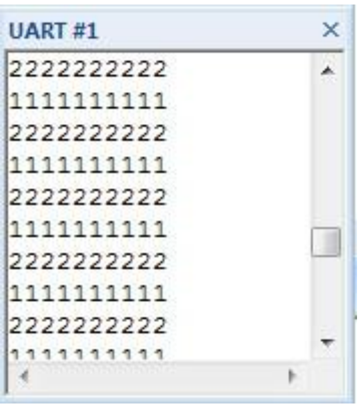
Here we can see that the output data stream is corrupted by each thread writing to the UART without any accessing control.

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.



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

Fig 49 Order restored by using a mutex

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

Exercise 16 Message queue

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

In the Pack Installer select “Ex 16 Message Queue” and copy it to your 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.

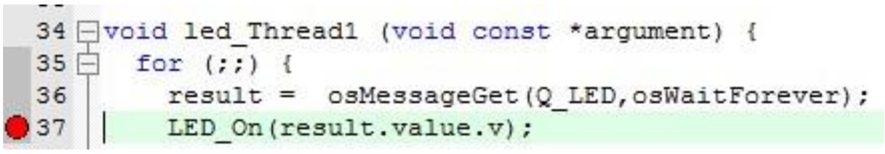
```
             osMessagePut(Q_LED,0x1,osWaitForever);

             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.



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

Fig 53 Set a breakpoint on the receiving thread

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

Exercise 17 Message queue

In the Pack Installer select “Ex 17 Message Queue” and copy it to your tutorial directory.

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

Led_Thread2 updates the message structure and posts a new message into the queue.

Led_Thread1 reads the queue and writes the transferred data to the LED.

Exercise 18 Zero Copy Mailbox

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

In the Pack Installer select “Ex 18 Memory Pool” 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.

```
static const osMemoryPoolAttr_t memorypoolAttr_mpool ={

		.name = "memory_pool",

};

void app_main (void *argument) {

   mpool = osMemoryPoolNew(16, sizeof(message_t),&memorypoolAttr_mpool );

   queue = osMessageQueueNew(16,4, NULL);

   osThreadNew(producer_thread, NULL, &ThreadAttr_producer);

   osThreadNew(consumer_thread, NULL, &ThreadAttr_consumer);

}
```

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

```
while (1){

		if(testData == 0xAA){

				testData = 0x55;

		}

		else{

				testData = 0xAA;

		}

		message = (message_t*)osMemoryPoolAlloc(mpool,osWaitForever); //Allocate a memory pool buffer

		for(index =0;index<8;index++){
```

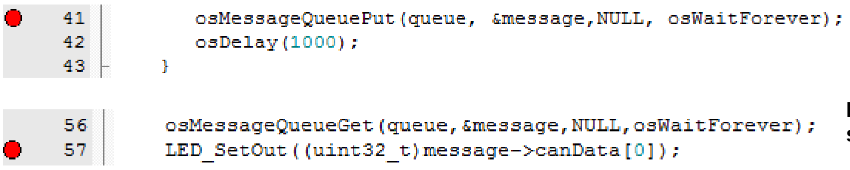
```
                message->canData[index] = testData;
        }

        osMessageQueuePut(queue, &message,NULL, osWaitForever);
        osDelay(1000);

  }
```

Then in the consumer thread we can read the message queue to get the next pointer 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.

```
while (1) {

        osMessageQueueGet(queue,&message,NULL,osWaitForever);

        LED_SetOut((uint32_t)message->canData[0]);
        osMemoryPoolFree(mpool, message);

}
```

Build the code and start the debugger.

Place breakpoints on the osMessagePut and osmessageGet functions.



```
        osMessageQueuePut(queue, &message,NULL, osWaitForever);
        osDelay(1000);
    }
```

```
    osMessageQueueGet(queue,&message,NULL,osWaitForever);
    LED_SetOut((uint32_t)message->canData[0]);
```

Fig 54 Set breakpoints on the sending and receiving threads

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