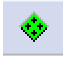
Exercise 2 Creating and managing threads

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 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.

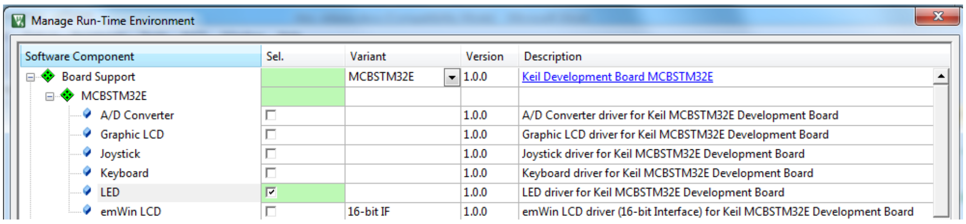


Fig 19 selecting the board support components

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.

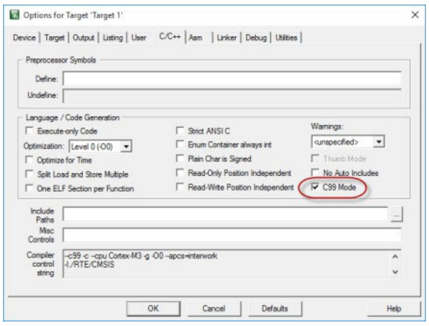
```
static const osThreadAttr_t threadAttr_app_main = {
    "app_main",
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    osPriorityNormal,
    NULL,
    NULL
};
```

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++



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

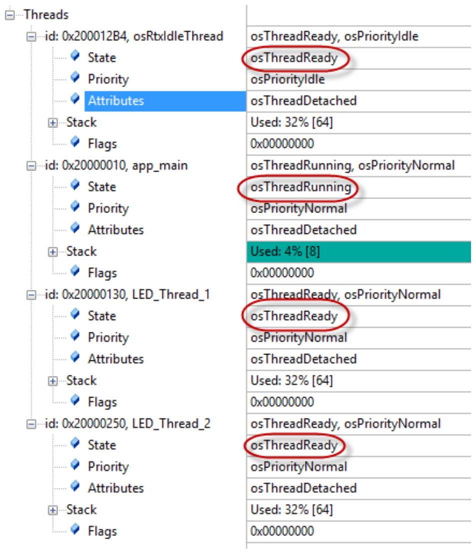


Fig 20 The running Threads

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

Now open the View → Analysis Windows → System Analyzer

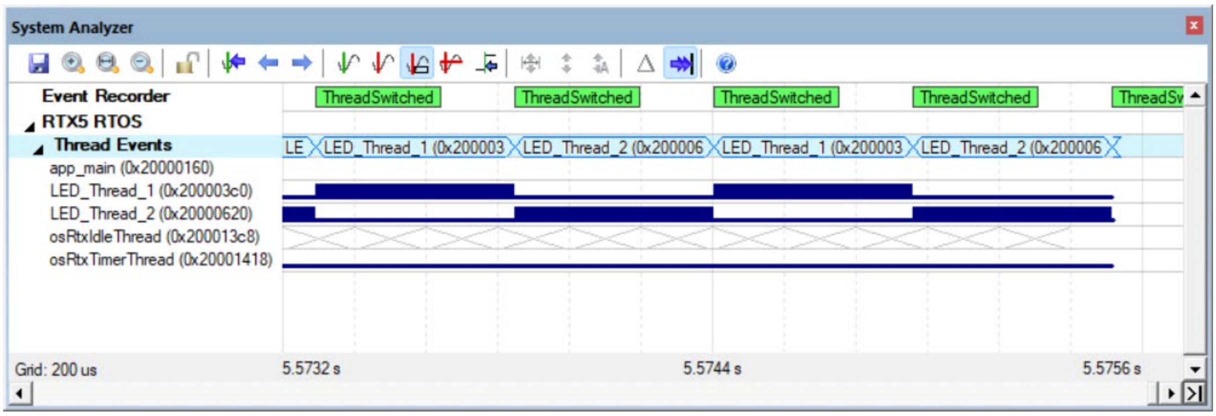


Fig 22 the event viewer shows the thread switching history.

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

Now open the Peripherals → General Purpose IO → GPIOB window

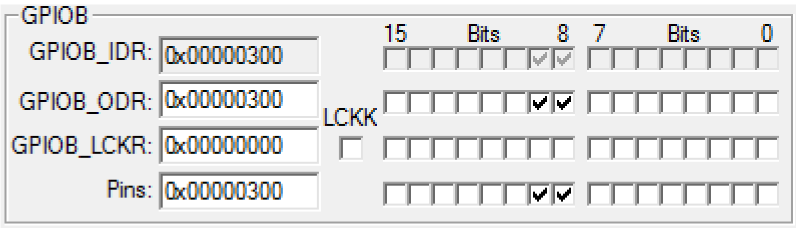


Fig 23 the peripheral window shows the LED pin activity

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.

Before continuing, make sure to read the section "Thread Management and Priority" in the RTOS2 tutorial.

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

First, change the priority of LED Thread2 to "Above Normal":

```
static const osThreadAttr_t ThreadAttr_LED1 = {
  .name = "LED_Thread_1",
  .priority = osPriorityNormal,
};

static const osThreadAttr_t ThreadAttr_LED2 = {
  .name = "LED_Thread_2",
  .priority = osPriorityAboveNormal,
};
```

Build the project and start the debugger.

Start the code running.

Open the View → Analysis Windows → System Analyzer.

Can you see which thread is running? It may be difficult to determine which thread is actually running...

Open the View → Analysis Windows → Event Recorder.

Try to explain the events you can see in the Event log. Which is the last event to be recorded? In order to determine what is going on, we can use a different approach.

Open the View → Analysis Windows → Performance Analyzer.

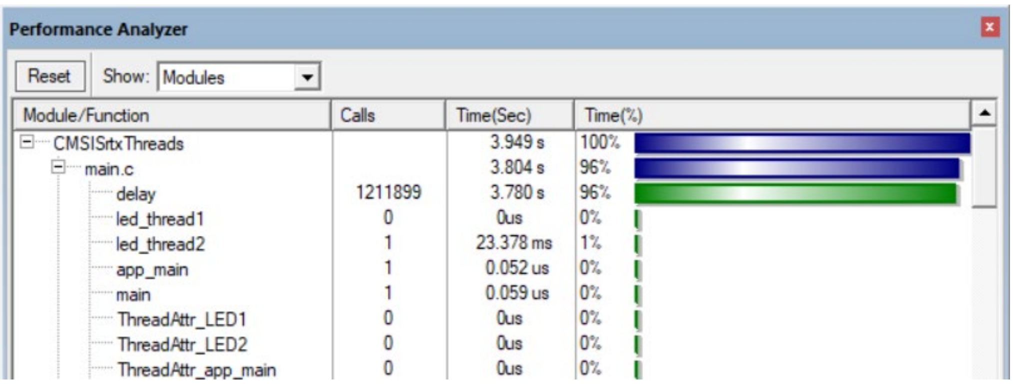


Fig 23 only led_thread2 is running.

Here we can see led_thread2 running but no sign of led_thread2. Looking at the coverage monitor for the two threads also shows us that led_thread1 has not run

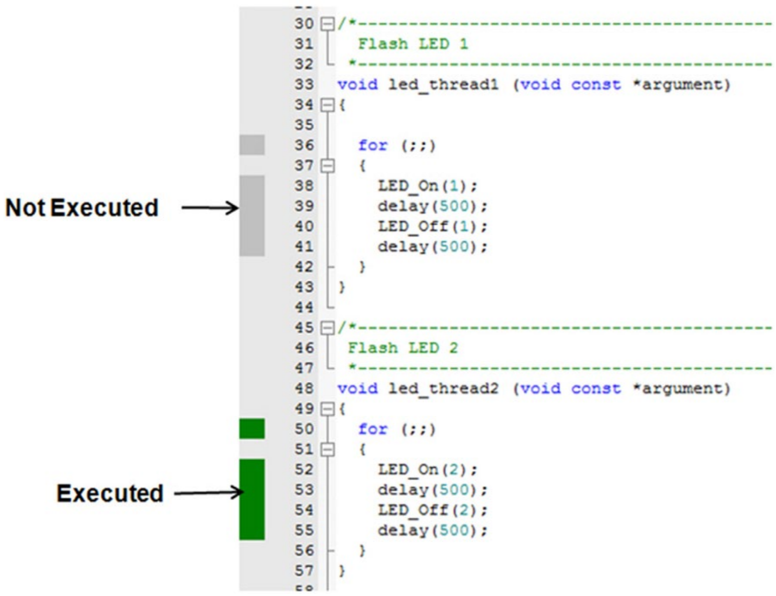


Fig 24 the coverage monitor shows what code has executed by coloring the margin green.

Led_thread1 is running at normal priority and led_thread2 is running at a higher priority so has *preempted* led_thread1. To make it even worse led_thread2 never yields so it will run forever, preventing the lower priority thread from ever running.

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