Real-Time Operating Systems (RTOS) Lab

This laboratory is not assessed

Aim

The aims for this RTOS lab are to.

1. Understand how a Real-Time Operating System works
2. Develop the skills to be able to design and implement a program using the CMSIS-RTOS RTX Real-Time Operating System.

Structure of the Lab sheet

This lab helps you develop the required skills to be able to use the RTOS. Additional tasks allow your understanding and ability to use functions and the RTOS to develop further. Solutions to the tasks are included in "RTOS Lab - Complete code and task solutions" document.

At the end of each task save the complete contents of you "main.c" file into a separate text file (copy and paste) so that you can access it easily at a later stage.

Date lab sheet is available from: Week 1 Monday 0900

Learning process

This lab sheet is generally self-contained. However, at times you will need to refer to other documents. You will be prompted when this is necessary and please take the effort to do so.

The purpose of the take home kit is to allow you to spend as much time as possible using the kit. It also allows you to learn independently and to explore the capabilities of the kit on your own. Please spend as much time as possible with the kit and work through the lab sheet in your own time. You might complete this lab without any problems and find you do not need much support in the structured lab session. Otherwise, you might need some support from the demonstrators in the structured lab session. Either way, if you attend the structured lab session you must be prepared with specific questions. The demonstrators will not sit down and take you through the laboratory step by step. **It is your responsibility to make a reasonable attempt at the laboratory before you seek assistance.**

This laboratory is not assessed. However, use of the techniques taught in this lab sheet will be required to achieve a good mark in the final assignment.

Support

If you have any problems with the equipment please contact Dr S A Pope immediately using the contact details provided below. **These inquiries should only relate to problems with the equipment, such as suspected faults.** Any inquiry should include a clear description of the problem. For example "It isn't working" would not be an acceptable inquiry on its own. **Any learning based inquiries must be brought to the timetabled support sessions (lectures, laboratories and demonstrations).**

Contact details

Dr S A Pope, Email: s.a.pope@sheffield.ac.uk, Internal telephone extension: 25186, Room: C07d AJB

Contents

Initialisation - Setup a project in Keil µVision

To start this lab you will first need to create a project in Keil µVision using the same process as you used in the introductory lab. Connect the STM32F4Discovery to the PC and follow steps 1 - 7 in the Introductory lab sheet (you should use a main clock speed of 168MHz). You will initially need the same components of the STM32F4Discovery that you used in the Introductory lab. So you should select the same components in the “Manage Run-Time Environment” window in step 3.

Using a Real-Time Operating System (RTOS)

Up until now we have placed all of our executable code in the main() function in a "super-loop". This is usually fine for simple embedded systems. However, more complex systems often require multiple tasks to be executed in response to different events in the embedded systems environment, or with a complex/changing sequence. Using a single infinite/super loop in the main() function isn't usually a very efficient way to implement a solution to such demands. The use of functions such as "goto" provide some control over the sequence of code execution, but they create "spaghetti" code that is difficult to thoroughly debug and doesn't create code that provides an acceptable level of robust performance. In general, the use of such functions should be avoided in embedded systems and the use of alternative approaches with dedicated functionality to meet real-time constraints in a robust manner should be considered.

A better approach is to define each of the tasks of the system as separate threads/tasks, which are effectively mini self-contained programs. A real-time operating system (RTOS) can then be used to determine which threads/tasks run at each particular time so as to meet any specified real-time constraints in a robust manner. Further details on RTOS can be found in Lecture 5 and the associated additional reading document "Priority Based Algorithms and Practical Issues with RTOS". A video demonstration taking you through the steps in each part of this lab is available on MOLE.

There are a number of RTOS that support the STM32F4 class of processors. The one that we will use, CMSIS-RTOS RTX, is supplied by Keil. It is relatively easy to use and is included as part of the installed CMSIS package for the STM32F4. The following examples will take you through a number of simple projects that demonstrate some of the important concepts of RTOS using the CMSIS-RTOS RTX. The principles and code are explained in each task, but you will find both the document "CMSIS_RTOS_Tutorial" provided in the pack and the information on the website below useful sources of reference, particularly if you develop you own projects using CMSIS-RTOS RTX.

http://www.keil.com/pack/doc/cmsis/rtos/html/index.html

Part 1 - A simple single thread program using the RTOS

We will now implement a simple single thread program using the CMSIS-RTOS RTX. Create a new project using the same process as before, but modify the steps according to the following points:

1. In the "Manage Run-Time Environment", in addition to all of the pervious components that we have used (including TIM and SPI), you will also need to select the "Keil RTX" which is found under the CMSIS->RTOS (API) expansion box.
2. We create the "my_headers.c" and "my_headers.h" as before, but we want to create the "main.c" file so that it uses the RTOS. We can do this by using the template "main.c" file that is available for the RTOS. In the "Add New Item to Group" window, instead of "C File", select "User Code Template" in the box on the left. Then in the CMSIS expansion in the window to the right select "CMSIS-RTOS 'main' function" and click on the "Add" button. If you open this "main.c" file you will see that it contains some pre-formatted code that sets up and uses the RTOS. It also creates an extra configuration file for the RTOS that we will use shortly.
3. We also need to add two more RTOS files. The first is a "Thread.c" file that contains all of the threads/tasks that will run in the RTOS. A template for this is again found in the "User Code Template" options in the "Add New Item to Group" window and listed as "CMSIS-RTOS Thread". After you have added this "Thread.c" file (which you will also see contains some pre-formatted code for defining threads), you will need to add a "Thread.h" header file in the same manner as has been used previously for "my_headers.h", i.e. a template isn't used.

Before we write any code we need to configure the settings of the RTOS. These can be found in the "RTX_conf_CM.c" file under the CMSIS expansion in the "Project" window on the left of the main Keil interface. If you open this file you will see typical code for configuration. However, at the bottom there is a tab called "Configuration Wizard". If you click on this and select "Expand All" in the window that opens, all of the available options will be displayed in a user interface. These options are editable and make configuring the RTOS straightforward. Each of these options are explained in the online CMSIS-RTOS RTX guide. We will only use a few of them during this lab. At this stage one important option needs to be changed. This is the "RTOS Kernel Timer input clock frequency [Hz] which needs to be set to the same value as the system clock, which we have set in the Introductory lab to 168MHz, i.e. 168000000Hz. Other options that might be important for future projects will be the maximum number of concurrent running threads (the default value is probably 6) and settings for the Round-Robin scheduling algorithm. After setting the clock frequency save and close this file.

In the "my_headers.h" and "my_headers.c" files create the same declarations and definitions for the `LED_initialise` and `Blink_LED` functions that were used in task B.1 of the Serial IO and Functions lab (provided in the "Serial IO and Functions Lab - Complete code and task solutions" document). You can omit both the declaration and definition for the delay function `delay_in_seconds`. We will use an alternative mechanism to implement the delay with the RTOS.

We now need to define the threads for our RTOS. The threads contain the main executable code for our program, i.e. the code which we previously placed in the infinite FOR loop. First we create the declarations for our threads in "Thread.h". The code that you need is provided below. The first line declares the operating system function that initialises our main thread which blinks the LED and the second line declares the main thread that blinks the LED.

```
extern int Init_Blink_LED_Thread (void); // Standard format to declare the
function to initialise the main thread function.
void Blink_LED_Thread (void const *argument); // Standard format to declare
the main thread function.
```

Next we will write the code for our thread in the “Thread.c” file. In this file you will see a template for a sample thread. Each time you define a thread you can use this format. The code below modifies this template to create the thread to blink the LED, i.e. `Blink_LED_Thread`. The detailed comments in the code below describe the purpose of each line in more detail. Note that you will need to add additional header files for any externally defined parameters or functions that you will use. In this case, in addition to the RTOS header, this is the standard system header and the header file for the functions that we have defined. Note that the template might enclose the header file names in < > instead of “ “, but the operation of these two different approaches is the same.

```
#include "cmsis_os.h"                               // CMSIS RTOS header file
#include "stm32f4xx.h"
#include "my_headers.h"

/*----------------------------------------------------------------------------
 *      Blink LED Thread
 *---------------------------------------------------------------------------*/

void Blink_LED_Thread (void const *argument); // Standard format to declare
the main thread function that is defined later in the code
osThreadId tid_Blink_LED_Thread; // Declares an ID that we will associate
with the thread and which allows easy reference to it when using some of
the OS functions.
osThreadDef (Blink_LED_Thread, osPriorityNormal, 1, 0); // Declares the
main thread object that we will use later. The parameters can be used to
adjust certain properties, such as the priority of a thread and how many
instances of it exist.

// Code to define the thread function to initialise the main thread - this
initialise function is called from the “main.c” file to start the thread.
int Init_Blink_LED_Thread (void) {

  tid_Blink_LED_Thread = osThreadCreate (osThread(Blink_LED_Thread), NULL);
      // Creates the main thread object that we have declared and assigns
      it the thread ID that we have declared.
  if(!tid_Blink_LED_Thread) return(-1); // Checks to make sure the thread
      has been created.

  return(0);
}

// Code to define the operation of the main thread. This is effectively the
code that was in the infinite FOR loop of our previous blinky program.
void Blink_LED_Thread (void const *argument) {


      uint8_t LED_on = 1; // Defines parameter for LED on
      uint8_t LED_off = 0; // Defines parameter for LED off

  while (1) { // Creates an infinite loop so that the blinking never
            terminates

            Blink_LED(LED_on); // Blinks the green LED on once

            osDelay(1000); // Uses the built in delay function for the OS
            to create a 1 second delay. The fundamental delay is specified
            in the “RTX_conf_CM.c” file and usually defaults to 1ms.
            Blink_LED(LED_off); // Blinks the green LED on once

            osDelay(1000); // Uses the built in delay function for the OS
            to create a 1 second delay. The fundamental delay is specified
            in the "RTX_conf_CM.c" file and usually defaults to 1ms.

            osThreadYield(); // This function tells the RTOS that when the
            thread gets to this stage the RTOS should suspend this thread
            and run the next thread that is ready to run. If there is no
            other thread ready (which is the case with this simple program
            since we only have one thread) then the calling thread
            continues. This function effectively forces the RTOS to
            reschedule and is useful in more complex systems and scheduling
            policies.

  }

}
```

The final stage is to modify the "main.c" file template that we created. The required modified version is provided below. Note that the standard system header, the user function header and the thread header file have been included at the start.

```
/*----------------------------------------------------------------------------
 * CMSIS-RTOS 'main' function - Blinky
 *---------------------------------------------------------------------------*/

#define osObjectsPublic                     // Define objects in main module
#include "osObjects.h"                      // RTOS object definitions
#include "stm32f4xx.h"
#include "my_headers.h"
#include "Thread.h"

int main (void) {

       osKernelInitialize (); //Initialize CMSIS-RTOS


      /* Initialise any peripherals or system components */

      // Initialize the LED
      Initialise_LED();

      /* Initialise any threads */

      // Initialise the main thread to blink the LED's
      Init_Blink_LED_Thread();



      osKernelStart (); // start thread execution

      while(1){}; // While loop so the program doesn't terminate
}
```

When complete compile and download your program to test/observe its operation. It should operate in the same manner as the simple blinky program developed in previous labs. The complete list of code for all of the files is provided in the “RTOS Lab - Complete Code and Task Solutions” document.

Part 2 - Implementing a multiple thread project using an RTOS

In the previous program we implemented a single thread and thus in reality its functionality is the same as using an infinite FOR or WHILE loop embedded in the main() function. One of the key advantages of RTOS is their ability to deal with multiple distinct tasks/threads. We will now modify the simple program so that instead of blinking a single LED on and off, in addition it turns the red LED on when the user push-button is pressed and held. For this we will have two tasks/threads. One will blink the green LED and the other will monitor the state of the button and turn the red LED on/off accordingly. To implement this as a single piece of code in main(), we would need to use a timer running in the background and use a number of IF statements to check the state of the switch and timer. However, with the RTOS we can just create another independent thread and set it running.

To implement this multithread program you can modify your code from part 1. First we need to add some code to initialise the red LED and enable the clock of the port which the push-button switch is connected to (port A). We can add this to the `Initialise_LED` function defined in "my_headers.c". In addition we will rename this function `Initialise_LED_and_button`. You will also need to rename all other instances of this function in your program. These can be found in the "my_headers.h" and "main.c" files. The new function definition is provided below.

```c
// Definition for the function to initialise the LED and button
void Initialise_LED_and_button(void){

	// Initialize GPIO Port for LEDs
	RCC->AHB1ENR |= RCC_AHB1ENR_GPIODEN; // Enable Port D clock
	GPIOD->MODER |= GPIO_MODER_MODER12_0; // Port D.12 output - green LED
	GPIOD->MODER |= GPIO_MODER_MODER14_0; // Port D.14 output - red LED

	//Initialize GPIO for push-button
	RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN; // Enable Port A clock

}
```

Next we need to define a function to turn the red LED on and off. The function declaration followed by the function definition that you need to add to your "my_headers.h" and "my_headers.c" are provided below.

Function declaration:

```
void Red_LED(uint8_t); // Declaration for the function to turn the red LED
on/off.
```

Function definition:

```
// Definition for the function to blink the LED
void Red_LED(uint8_t LED_state){

      if(LED_state == 1){ // Checks to see if the request is to turn the
            LED on or off

                  GPIOD->BSRR = 1<<14; // Turn on the red LED

      }
      else{

                  GPIOD->BSRR = 1<<(14+16); // Turn off the red LED

      }

}
```

Next we need to declare and define our threads for the push-button. We can use the same templates as we used for the thread to blink the LED. The thread declarations and definitions that need to be added to your "Thread.h" and "Thread.c" files are provided below.

Thread declarations:

```
extern int Init_Button_Thread (void); // Standard format to declare the
function to initialise the main thread function.
void Button_Thread (void const *argument); // Standard format to declare
the main thread function.
```

Thread definitions:

You will need to place all of the declarations for all of the defined threads (the parts highlighted in blue) near the top of this file and all of the definitions after these declarations. If you are unsure please refer to the complete code in the "RTOS Lab - Complete code and task solutions" document. If you don't do this then there can be problems communicating between tasks, which is something that is covered in part 3.

```
void Button_Thread (void const *argument); // Declares the main thread
function that is defined later in the code
osThreadId tid_Button_Thread; // Declares an ID that we will associate with
the thread and which allows easy reference to it when using some of the OS
functions.
osThreadDef (Button_Thread, osPriorityNormal, 1, 0); // Declares the main
thread object that we will use later. The parameters can be used to adjust
certain properties, such as the priority of a thread and how many instances
of it exist.
/*----------------------------------------------------------------------------
 *      Red LED on when button pressed thread
 *---------------------------------------------------------------------------*/

// Code to define the thread function to initialise the main thread - this
initialise function is called from the "main.c" file to start the thread.
int Init_Button_Thread (void) {

  tid_Button_Thread = osThreadCreate (osThread(Button_Thread), NULL); //
Creates the main thread object that we have declared and assigns it the
thread ID that we have declared.
  if(!tid_Button_Thread) return(-1); // Checks to make sure the thread has
been created.

  return(0);
}

// Code to define the operation of the main thread.
void Button_Thread (void const *argument) {

      uint8_t LED_on = 1; // Defines parameter for LED on
      uint8_t LED_off = 0; // Defines parameter for LED off

  while (1) { // Creates an infinite loop so that the blinking never
            terminates
            // Checks the state of the push-button and only turns the red
            LED on if the button has only just been pressed, which is
            indicated by the state of the red LED.
            if(((GPIOA->IDR & 0x00000001) == 0x00000001) & ((GPIOD->ODR &
            (1<<14)) != (1<<14))){

                  Red_LED(LED_on); // Turn red LED on

            }
            // Checks the state of the push-button and only turns the red
            LED off if the button has only just been released, which is
            indicated by the state of the red LED.
            else if(((GPIOA->IDR & 0x00000001) != 0x00000001) & ((GPIOD-
            >ODR & (1<<14)) == (1<<14))){

                  Red_LED(LED_off); // Turn red LED off

            }

            osThreadYield(); // This function tells the RTOS that when the
            thread gets to this stage the RTOS should suspend this thread
            and run the next thread that is ready to run. If there is no
            other thread ready (which is the case with this simple program
            since we only have one thread) then the calling thread
            continues. This function effectively forces the RTOS to
            reschedule and is useful in more complex systems and scheduling
            policies.
     }
}
```

Finally, we just need to initialise our new thread in “main.c”. As before this is done using the initialise thread function that we have defined and we place `Init_Button_Thread()` immediately after `Init_Blink_LED_Thread()` in main().

When complete, compile and download your program to test/observe its operation. The green LED should blink on/off at regular 1 second intervals. When the blue user push-button is pressed the red LED should come and stay on until the button is released. While the button is pressed and the red LED on, the green LED should continue to blink. This illustrates the flexible approach that RTOS present to implementing solutions with distinct independent tasks.

The complete list of code for all of the files is provided in the “RTOS Lab - Complete code and task solutions” document.

Task 1 - Calling the same function in different threads

For this multi-thread program we wrote a separate function to turn on/off the red LED, i.e. `Red_LED`. This was in addition to the existing function `Blink_LED` to turn the green LED on/off. Modify this program so that it uses a single blink function where, in addition to its on/off state, it has an additional input argument that specifies which LED to turn on/off.

When complete, compile and download your program to test/observe its operation.

A solution to this task is provided in the “RTOS Lab - Complete code and task solutions” document.

IMPORTANT: Note the use of **`extern`** for the output data type in the thread declaration for the thread initialisation functions, for example `Init_Button_Thread.` These functions are called outside of the main Thread.c file, i.e. in the main.c file. The use of the **`extern`** data type makes this possible by declaring the output data argument as global as opposed to local. This indicates to the compiler that this output will be used in different files so that it can take this into consideration when compiling the code.

When you develop more complex programmes with multiple files you will need to consider the use of local and global variables for your user defined variables. For example, if you have variables defined in “my_headers.c” that you want to use in different Threads or functions in “Thread.c” or “main.c”, then you will need to use **`extern`** to define them as global variables so that the compiler knows that they will be used in multiple files.

Part 3 - Inter-thread communication

In the previous section we created a program in which two independent tasks ran concurrently. The CMSIS-RTOS RTX implements a Round-Robin scheduling process whereby each thread/task is run in sequence for a fixed amount of time before the RTOS switches to the next thread/task (refer to Lecture 4 for further details). With two tasks this means that the RTOS is constantly alternating between the two tasks and each one is run for a fixed amount of time in its slot. The amount of time that each task runs for is specified in the "RTX_Conf_CM.c" file that we viewed earlier. Checking this file again we see that the Round-Robin Timeout is set to 5 clock ticks, i.e. each task runs for 5 clock ticks before switching to the next task. By using the provided OS delay function `osDelay` the operating system can implement an accurate delay even when a task is not running.

It is common for many embedded systems to have tasks/threads that are not independent (see the examples in Lecture 5 for further details). For example, we might want to pause a task/thread, when another task/thread reaches a particular point. We will now look at such an example of task/thread dependence. We will modify our previous multi-thread program so that when the user button is pressed the task that blinks the green LED pauses and then resumes when the button is released. The starting point will be the code developed for the previous task 1 in part 2, which is provided in the "RTOS Lab - Complete code and task solutions" document.

There are several solutions to implement task synchronisation, such as Semaphore which is discussed in Lecture 5. CMSIS-RTOS RTX supports semaphore, but it also includes specific signalling functions to efficiently implement task synchronisation (see the online guide for further details http://www.keil.com/pack/doc/cmsis/rtos/html/group__CMSIS__RTOS__SignalMgmt.html). It is this method that we will use. With this method specific flags associated with each of the tasks can be set/cleared and functions can be required to wait for flags to be set. We will need to use all three functions that are available. The operation of these are summarised below.

`osSignalWait(int32_t signals, uint32_t millisec)`
When a thread calls this function the thread is suspending until the appropriate flag is set.

`osSignalSet(osThreadId thread_id, int32_t signals)`
When this function is called the specified flag is set for the thread specified on the input argument of this function.

`osSignalClear(osThreadId thread_id, int32_t signals)`
When this function is called the specified flag is cleared for the thread specified on the input argument of this function.

To specify which threads flag we want to set/clear we use the thread ID's that we have previously defined. The code below is a modified version of "Thread.c" that uses these signal functions. `osSignalSet` is used before the main while loop of `Blink_LED_Thread` to initialise the required flag by setting it on. By using `osSignalWait` the blink LED thread is suspended for the specified amount of time (here it is forever) or until the flag is set. When it completes its wait the flag is cleared, so `osSignalSet` needs to be called to reset the flag, otherwise the function will be suspend at the next instance of `osSignalWait`. We can now suspend/restart `Blink_LED_Thread` from `Button_Thread` by setting and clearing the flag associated with `Blink_LED_Thread`. We suspend it by using `osSignalClear` when the button is pressed and re-start it using `osSignalSet` when the button is released. The 0x01 argument in each of these calls specifies which flag (each thread has multiple flags) that we are using.

```
#include "cmsis_os.h"                          // CMSIS RTOS header file
#include "stm32f4xx.h"
#include "my_headers.h"

// Thread Declarations
void Blink_LED_Thread (void const *argument); // Standard format to declare
the main thread function that is defined later in the code
osThreadId tid_Blink_LED_Thread; // Declares an ID that we will associate
with the thread and which allows easy reference to it when using some of
the OS functions.
osThreadDef (Blink_LED_Thread, osPriorityNormal, 1, 0); // Declares the
main thread object that we will use later. The parameters can be used to
adjust certain properties, such as the priority of a thread and how many
instances of it exist.

void Button_Thread (void const *argument); // Declares the main thread
function that is defined later in the code
osThreadId tid_Button_Thread; // Declares an ID that we will associate with
the thread and which allows easy reference to it when using some of the OS
functions.
osThreadDef (Button_Thread, osPriorityNormal, 1, 0); // Declares the main
thread object that we will use later. The parameters can be used to adjust
certain properties, such as the priority of a thread and how many instances
of it exist.

/*----------------------------------------------------------------------------
 *      Blink LED Thread
 *---------------------------------------------------------------------------*/

// Code to define the thread function to initialise the main thread - this
initialise function is called from the “main.c” file to start the thread.
int Init_Blink_LED_Thread (void) {

  tid_Blink_LED_Thread = osThreadCreate (osThread(Blink_LED_Thread), NULL);
      // Creates the main thread object that we have declared and assigns
      it the thread ID that we have declared.
  if(!tid_Blink_LED_Thread) return(-1); // Checks to make sure the thread
      has been created.

  return(0);
}

// Code to define the operation of the main thread. This is effectively the
code that was in the infinite FOR loop of our previous blinky program.
void Blink_LED_Thread (void const *argument) {


      uint8_t LED_on = 1; // Defines parameter for LED on
      uint8_t LED_off = 0; // Defines parameter for LED off
      uint8_t green_LED = 12; // Defines parameter for green LED (GPIOD pin
                              12)

      osSignalSet(tid_Blink_LED_Thread,0x01);// Set flag 0x01 of the blink
      LED thread so that it resumes next time wait is called

  while (1) { // Creates an infinite loop so that the blinking never
            terminates
            osSignalWait(0x01,osWaitForever); // Waits until flag 0x01 of
            this thread is set
            osSignalSet(tid_Blink_LED_Thread,0x01);// Set flag 0x01 of the
            blink LED thread so that it resumes next time wait is called

            Blink_LED(LED_on,green_LED); // Blinks the green LED on once

            osDelay(1000); // Uses the built in delay function for the OS
            to create a 1 second delay. The fundamental delay is specified
            in the "RTX_conf_CM.c" file and usually defaults to 1ms.

            osSignalWait(0x01,osWaitForever); // Waits until flag 0x01 of
            this thread is set
            osSignalSet(tid_Blink_LED_Thread,0x01);// Set flag 0x01 of the
            blink LED thread so that it resumes next time wait is called

            Blink_LED(LED_off,green_LED); // Blinks the green LED on once

            osDelay(1000); // Uses the built in delay function for the OS
            to create a 1 second delay. The fundamental delay is specified
            in the "RTX_conf_CM.c" file and usually defaults to 1ms.

            osThreadYield(); // This function tells the RTOS that when the
            thread gets to this stage the RTOS should suspend this thread
            and run the next thread that is ready to run. If there is no
            other thread ready (which is the case with this simple program
            since we only have one thread) then the calling thread
            continues. This function effectively forces the RTOS to
            reschedule and is useful in more complex systems and scheduling
            policies.

  }
}

/*----------------------------------------------------------------------------
 *      Red LED on when button pressed thread
 *---------------------------------------------------------------------------*/

// Code to define the thread function to initialise the main thread - this
initialise function is called from the "main.c" file to start the thread.
int Init_Button_Thread (void) {

  tid_Button_Thread = osThreadCreate (osThread(Button_Thread), NULL); //
Creates the main thread object that we have declared and assigns it the
thread ID that we have declared.
  if(!tid_Button_Thread) return(-1); // Checks to make sure the thread has
been created.

  return(0);
}

// Code to define the operation of the main thread.
void Button_Thread (void const *argument) {

      uint8_t LED_on = 1; // Defines parameter for LED on
      uint8_t LED_off = 0; // Defines parameter for LED off
```

```
    uint8_t red_LED = 14; // Defines parameter for red LED (GPIOD pin 14)

  while (1) { // Creates an infinite loop so that the blinking never
              terminates

              // Checks the state of the push-button and only turns the red
              LED on if the button has only just been pressed, which is
              indicated by the state of the red LED.
              if(((GPIOA->IDR & 0x00000001) == 0x00000001) & ((GPIOD->ODR &
              (1<<14)) != (1<<14))){

                    osSignalClear(tid_Blink_LED_Thread,0x01); //Clear flag
                    0x01 of the blink LED thread so that it resumes

                    Blink_LED(LED_on,red_LED);// Turn red LED on

              }
              // Checks the state of the push-button and only turns the red
              LED off if the button has only just been released, which is
              indicated by the state of the red LED.
              else if(((GPIOA->IDR & 0x00000001) != 0x00000001) & ((GPIOD-
              >ODR & (1<<14)) == (1<<14))){

                    Blink_LED(LED_off,red_LED); // Turn red LED off

                    osSignalSet(tid_Blink_LED_Thread,0x01); // Set flag 0x01
                    of the blink LED thread so that it resumes


              }

              osThreadYield(); // This function tells the RTOS that when the
              thread gets to this stage the RTOS should suspend this thread
              and run the next thread that is ready to run. If there is no
              other thread ready (which is the case with this simple program
              since we only have one thread) then the calling thread
              continues. This function effectively forces the RTOS to
              reschedule and is useful in more complex systems and scheduling
              policies.

  }

}
```

Replace the "Thread.c" from part 2 with this new "Thread.c" (the other files remain the same) and when complete, compile and download your program to test/observe its operation. The green LED should flash until the button is pressed, at which the red LED comes on and the green LED stays in the same state it was in when the button was pressed.

The complete list of code for all of the files is provided in the "RTOS Lab - Complete code and task solutions" document.

Task 2 - Better timings

This program isn't perfect since when the blink LED task is suspended the osDelay function continues. You can observe this if you hold the button down for greater than one second and then release it. You should see the green LED change state straightaway since the 1 second timer has already expired. If we wanted to suspend the delay we would need to use an alternative approach. For example, we could use a timer that in addition to checking the flag of the timer checks the wait thread flag during a while loop.

Based on this knowledge, modify this program so that instead of using osDelay, you use a timer and a WHILE loop that checks the status of the timer flag and which also allows the timer to be suspended.

When complete, compile and download your program to test/observe its operation.

A solution to this task is provided in the "RTOS Lab - Complete code and task solutions" document.

Task 3 - A multi-thread program to blink two different LED's

In part 1 we developed a single-thread program to blink a single LED. Subsequently in parts 2 and 3 we covered multi-thread programs and inter-task communication. Your task now is to develop your own multi-thread program which should have the following specification:

It should consist of two threads. One thread should turn the green LED on and off. A second thread should do the same for the red LED. These two threads should be synchronised so that the green and red LED's alternately come on for 0.5 seconds, i.e. red on and green off for 0.5s, then red off and green on for 0.5s, red on and green off for 0.5s, etc..

When complete, compile and download your program to test/observe its operation.

A solution to this task is provided in the "RTOS Lab - Complete code and task solutions" document.

Final note - priorities

This is a brief practical introduction to some of the basic, but most important, concepts of an RTOS. However, we have used a non-priority driven scheduling procedure (all priorities were assigned the same value). It is possible to use the Round-Robin scheduling procedure in CMSIS-RTOS RTX with priorities. More information on this is available in the "CMSIS RTOS tutorial" and on the reference website. However, care must be taken to ensure that your threads run as desired if you do this, since a ready to run higher priority task will always block a lower priority task with the CMSIS-RTOS RTX.