Sheffield Hallam University

Department of Engineering

BEng (Hons) Computer and Network Engineering

Beng (Hons) Electrical and Electronic Engineering

Activity ID		Activity Title			Laboratory Room No.	Level
Lab 103		Using a real-time operating system			4302	6
Term	**Duration [hrs]**	**Group Size**	**Max Total Students**	**Date of approval/review**	**Lead Academic**	
1	6	2	25	09-18	Alex Shenfield	

Equipment (per student/group)

Number	Item
1	STM32F7 discovery board lab kit

Learning Outcomes

	Learning Outcome
2	Demonstrate an understanding of the various tools, technologies and protocols used in the development of embedded systems with network functionality
3	Design, implement and test embedded networked devices

Using a real-time operating system with the STM32F7 discovery board

Introduction

Computing is about more than the PC on your desktop! Embedded devices are everywhere – from wireless telecommunications infrastructure points to electronic point of sale terminals. One definition of an embedded system is:

> "An embedded system is a computer system designed to perform one or a few dedicated functions often with real-time computing constraints."
>
> (http://en.wikipedia.org/wiki/Embedded_system)

In the laboratory sessions for this module you are going to be introduced to the STM32F7 discovery board – a powerful ARM Cortex M7 based microcontroller platform capable of prototyping advanced embedded systems designs. The STM32F7 discovery board includes advanced functionality such as Ethernet connectivity, UART over the USB connection, an LCD screen, and a micro-SD slot. Appendix A shows the various pins that are broken out from the STM32F7 discovery board (onto the Arduino form factor header), and Appendix B provides a schematic showing how these map to the headers on the SHU base board.

This laboratory session will focus on the basics of using a real-time operating system with the STM32F7 discovery board. We will use the CMSIS-RTOS abstraction layer to provide an operating system agnostic interface to the RTOS functions (as discussed in the first set of lectures). Remember, one of the key benefits to using a real-time operating system with an embedded system is the ability to create easily extensible applications (and thus improve productivity).

Bibliography

There are no essential bibliographic resources for this laboratory session aside from this tutorial sheet. However the following websites and tutorials may be of help (especially if you haven't done much electronics previously or your digital logic and/or programming is a bit rusty):

- http://www.cs.indiana.edu/~geobrown/book.pdf[1]
- https://visualgdb.com/tutorials/arm/stm32/
- http://www.keil.com/appnotes/files/apnt_280.pdf
- https://developer.mbed.org/platforms/ST-Discovery-F746NG/

1 Note: this book is for a slightly different board – however, much of the material is relevant to the STM32F7 discovery

Methodology

Check that you have all the necessary equipment (see Figure 1)!

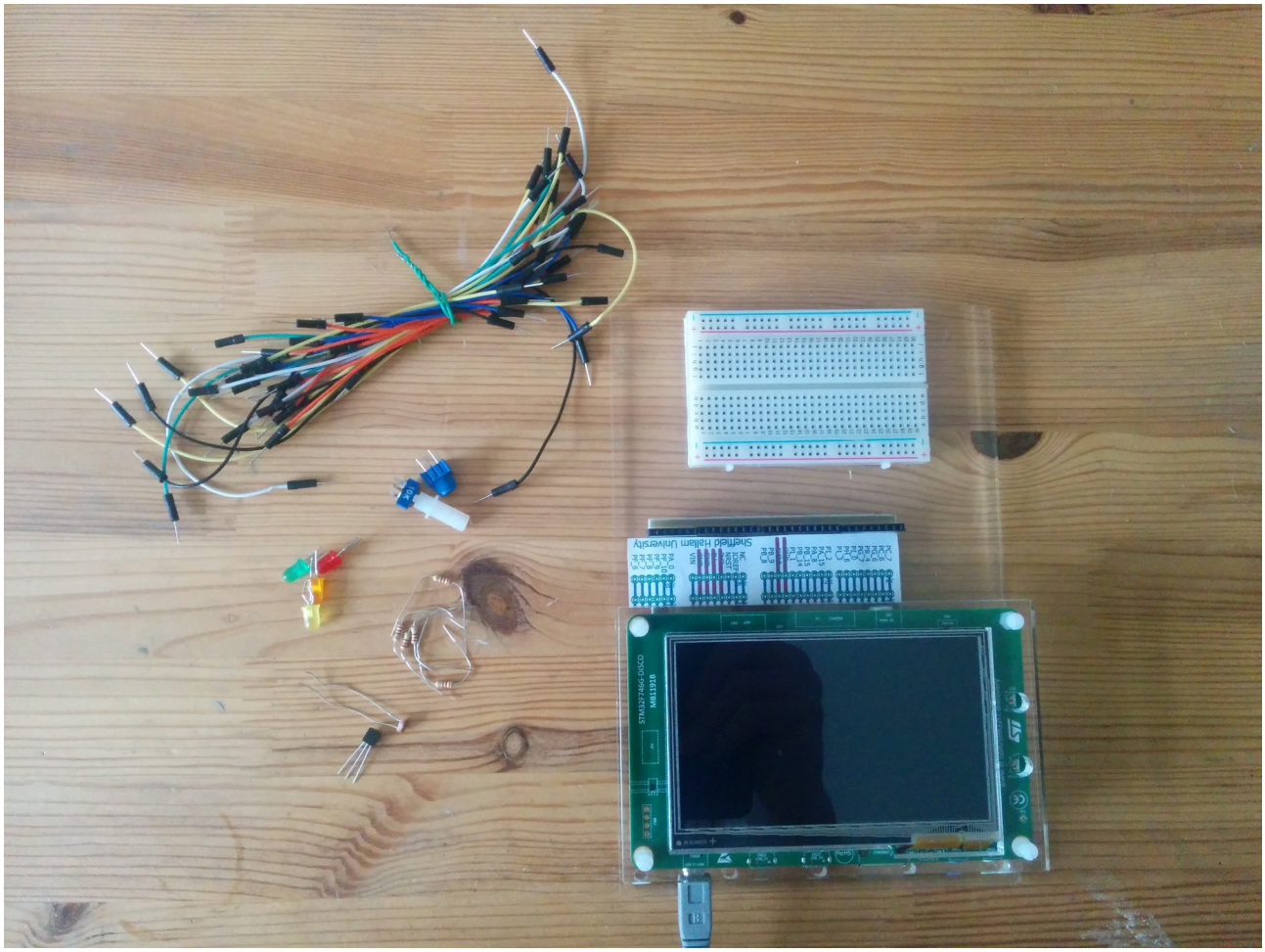


Figure 1 – The necessary equipment for this lab

Task 1

The first thing we did in these lab sessions was to blink an LED connected to the STM32F7 discovery boards – this is the embedded computing equivalent of the ubiquitous “Hello World!” program! We are now going to revisit this example using threads and a real-time operating system. For this example, we are going to connect LEDs to PI_1, PB_14, and PB15 (via 220 Ω current limiting resistors) – see Figure 2.

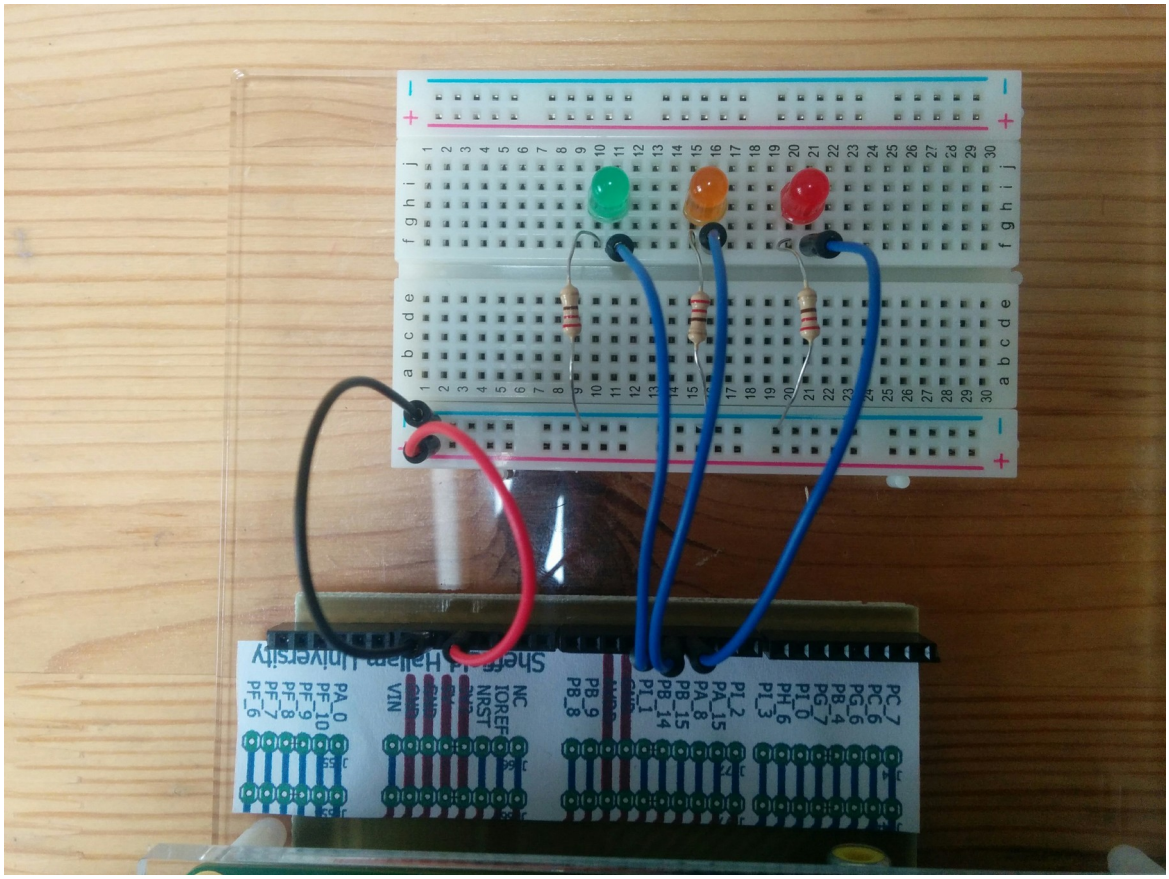


Figure 2 – The STM32F7 discovery board with connected LEDs

Figure 3 shows the logic for the RTOS based blinky program in pseudo-code.

```
Procedure RTOS-BLINKY:

  while not ready:
        initialise LEDs
  end while

  create led thread 1
  create led thread 2
  create led thread 3

  pass control to RTOS scheduler

end Procedure RTOS-BLINKY
```

Figure 3 – Pseudo code for the “blinky” program

We could implement all this logic in a single **`main.c`** file. However, one of the key advantages of using threads in a real-time operating system is that we can write much more modular (and reusable) code. To use threads in this way we will write a broad and easily reusable **`main.c`** file (see code listing 1) and then implement our threads in a separate file (called **`blinky_thread.c`** in this example – see code listing 2).

By designing our application in this fashion we ensure the responsibilities for each aspect of the system are easily separable.

Code listing 1:

```
/*
 * main.c
 *
 * this is the main rtos blinky application
 *
 * author:    Dr. Alex Shenfield
 * date:      01/09/2017
 * purpose:   55-604481 embedded computer networks : lab 103
 */

// include the basic headers for the hal drivers
#include "stm32f7xx_hal.h"
#include "cmsis_os.h"

// include the clock configuration file from the shu bsp library
#include "clock.h"

// declare the extern method that sets everything up for us
extern int init_thread(void);

// this is the main method
int main()
{
  // initialise the real time kernel
  osKernelInitialize();

  // we need to initialise the hal library and set up the SystemCoreClock
  // properly
  HAL_Init();
  init_sysclk_216MHz();

  // at this stage the microcontroller clock setting is already configured.
  // this is done through SystemInit() function which is called from the
  // startup file (startup_stm32f7xx.s) before to branch to application main.
  // to reconfigure the default setting of SystemInit() function, refer to
  // system_stm32f7xx.c file
  //
  // note also that we need to set the correct core clock in the RTX_Conf_CM.c
  // file (OS_CLOCK) which we can do using the configuration wizard

  // initialise our threads
  init_thread();

  // start everything running
  osKernelStart();
}
```

Code listing 2:

```
/*
 * blinky_thread.c
 *
 * simple thread to handle blinking some leds on the discovery board - updated
 * for the stm32f7xx hal libraries
 *
 * author:    Dr. Alex Shenfield
 * date:      01/09/2017
 * purpose:   55-604481 embedded computer networks : lab 103
 */

// include the basic headers for the hal drivers
#include "stm32f7xx_hal.h"
#include "cmsis_os.h"

// include the shu bsp libraries for the stm32f7 discovery board
#include "clock.h"
#include "gpio.h"
#include "pinmappings.h"

// HARDWARE DEFINES

// specify some leds
gpio_pin_t led1 = {PF_6, GPIOF, GPIO_PIN_6};
gpio_pin_t led2 = {PF_7, GPIOF, GPIO_PIN_7};
gpio_pin_t led3 = {PF_8, GPIOF, GPIO_PIN_8};

// RTOS DEFINES

// declare the thread function prototypes
void led_1_thread(void const *argument);
void led_2_thread(void const *argument);
void led_3_thread(void const *argument);

// declare the thread ids
osThreadId tid_led_1_thread;
osThreadId tid_led_2_thread;
osThreadId tid_led_3_thread;

// define thread priorities
osThreadDef(led_1_thread, osPriorityNormal, 1, 0);
osThreadDef(led_2_thread, osPriorityNormal, 1, 0);
osThreadDef(led_3_thread, osPriorityNormal, 1, 0);

// OTHER FUNCTIONS

// function prototype for our dumb delay function
void dumb_delay(uint32_t delay);
```

```
// THREAD INITIALISATION

// create the threads
int init_thread(void)
{
  // initialize peripherals here
  init_gpio(led1, OUTPUT);
  init_gpio(led2, OUTPUT);
  init_gpio(led3, OUTPUT);

  // create the thread and get its taks id
  tid_led_1_thread = osThreadCreate(osThread(led_1_thread), NULL);
  tid_led_2_thread = osThreadCreate(osThread(led_2_thread), NULL);
  tid_led_3_thread = osThreadCreate(osThread(led_3_thread), NULL);

  // check if everything worked ...
  if(!(tid_led_1_thread && tid_led_2_thread && tid_led_3_thread))
  {
    return(-1);
  }
  return(0);
}

// ACTUAL THREADS

// blink led 1
void led_1_thread(void const *argument)
{
  while(1)
  {
    // toggle the first led on the gpio pin
    toggle_gpio(led1);
    dumb_delay(500);
  }
}

// blink led 2
void led_2_thread(void const *argument)
{
  while(1)
  {
    // toggle the second led on the gpio pin
    toggle_gpio(led2);
    dumb_delay(200);
  }
}

// blink led 3
void led_3_thread(void const *argument)
{
  while(1)
  {
    // toggle the second led on the gpio pin
    toggle_gpio(led3);
    dumb_delay(1000);
  }
}
```

```
// OTHER FUNCTIONS

// dumb delay function
void dumb_delay(uint32_t delay)
{
  // just spin through processor cycles to introduce a delay
  long delaytime;
  for(delaytime = 0; delaytime < (delay * 10000); delaytime++)
  {
    __nop();
  }
}
```

Compile this project and load it on to the STM32F7 discovery board. Make sure everything works as expected.

We can use the debugging features in uVision to help us see exactly what is happening in this example. First of all we need to make sure that the debugger is set up correctly (otherwise things either wont work, or will give us weird timing errors!).

Make sure that the “Core Clock” setting in the Debug > Trace tab is set to 216MHz, “Trace Enable” is checked and (at the very least) ITM Stimulus Port 31 is checked (this enables us to see thread switches) – see Figure 4 for the debug settings.

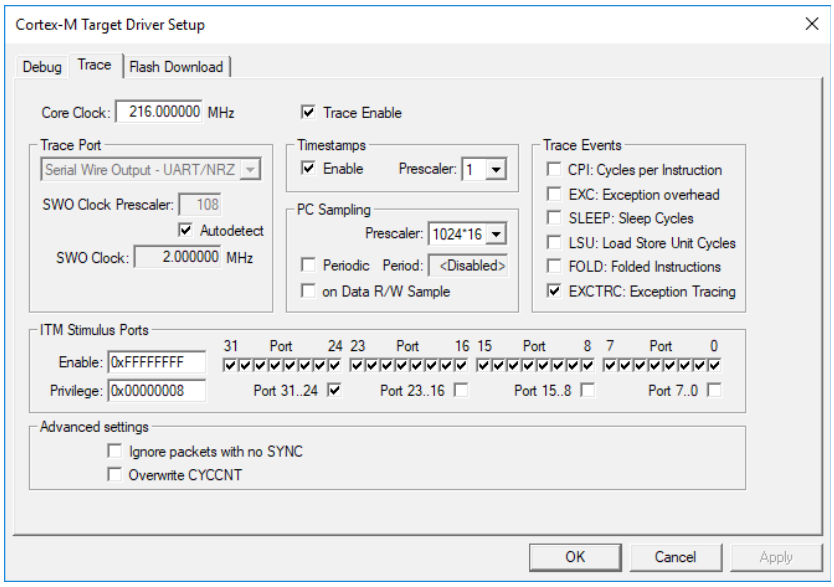


Figure 4 – Debug > Trace settings

Next, we need to ensure that the “Initialization File” option is set to point to the “stm32_swo.ini” which helps set up the serial wire view in the debugger for us (see Figure 5).

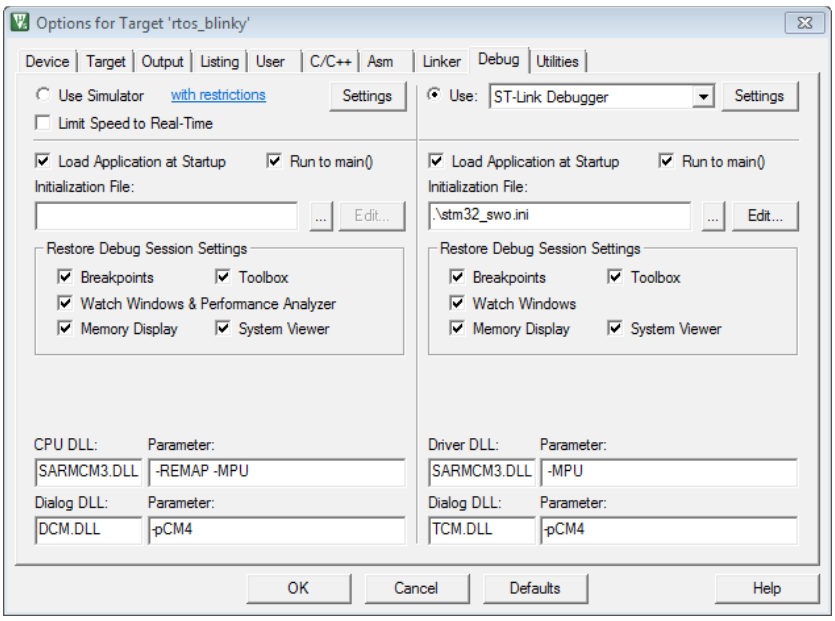


Figure 5 – Project options > Debug tab

Now you can see which threads are in which state in the Debug > OS Support > System and Thread Viewer (see Figure 6). You can also use the Debug > OS Support > Event Viewer to see the time spent in each thread and the actual time slices between them (see Figure 7). As all three of our LED threads are of the same priority, you can see that the RTOS scheduler evenly alternates between them.

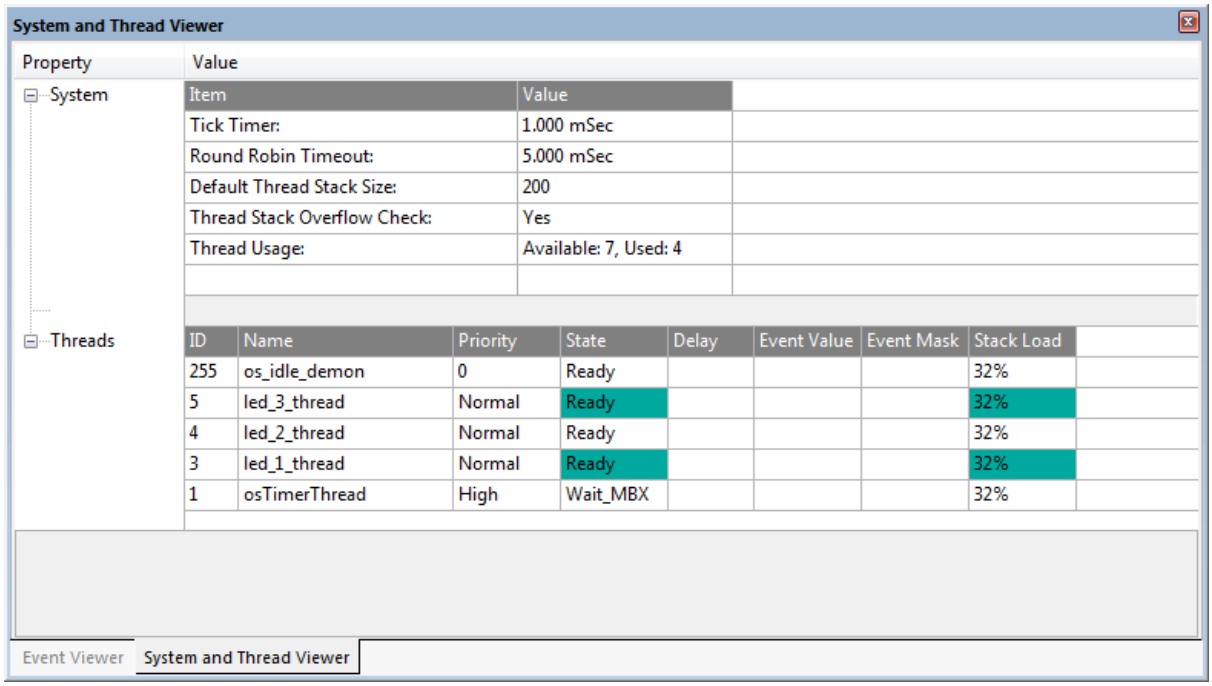


Figure 6 – The System and Thread Viewer window

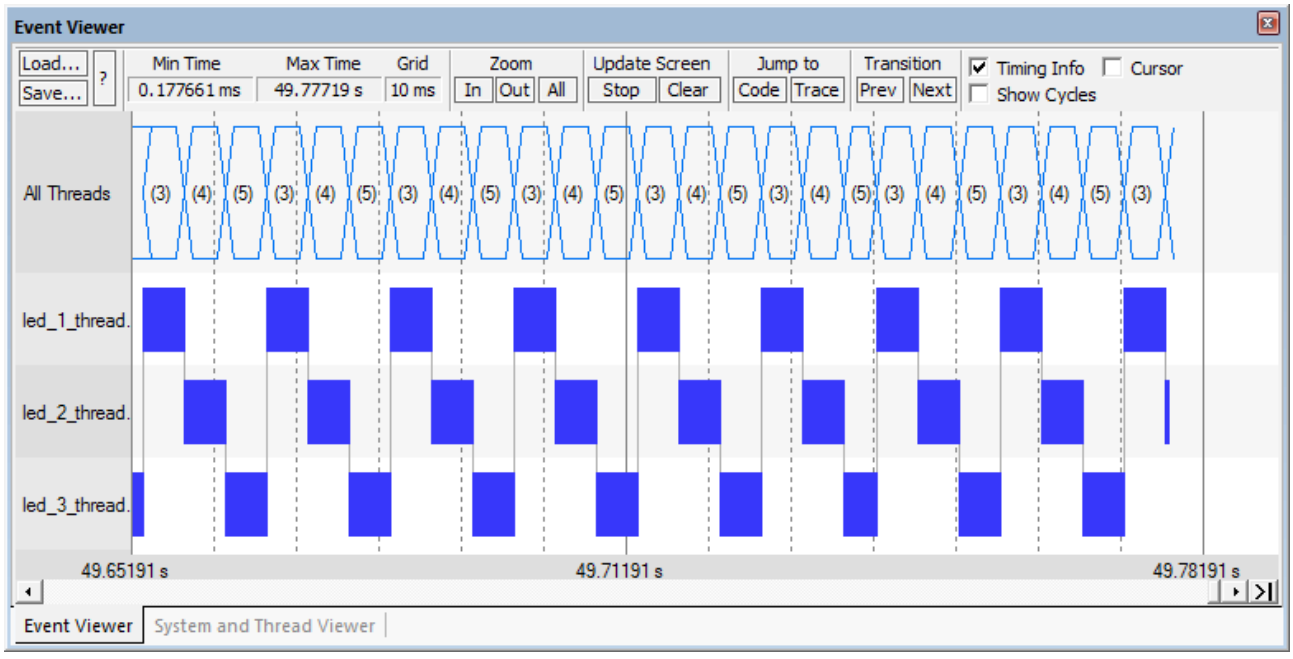


Figure 7 – The Event Viewer window

You should be able to see that, in the System and Thread Viewer shown in Figure 6, the os_idle_demon never runs. This is because there is no point in our application (after we start the RTOS scheduler running) where we are not in one of the LED threads.

Now change "`dumb_delay(xxx)`" to "`osDelay(xxx)`".

What happens to the os_idle_demon now?

What about the event viewer trace?

Task 2

When we design an application using threads (as in the previous example) we are focussing on the flow of messages in our application (see Figure 8).

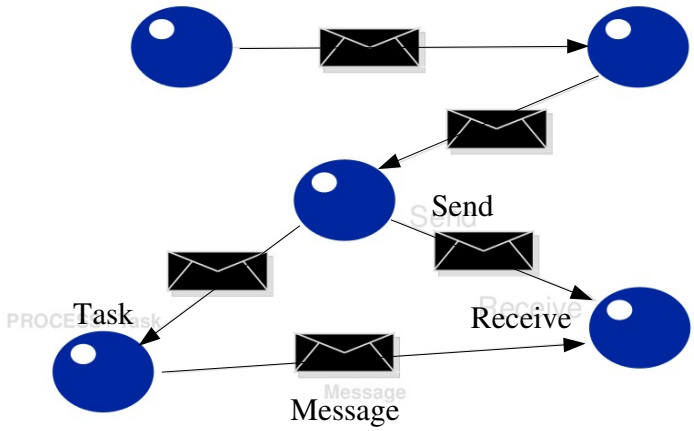


Figure 8 – Threaded application design

One of the powerful features of the CMSIS RTOS abstraction layer is it's provision of data exchange functionality. Whilst features such as signalling, semaphores, and mutexes allow us to synchronise code and pass control to other threads, any real-world application will need to provide some way of passing data between processes. Theoretically this is possible using global shared variables with mutexes or semaphores controlling access to them – however, this is not an elegant solution and (in anything but very simple programs) often leads to unforeseen errors. What we really want is a robust, asynchronous form of message passing.

CMSIS-RTOS provides two main methods of transferring data between threads: message queues (see Figure 9) and mail queues (see Figure 10). The key difference is that message queues are designed to transfer integer values, whilst mail queues can be easily used with blocks of data (such as structs).

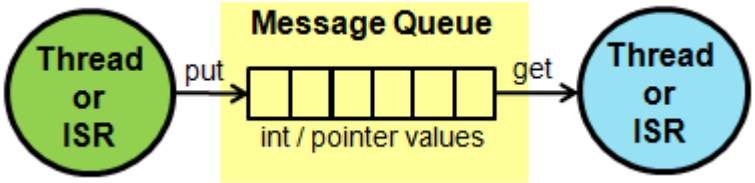


Figure 9 – Message queue

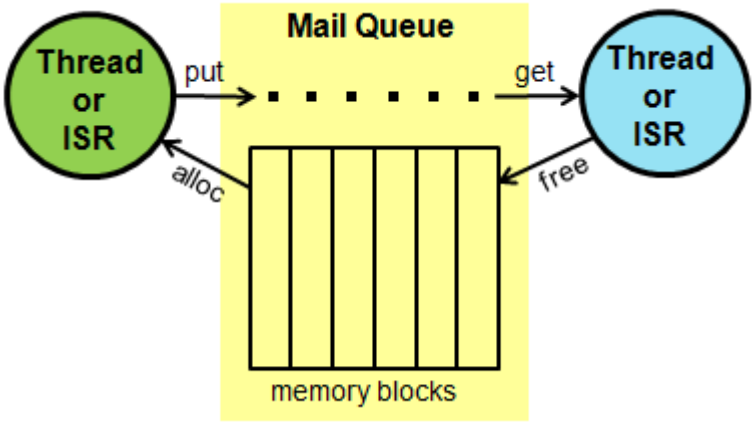


Figure 10 – Mail queue

In this exercise we are going to use one thread to create some data (simulating something like an analog to digital conversion) and another thread to display that data over the serial port. Exchange of this data between threads will be performed using a mail queue. Figure 11 shows our application architecture.

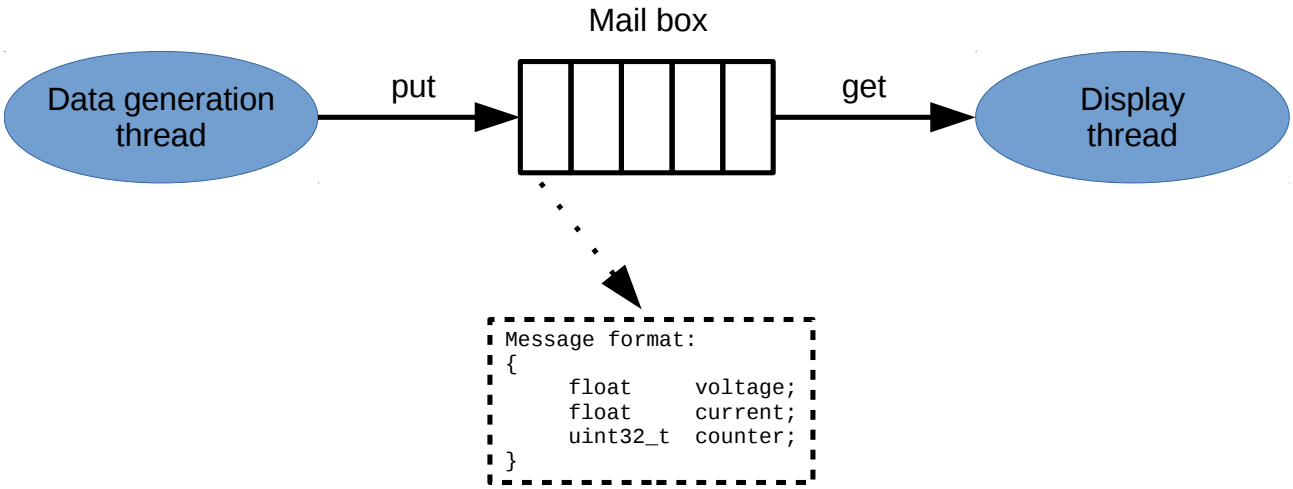


Figure 11 – Our application architecture

Code listing 3 shows our **`main.c`** file – you can see that is is almost identical to that used in the previous exercise (except we are now initialising two sets of threads rather than one).

Code listing 3:

```
/*
 * main.c
 *
 * this is the main rtos mail queue application
 *
 * author:    Dr. Alex Shenfield
 * date:      06/09/2017
 * purpose:   55-604481 embedded computer networks : lab 103
 */

// include the basic headers for the hal drivers
#include "stm32f7xx_hal.h"
#include "cmsis_os.h"

// include the clock configuration file from the shu bsp library
#include "clock.h"

// declare the extern methods that set the threads up for us
extern int init_data_thread(void);
extern int init_display_thread(void);

// this is the main method
int main()
{
  // initialise the real time kernel
  osKernelInitialize();

  // we need to initialise the hal library and set up the SystemCoreClock
  // properly
  HAL_Init();
  init_sysclk_216MHz();

  // at this stage the microcontroller clock setting is already configured.
  // this is done through SystemInit() function which is called from the
  // startup file (startup_stm32f7xx.s) before to branch to application main.
  // to reconfigure the default setting of SystemInit() function, refer to
  // system_stm32f7xx.c file
  //
  // note also that we need to set the correct core clock in the RTX_Conf_CM.c
  // file (OS_CLOCK) which we can do using the configuration wizard

  // initialise our threads
  init_data_thread();
  init_display_thread();

  // start everything running
  osKernelStart();
}
```

Code listing 4 shows the **`main.h`** header file for our application – the most important thing to note in this file is the type definition for our "mail_t" data structure. This data structure is what we are going to use to store our data in and what we are going to stuff into the mailbox to exchange this data between threads.

Code listing 4:

```
/*
 * main.h
 *
 * this is the header for the main rtos mail queue application
 *
 * author:    Dr. Alex Shenfield
 * date:      06/09/2017
 * purpose:   55-604481 embedded computer networks : lab 103
 */

// define to prevent recursive inclusion
#ifndef __MAIN_H
#define __MAIN_H

// include the basic headers for the hal drivers and the rtos library
#include "stm32f7xx_hal.h"
#include "cmsis_os.h"

// set up our mailbox

// mail data structure
typedef struct
{
  float     voltage;
  float     current;
  uint32_t  counter;
}
mail_t;

#endif // MAIN_H
```

Code listing 5 shows our data generation thread. The initialisation function for this thread creates our mailbox, and then the data_thread function generates the fake data, fills the mail_t data structure, and then puts it into the mailbox. This is then picked up by our display thread (see code listing 6).

Code listing 5:

```
/*
 * data_generation_thread.c
 *
 * this is a thread that periodically generates some data and puts it in a
 * mail queue
 *
 * author:    Dr. Alex Shenfield
 * date:      06/09/2017
 * purpose:   55-604481 embedded computer networks : lab 103
 */

// include cmsis_os for the rtos api
#include "cmsis_os.h"

// include main.h (this is where we initialise the mail type)
#include "main.h"

// include the shu bsp libraries for the stm32f7 discovery board
#include "pinmappings.h"
#include "clock.h"
#include "random_numbers.h"
#include "gpio.h"

// RTOS DEFINES

// declare the thread function prototypes, thread id, and priority
void data_thread(void const *argument);
osThreadId tid_data_thread;
osThreadDef(data_thread, osPriorityNormal, 1, 0);

// set up the mail queue
osMailQDef(mail_box, 16, mail_t);
osMailQId  mail_box;

// HARDWARE DEFINES

// led is on PI 1 (this is the inbuilt led)
gpio_pin_t led1 = {PI_1, GPIOI, GPIO_PIN_1};
```

```
// THREAD INITIALISATION

// create the data generation thread
int init_data_thread(void)
{
  // initialize peripherals (i.e. the led and random number generator) here
  init_gpio(led1, OUTPUT);
  init_random();

  // create the mailbox
  mail_box = osMailCreate(osMailQ(mail_box), NULL);

  // create the thread and get its task id
  tid_data_thread = osThreadCreate(osThread(data_thread), NULL);

  // check if everything worked ...
  if(!tid_data_thread)
  {
    return(-1);
  }
  return(0);
}

// ACTUAL THREADS

// data generation thread - create some random data and stuff it in a mail
// queue
void data_thread(void const *argument)
{
  // set up our counter
  uint32_t i = 0;

  // infinite loop generating our fake data (one set of samples per second)
  // we also toggle the led so we can see what is going on ...
  while(1)
  {
    // create our mail (i.e. the message container)
    mail_t* mail = (mail_t*) osMailAlloc(mail_box, osWaitForever);

    // get a random number
    float random = get_random_float();

    // toggle led
    toggle_gpio(led1);

    // generate our fake data
    i++;
    mail->counter = i;
    mail->current = (1.0f / (random * i));
    mail->voltage = (5.0f / (random * i));

    // put the data in the mail box and wait for one second
    osMailPut(mail_box, mail);
    osDelay(1000);
  }
}
```

Code listing 6 shows our data display thread. In this example we are using the UART to send the data over the serial port to our terminal program (e.g. TeraTerm). The initialisation function for this thread sets up our hardware (i.e. the COM port) using the routines in **`serial.c`** / **`serial.h`**[2], before creating the actual data display thread. The ***display_thread*** function then grabs the data from the mailbox (when available) and prints it to the serial port.

It is important to note the use of

```
// the mailbox we are pulling data from is declared elsewhere ...
extern osMailQId mail_box;
```

at the top of our "**`data_display_thread.c`**" file. This use of the "***extern***" key word tells the compiler that the mailbox is declared outside this source code file – this allows us to declare the mailbox in the data generation thread and pull data out of the mailbox in the data display thread.

Code listing 6:

```
/*
 * data_display_thread.c
 *
 * this is a thread that pulls the data generated by another thread from a
 * mail queue and then displays it in a terminal
 *
 * author:    Dr. Alex Shenfield
 * date:      06/09/2017
 * purpose:   55-604481 embedded computer networks : lab 103
 */

// include cmsis_os for the rtos api and the stdio package for printing
#include "cmsis_os.h"
#include <stdio.h>

// include main.h (this is where we initialise the mail type)
#include "main.h"

// include the shu bsp libraries for the stm32f7 discovery board and the serial
// configuration files
#include "pinmappings.h"
#include "clock.h"
#include "serial.h"

// RTOS DEFINES

// declare the thread function prototypes, thread id, and priority
void display_thread(void const *argument);
osThreadId tid_display_thread;
osThreadDef(display_thread, osPriorityNormal, 1, 0);

// the mailbox we are pulling data from is declared elsewhere ...
extern osMailQId mail_box;
```

2 In the same way as we exposed the serial communication functions in lab 1, task 5.

```
// THREAD INITIALISATION

// create the data display thread
int init_display_thread(void)
{
  // initialize peripherals (i.e. the uart) here
  init_uart(9600);
  printf("display thread up and running ...\r\n");

  // create the thread and get its task id
  tid_display_thread = osThreadCreate(osThread(display_thread), NULL);

  // check if everything worked ...
  if(!tid_display_thread)
  {
    return(-1);
  }
  return(0);
}

// ACTUAL THREADS

// data display thread - pull the data out of the mail queue and print it to
// the uart
void display_thread(void const *argument)
{
  // infinite loop getting out fake data ...
  while(1)
  {
    // get the data ...
    osEvent evt = osMailGet(mail_box, osWaitForever);
    if(evt.status == osEventMail)
    {
      mail_t *mail = (mail_t*)evt.value.p;
      printf("\nVoltage: %.2f V\n\r"    , mail->voltage);
      printf("Current: %.2f A\n\r"     , mail->current);
      printf("Number of cycles: %u\n\r", mail->counter);

      osMailFree(mail_box, mail);
    }
  }
}
```

Note that, as well as this code, we have to make some change to **`rtx_conf_cm.c`** which contains the RTOS settings. In this case we have to change the default thread stack size and main thread stack size to 400 bytes. If we do not do this then we will get stack overflows when trying to initialise and print to the UART[3]!

Figure 12 shows this revised **`rtx_conf_cm.c`** file.

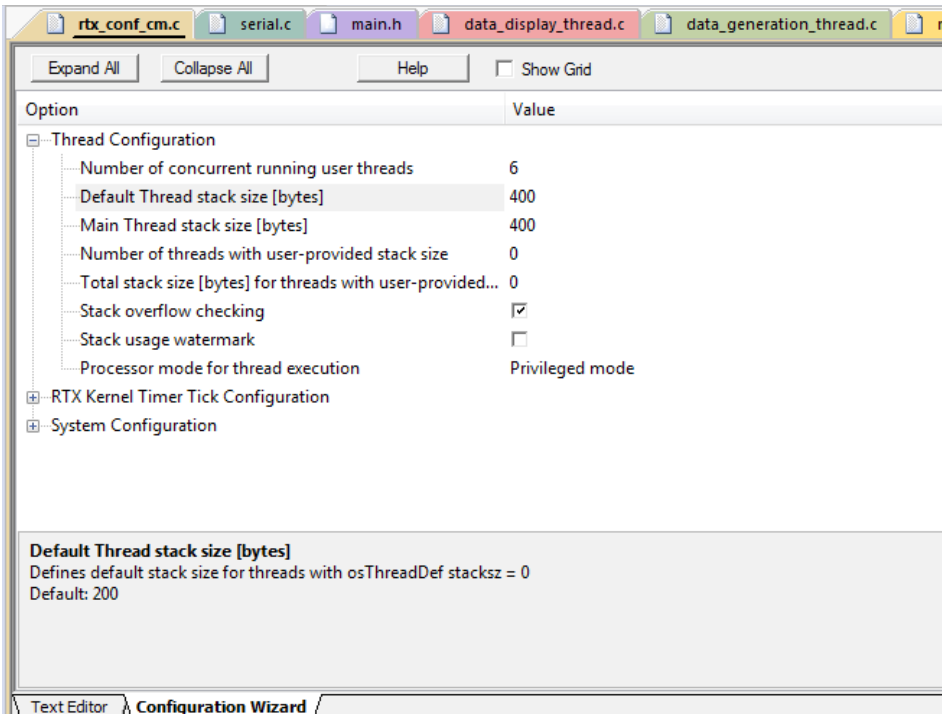


Figure 12 – The RTOS settings

Compile this project and load it on to the STM32F7 discovery board. Make sure everything works as expected (including the display of data over the serial port).

Now make some modifications to your program to enhance its functionality:

1. Try adding additional threads to blink some LEDs.

2. Add additional fields to the mail_t type to simulate sending additional data between threads.

3 This is the first thing I tend to look at when an RTOS program doesn't do what I want it to do!

Task 3

We are now going to extend the previous example to read some analog inputs (either potentiometers or temperature / light sensors) and display that data on both the LCD and over the UART. I suggest using a similar application architecture as the previous example (see Figure 13) and adapting the project from the previous example.

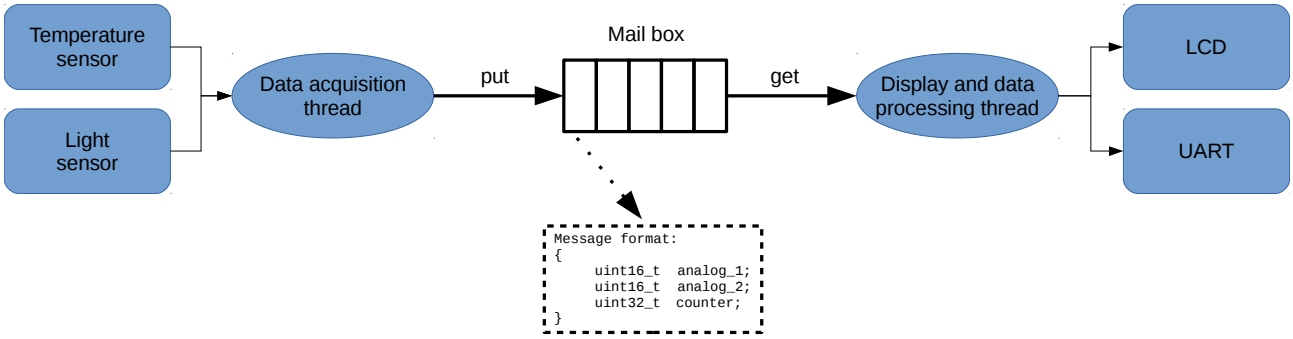


Figure 13 – Our application architecture for the real-world data acquisition and display example

Figure 14 shows an illustration of what this system may look like.

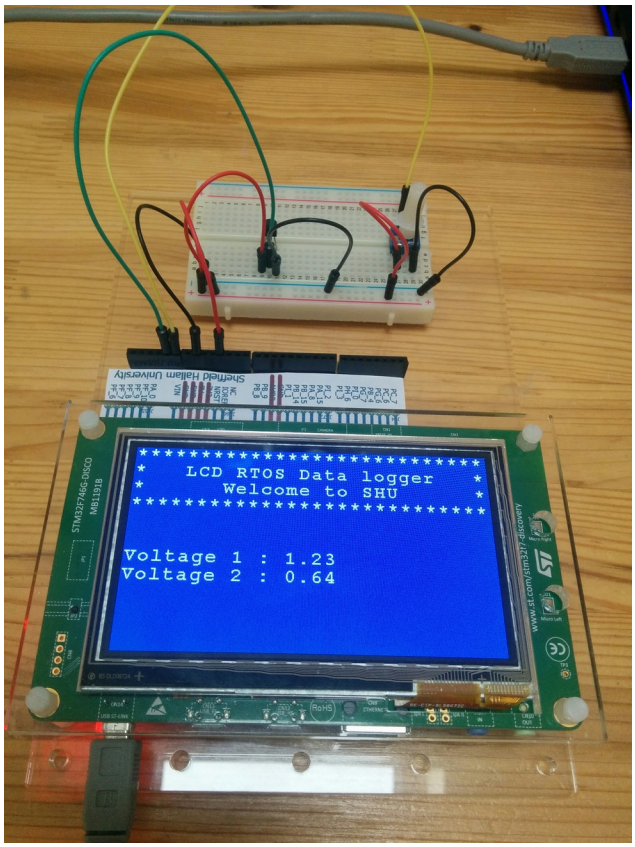


Figure 14 – Working data acquisition and display system

I recommend developing this application using the following steps:

1. Make a clone of the previous project by copying and pasting the inc, mdk-arm, and src directories into the provided 3_rtos_application folder

2. Add the **adc.c** library into the **stm32f7_discovery_bsp_shu** group (this file is located in **libraries\bsp\stm32f7discovery_shu_kit\src**)

3. Add the **stm32746g_discovery.c**, **stm32746g_discovery_lcd.c**, and **stm32746g_discovery_sdram.c** into the **stm32f7_discovery_bsp** group (these files are located in **libraries\bsp\stm32f7_discovery\bsp\src**)

4. Adapt the code from lab 1, task 5 to read from the ADC and print those values on to the LCD screen. To do this you will have to create formatted strings using sprintf (as we did in lab 2).

5. To use the STM32F7 discovery board LCD screen with the real-time operating system you need to make sure to override the **HAL_Delay** function so as to use the **osDelay** function from the CMSIS RTOS library. You can do this by adding the code below to the main.c file:

```
// OVERRIDE HAL DELAY

// make HAL_Delay point to osDelay (otherwise any use of HAL_Delay breaks things)
void HAL_Delay(__IO uint32_t Delay)
{
  osDelay(Delay);
}
```

   If the built in **HAL_Delay** function is used, then the application will freeze (because the RTOS does not provide the HAL library with an accurate SysTick value – which is needed for the **HAL_Delay**).

6. The final modification that is almost certain to be needed to make the code run properly is to increase the default stack size for all the threads (as discussed in the previous task). Both our data acquisition thread and our data display thread are doing quite a lot of work in this application – and therefore we will give them a sizeable stack to work with. Figure 15 shows the RTOS configuration settings that I have used successfully.

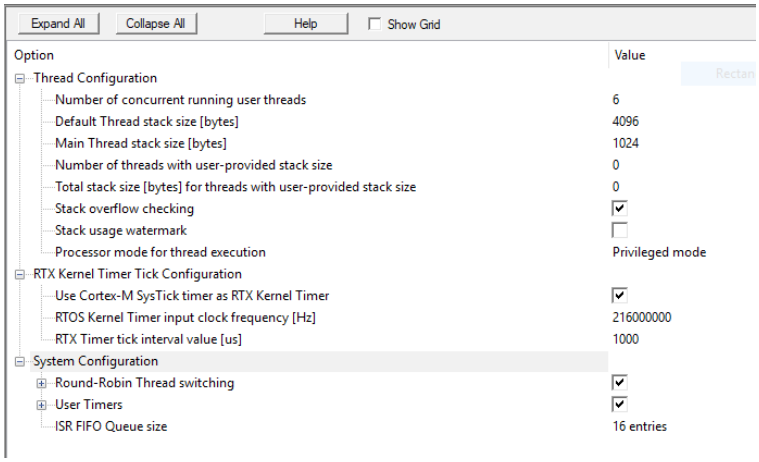


Figure 15 – RTOS configuration settings for the full application

Additional desirable features you should look to implement are:

- Conversion back into real-world measurements (voltage / temperature / etc,)
- Display of data as graphs on the LCD
- Lighting up LEDs when temperature or light levels drop below a certain set point

Task 4

In this task, we are going to send serial commands to our embedded system and have it process that data and react to these commands. The outline specification for our application is:

- The system must respond to commands coming from the serial port
- Commands always end with a new line / line feed character (i.e. '\n')
- Commands arrive one at a time
- The serial port has minimal hardware buffering capability and characters may arrive quickly
- The system can respond to those commands relatively slowly

To do this we will have to use interrupts to store the data coming in on the serial port. One possible way would be to have all the data processing (e.g. looking for the new line character and responding to those commands within the interrupt – however, we have established in the lectures that this is **A TERRIBLE IDEA!** This will lead to code that is complex, difficult to debug, and will have non-deterministic behaviour (because the interrupts will mess up the rest of the program flow – even low priority interrupts will run before high priority tasks).

Instead, a better approach is to put minimal data processing in the interrupt itself and use a different thread to do all the processing of the data. There are a few ways to accomplish this, but one of the simplest is just to put all the received bytes into a message queue and let the UART processing thread pull them out whenever they are available. This application architecture is shown in Figure 16.

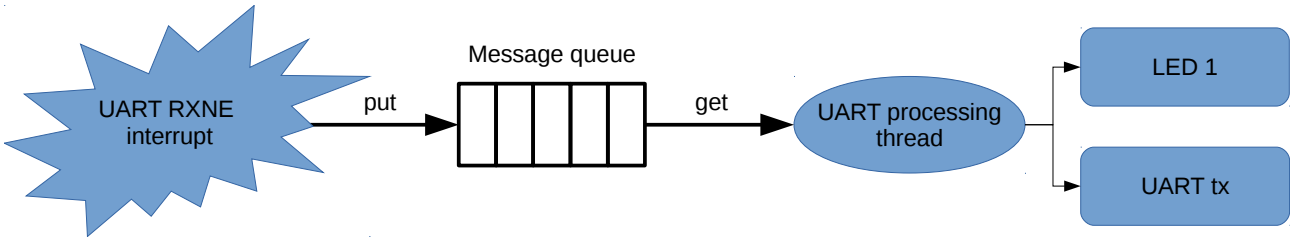


Figure 16 – The UART application architecture (note: RXNE stands for receive not empty and means that there is received data to be processed by the interrupt)

Although this application architecture looks fairly simple, the implementation is reasonably complex. There are a few things we have to be really careful about:

1. We must make sure we enable the interrupt correctly.
2. We cannot block in the interrupt – therefore any methods we call (for example, to put a message in a queue) must return immediately.
3. We must clear the interrupt flag before returning to the program.
4. We must use an RTOS that is interrupt aware.

Our **`main.c`** file remains practically the same as in the previous exercises, so I will not replicate it here again. It is available as part of the project on blackboard / github.

Code listing 7 shows part of the **`stm32f7xx_it.c`** file (which, by convention, is where we define interrupt handlers).

Code listing 7

```
...

// the uart handle structure is defined elsewhere (in this case in serial.c)
extern UART_HandleTypeDef uart_handle;

// interrupt handler for uart 6
void USART1_IRQHandler(void)
{
  // so under the stm32f7xx hal setup, we pass off the uart interrupt to the hal
  // uart interrupt request handler for processing - this then triggers the
  // rx / tx callbacks defined elsewhere
  HAL_UART_IRQHandler(&uart_handle);
}


...
```

When an interrupt is triggered this interrupt handler is called. However, using the STM32F7 HAL libraries, rather than using the interrupt handler to put the data in a message queue, we pass control to the UART callbacks (see code listing 8). These UART callbacks can then be defined in our "**`serial.c`**" file (or elsewhere). This enables a neater separation of code – all the interrupt handler code goes in **`stm32f7xx_it.c`**, all the UART code goes in **`serial.c`**, and all the application code goes in its own file(s).

This is a new way of working introduced with the HAL libraries.

Code listing 8

```
...

// RTOS DEFINES

// the message queue we are using to pass characters around is defined
// elsewhere (in this case in the uart_processing_thread.c file)
extern osMessageQId msg_q;

...

// UART IRQ CALLBACKS

// uart receive callback
void HAL_UART_RxCpltCallback(UART_HandleTypeDef * uart_handle)
{
  // stuff characters into the message queue (and return immediately - as this
  // is basically an isr ...)
  osMessagePut(msg_q, c, 0);

  // enable the interrupt again ...
  HAL_UART_Receive_IT(uart_handle, &c, 1);
}

...
```

Code listing 9 shows our **`uart_processing_thread.c`** file. This contains our **`uart_rx_thread()`** method (which does all the data processing) and our thread initialisation function (which is called from our main method and sets up all the hardware, initialises the message queue, and creates our actual thread).

Code listing 9

```
/*
 * uart_processing_thread.c
 *
 * a thread to handle uart communications
 *
 * - data is put into the message queue byte by byte as it comes in over the
 *   uart by the irq
 * - the rx thread grabs the available data from the message queue and decides
 *   whether to turn the led on or off
 *
 * author:    Dr. Alex Shenfield
 * date:      06/09/2017
 * purpose:   55-604481 embedded computer networks : lab 103
 */

// include the relevant header files (from the c standard libraries)
#include <stdio.h>
#include <string.h>

// include the serial configuration files and the rtos api
#include "serial.h"
#include "cmsis_os.h"

// include the shu bsp libraries for the stm32f7 discovery board
#include "pinmappings.h"
#include "clock.h"
#include "gpio.h"

// RTOS DEFINES

// declare the thread function prototypes, thread id, and priority
void uart_rx_thread(void const *argument);
osThreadId tid_uart_rx_thread;
osThreadDef(uart_rx_thread, osPriorityAboveNormal, 1, 0);

// setup a message queue to use for receiving characters from the interrupt
// callback
osMessageQDef(message_q, 128, uint8_t);
osMessageQId msg_q;

// HARDWARE DEFINES

// map the led to GPIO PI2
gpio_pin_t led = {PI_2, GPIOI, GPIO_PIN_2};
```

```
// THREAD INITIALISATION

// create the uart thread(s)
int init_uart_threads(void)
{
  // initialise gpio output for led
  init_gpio(led, OUTPUT);

  // set up the uart at 9600 baud and enable the rx interrupts
  init_uart(9600);
  enable_rx_interrupt();

  // print a status message
  printf("we are alive!\r\n");

  // create the message queue
  msg_q = osMessageCreate(osMessageQ(message_q), NULL);

  // create the thread and get its task id
  tid_uart_rx_thread = osThreadCreate(osThread(uart_rx_thread), NULL);

  // check if everything worked ...
  if(!tid_uart_rx_thread)
  {
    return(-1);
  }
  return(0);
}
```

```
// ACTUAL THREAD(S)

// uart receive thread
void uart_rx_thread(void const *argument)
{
  // print some status message ...
  printf("still alive!\r\n");

  // create a packet array to use as a buffer to build up our string
  uint8_t packet[128];
  int i = 0;

  // infinite loop ...
  while(1)
  {
    // wait for there to be something in the message queue
    osEvent evt = osMessageGet(msg_q, osWaitForever);

    // process the message queue ...
    if(evt.status == osEventMessage)
    {
      // get the message and increment the counter
      uint8_t byte = evt.value.v;

      // if we get a new line character ...
      if(byte == '\n')
      {
        // add string terminator (we need this - otherwise we will keep reading
        // past the end of the last packet received ...)
        packet[i] = '\0';

        // decide whether to turn the led on or off by doing a string
        // comparison ...
        // note: the strcmp function requires an exact match so if you are
        // using teraterm or putty you will need to set the tx options
        // correctly
        // see: http://www.tutorialspoint.com/c_standard_library/c_function_strcmp.htm
        // for strcmp details ...
        if(strcmp((char*)packet, "on-led1\r") == 0)
        {
          write_gpio(led, HIGH);
          printf("led 1 on\r\n");
        }
        if(strcmp((char*)packet, "off-led1\r") == 0)
        {
          write_gpio(led, LOW);
          printf("led 1 off\r\n");
        }

        // print debugging message to the uart
        printf("DEBUGGING: %s\r\n", packet);

        // zero the packet index
        i = 0;
      }
      else
      {
        packet[i] = byte;
        i++;
      }
    }
  }
}
```

Build this project and load it on to the discovery board.

Note: you will need to make sure that your terminal emulator (e.g. teraterm) is configured properly to send new lines – see Figure 17.

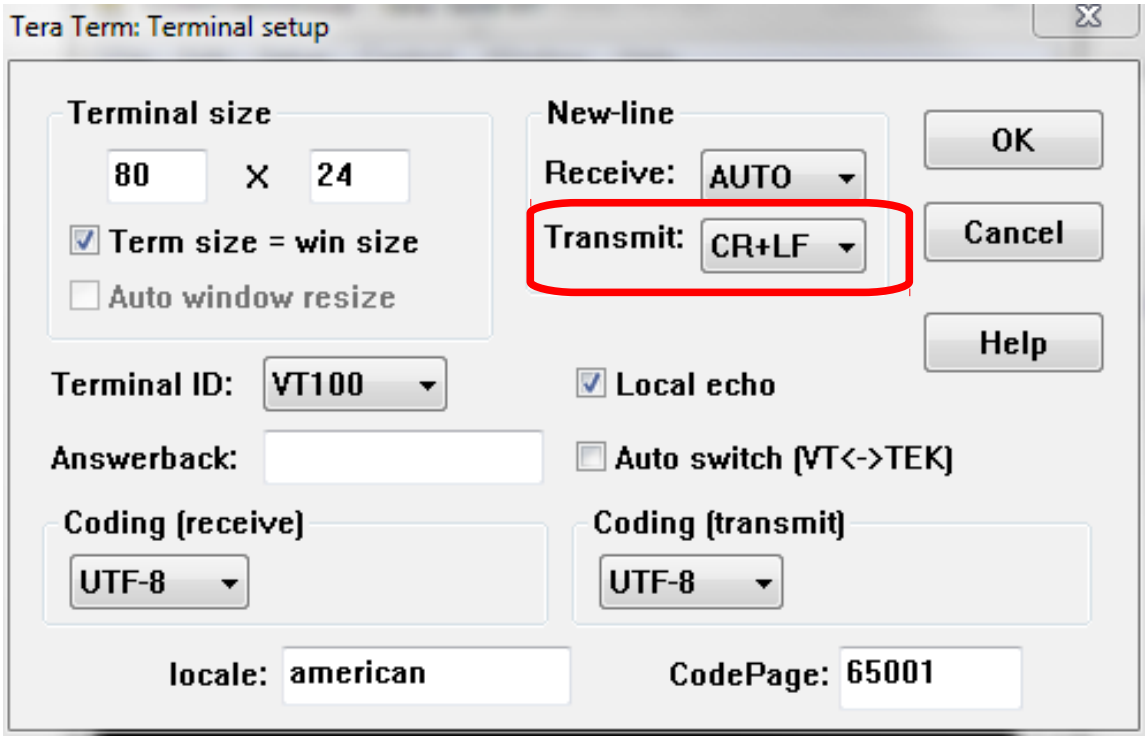


Figure 17 – Teraterm configuration

All being well, you should see a welcome message and be able to send commands to the system (see Figure 18).

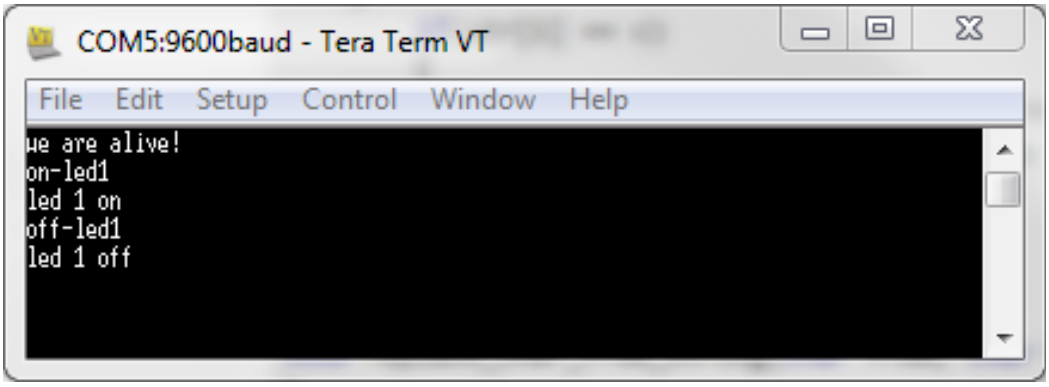
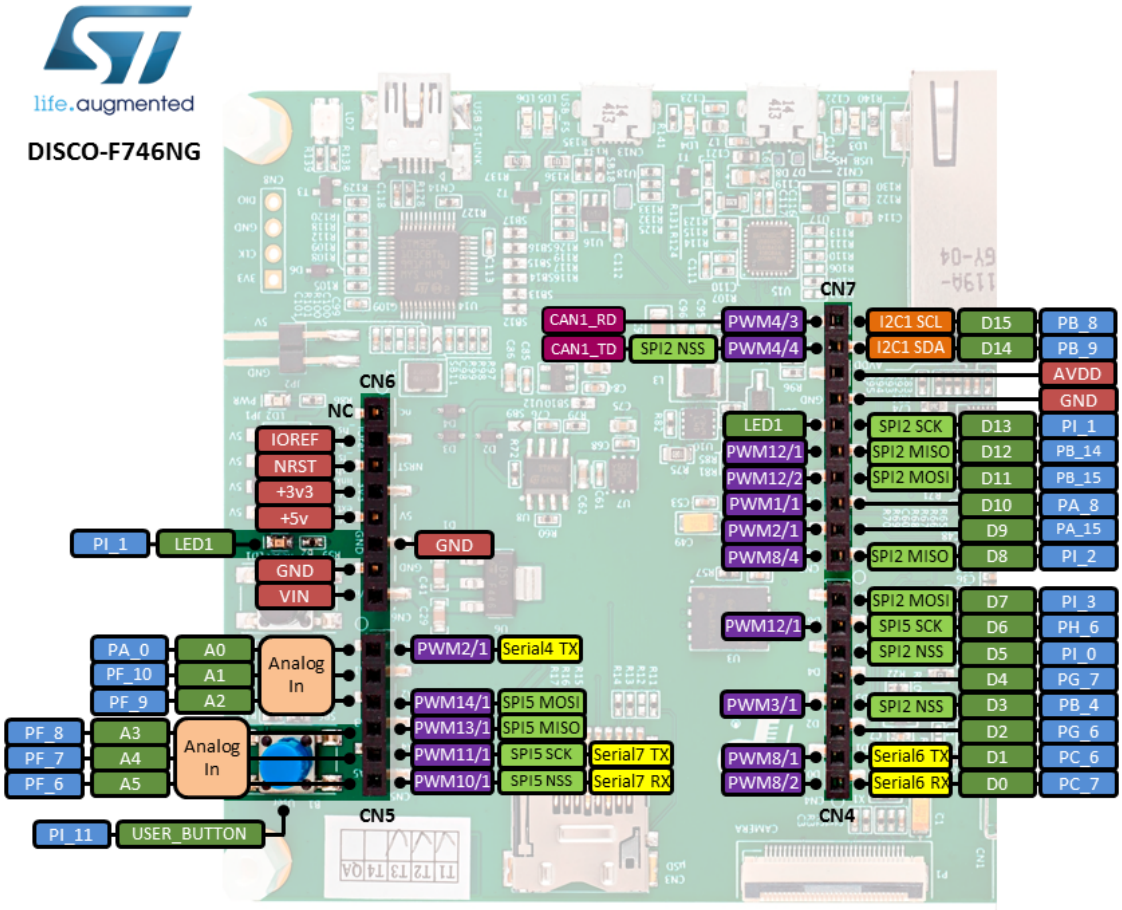


Figure 18 – Teraterm output

Additional desirable features you should implement are:

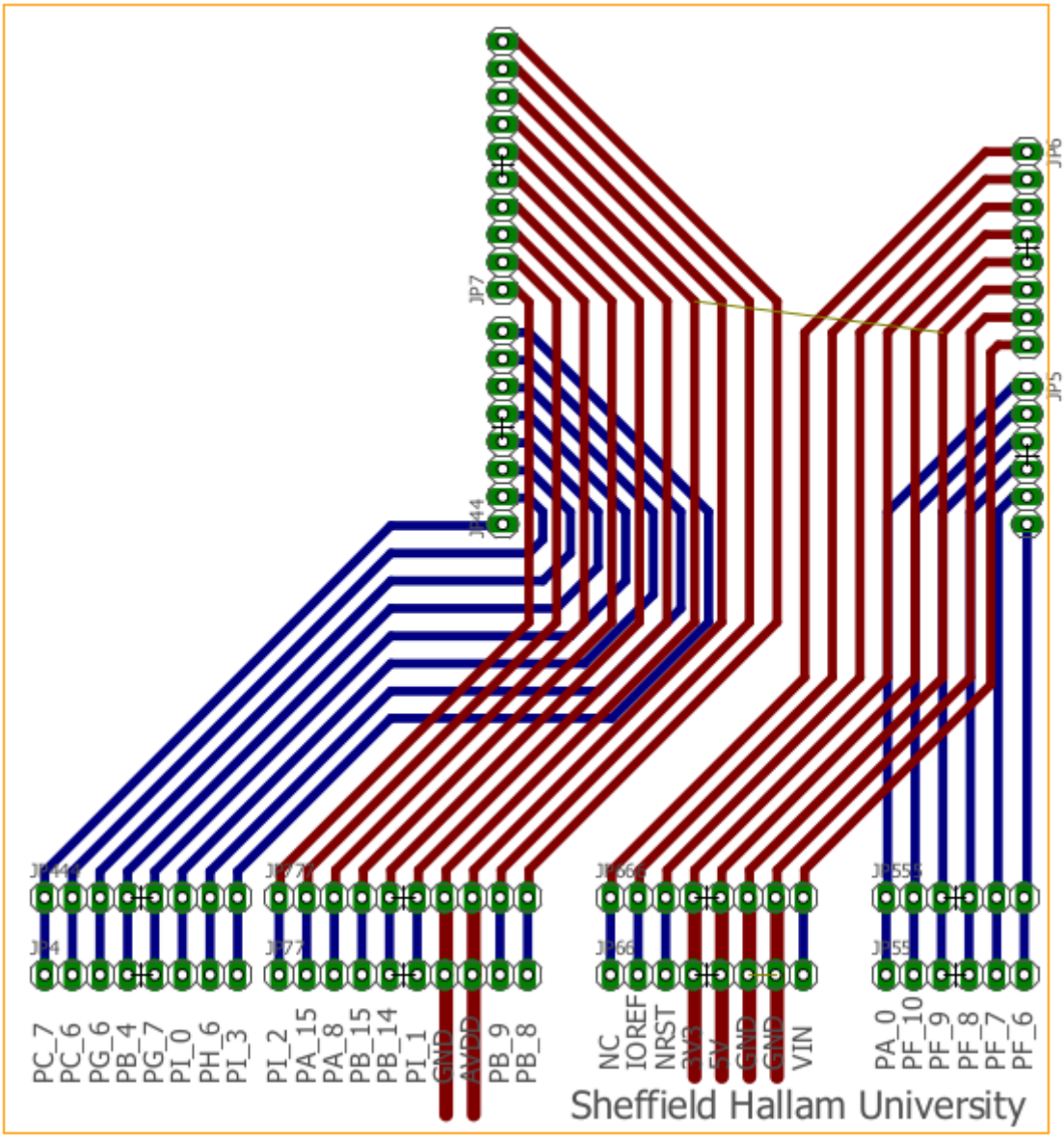
- Switching on and off other LEDs
- Reading the state of a button with a command
- Reading an analog input with a command

Appendix A – The STM32F7 discovery board schematic



STM32F7 discovery board pin outs

Appendix B – The STM32F7 discovery board SHU base board schematic



SHU breakout board for the STM32F7 discovery