Sheffield Hallam University
Department of 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	1	20	09-22	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).

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-RTOS2 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)!

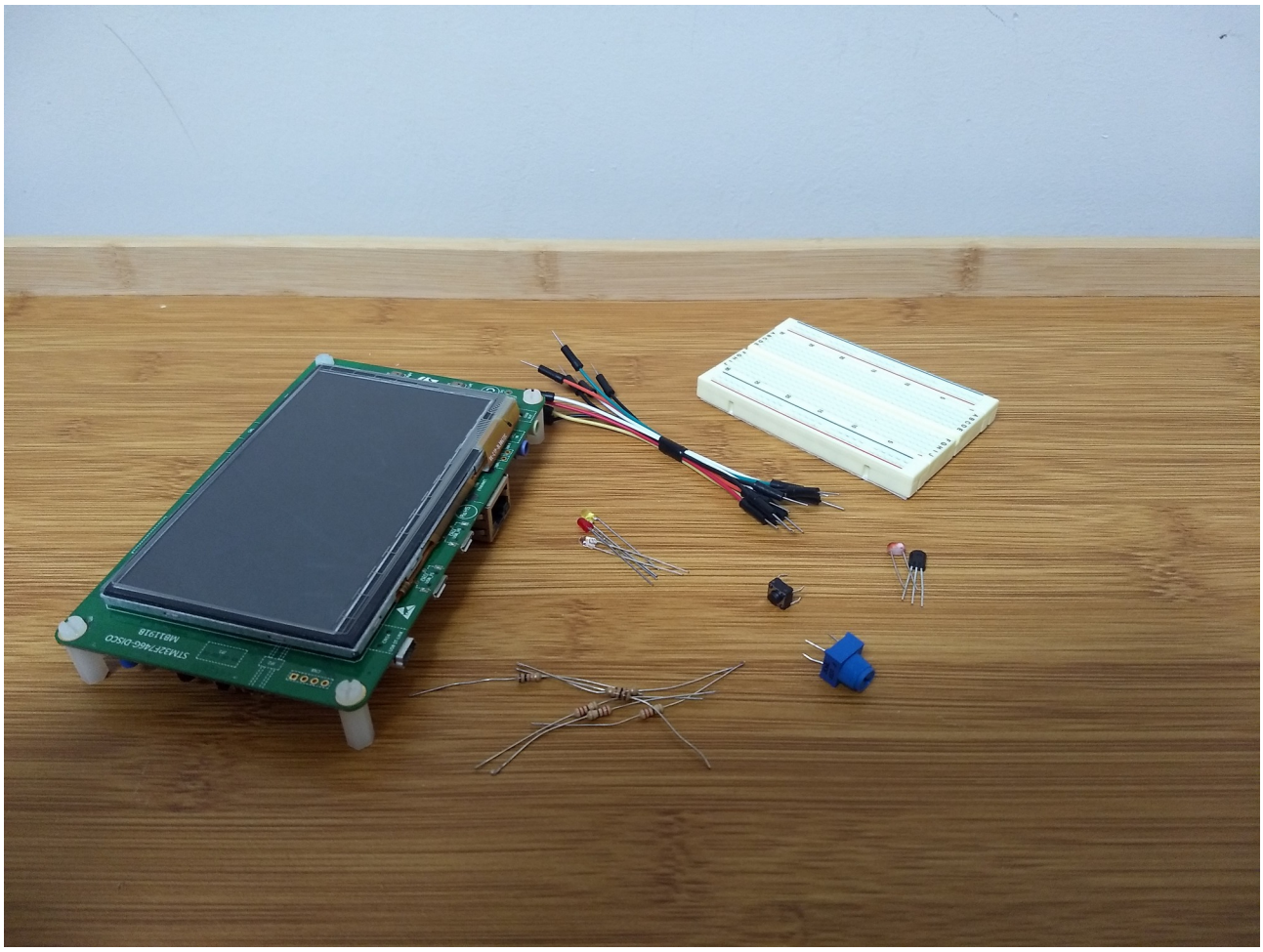


Figure 1 – The necessary equipment for this lab

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 PB_14, PB_15, and PA_8 (D12, D11, and D10 on the Arduino header)[2] – see Figure 2.

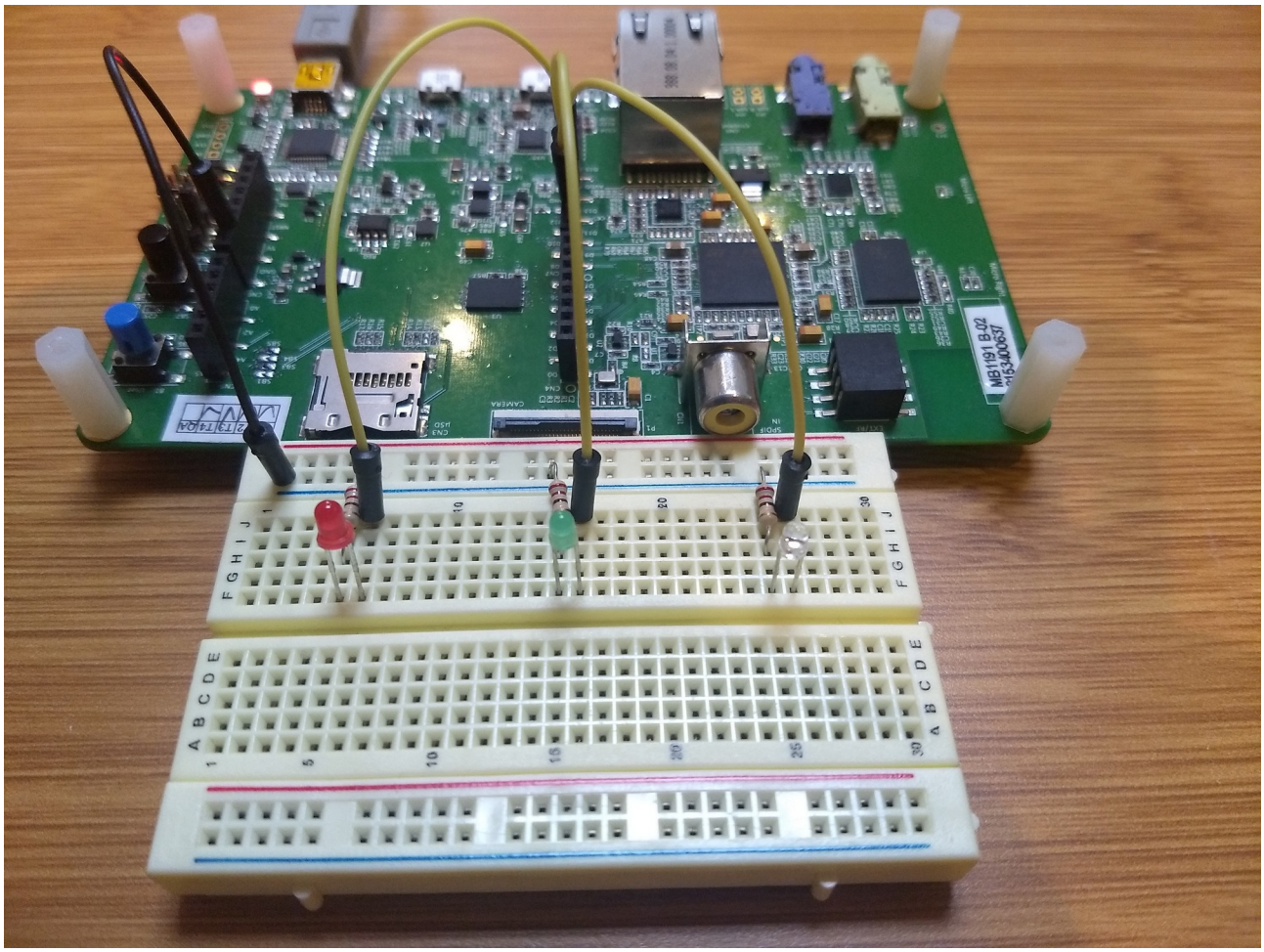


Figure 2 – The STM32F7 discovery board with connected LEDs

2 Note the use of 220Ω current limiting resistors to ground!

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 therefore 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 application
 *
 * this is deliberately written to be as generic as possible so we can reuse
 * it - we will have all our thread specific stuff in a separate file
 *
 * author:    Dr. Alex Shenfield
 * date:      12/10/2021
 * purpose:   55-604481 embedded computer networks - lab 103
 */

// include the basic headers for the hal drivers and cmsis-rtos2
#include "stm32f7xx_hal.h"
#include "cmsis_os2.h"

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

// EXTERNAL DECLARATIONS

// this is our main thread - it's declared elsewhere (in our thread specific
// file)
extern void app_main(void *arg);

// CODE

// this is the main method
int main()
{
  // 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

  // initialise the real time kernel
  osKernelInitialize();

  // create application main thread
  osThreadNew(app_main, NULL, NULL);

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

Code listing 2:

```
/*
 * blinky_thread.c
 *
 * simple threads to handle blinking some leds on the discovery board - updated
 * for the stm32f7xx hal libraries and cmsis-rtos2
 *
 * to get the thread execution information in debugging mode:
 *
 * - view -> analysis windows -> system analyzer (thread execution times)
 * - view -> watch windows -> rtx rtos (thread behaviour - e.g. waiting on
 *   delay)
 *
 * author:    Dr. Alex Shenfield
 * date:      12/10/2021
 * purpose:   55-604481 embedded computer networks - lab 103
 */

// include the basic headers for the hal drivers
#include "arm_acle.h"
#include "stm32f7xx_hal.h"
#include "cmsis_os2.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 = {PB_14, GPIOB, GPIO_PIN_14};
gpio_pin_t led2 = {PB_15, GPIOB, GPIO_PIN_15};
gpio_pin_t led3 = {PA_8,  GPIOA, GPIO_PIN_8};

// RTOS DEFINES

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

// OTHER FUNCTIONS

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

```
// ACTUAL WORKER THREADS

// THREAD 1

// led thread 1 attributes
static const osThreadAttr_t thread_1_attr =
{
  .name = "led_1",
  .priority = osPriorityNormal,
};

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

// THREAD 2

// led thread 2 attributes
static const osThreadAttr_t thread_2_attr =
{
  .name = "led_2",
  .priority = osPriorityNormal,
};

// blink led 2
void led_2_thread(void *argument)
{
  while(1)
  {
    // toggle the second led
    toggle_gpio(led2);
    dumb_delay(1000);
  }
}

// THREAD 3

// led thread 3 attributes
static const osThreadAttr_t thread_3_attr =
{
  .name = "led_3",
  .priority = osPriorityNormal,
};

// blink led 3
void led_3_thread(void *argument)
{
  while(1)
  {
    // toggle the third led
    toggle_gpio(led3);
    dumb_delay(1500);
  }
}
```

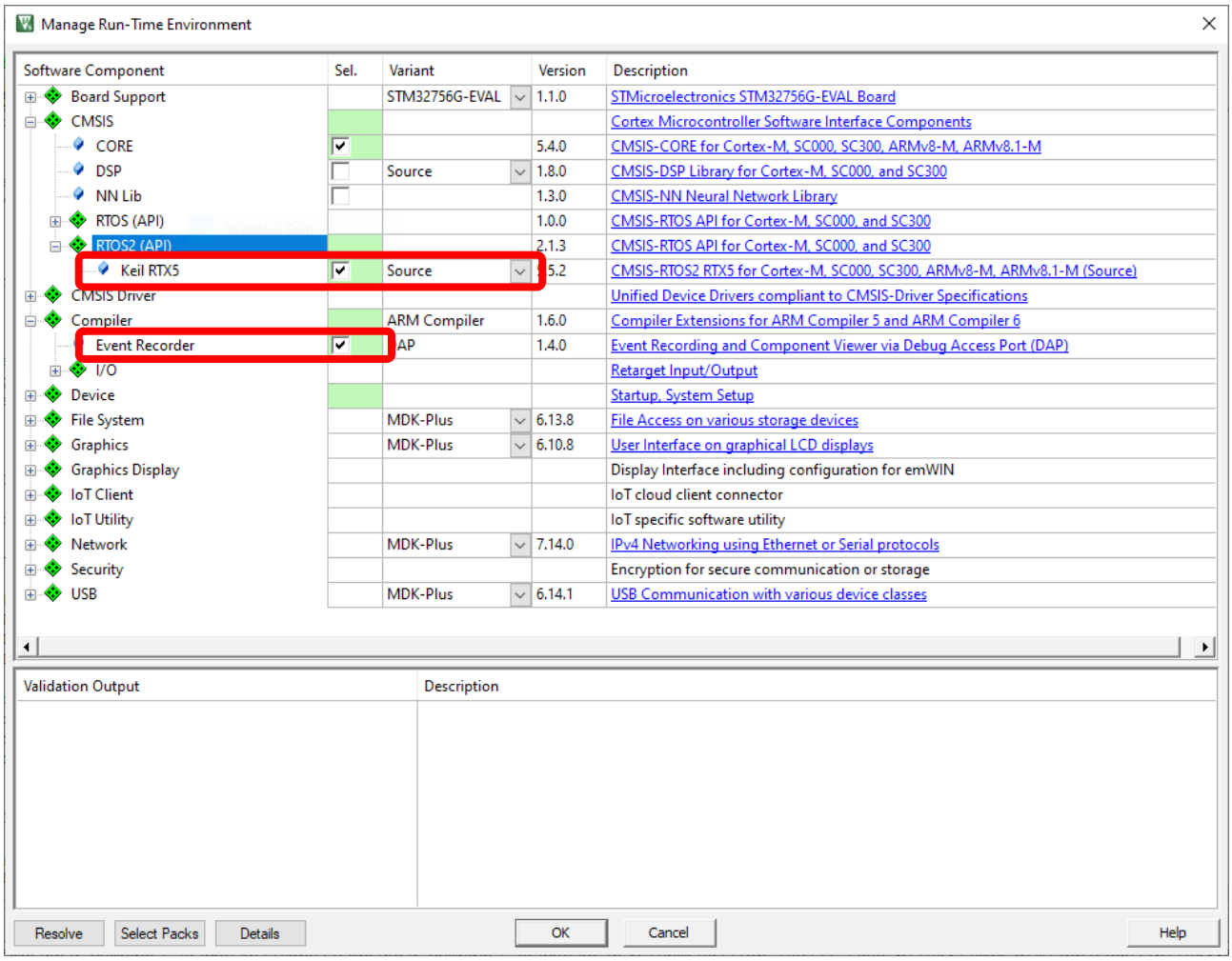
```
// APPLICATION MAIN THREAD

// application main thread - initialise peripherals and start the worker
// threads
void app_main(void *argument)
{
  // initialise peripherals here
  init_gpio(led1, OUTPUT);
  init_gpio(led2, OUTPUT);
  init_gpio(led3, OUTPUT);

  // create the threads
  tid_led_1_thread = osThreadNew(led_1_thread, NULL, &thread_1_attr);
  tid_led_2_thread = osThreadNew(led_2_thread, NULL, &thread_2_attr);
  tid_led_2_thread = osThreadNew(led_3_thread, NULL, &thread_3_attr);
}

// 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.

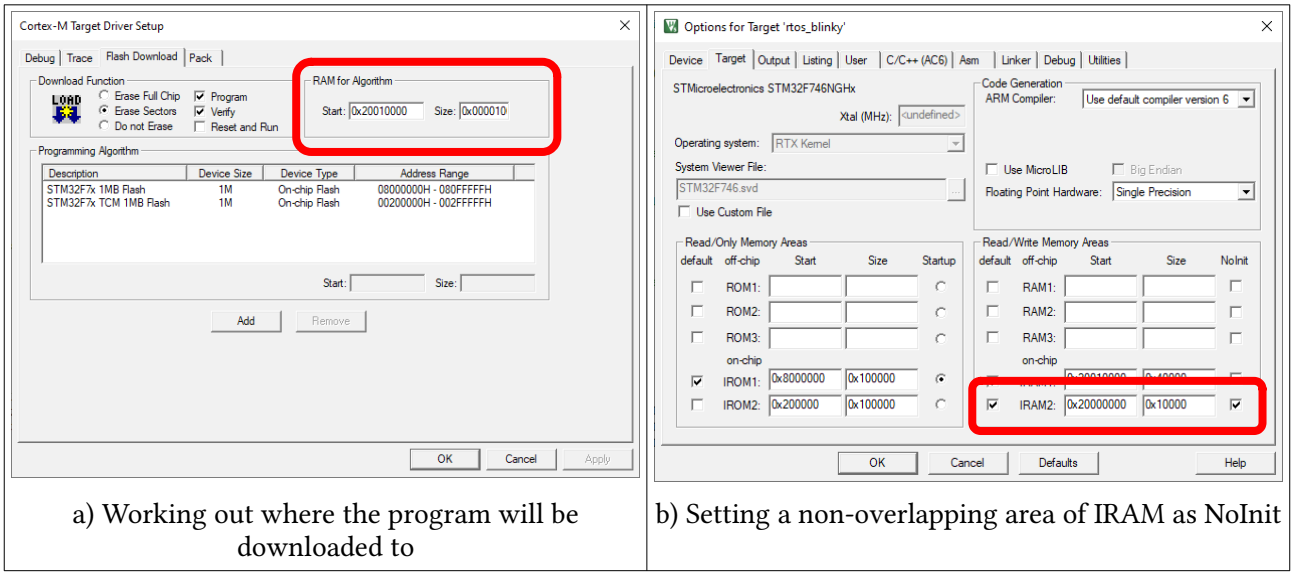
We can use the event recorder features in conjunction with the real-time operating system in uVision to help us see exactly what is happening in this example.

Although the Keil Event Recorder is enabled in this project, it is useful to understand how this works in case you wish to add this functionality to your own applications.

1. Firstly we need to select the software component **`"Compiler:Event Recorder"`** in the RTE manager and ensure we are using the **`"Source"`** variant of **`"CMSIS:RTOS2 (API):Keil RTX5"`** (see Figure 4).



2. We then need to locate event recorder in uninitialised memory otherwise the debugger will nag us about it (and it wont be able to keep recording across program resets). To do this we need to work out where our program will be written to (see Figure 5a) and then create an area of memory (IRAM) that doesn't overlap this that we then set the **`NoInit`** flag for (see Figure 5b).



a) Working out where the program will be downloaded to

b) Setting a non-overlapping area of IRAM as NoInit

Figure 5 – Setting up a memory region for Event Recorder

3. Next we need to tell the event recorder which bit of memory to use. If we right-click on **`EventRecorder.c`** and go to the **`Memory`** tab, we can set the area of zero-initialised data (See Figure 6).

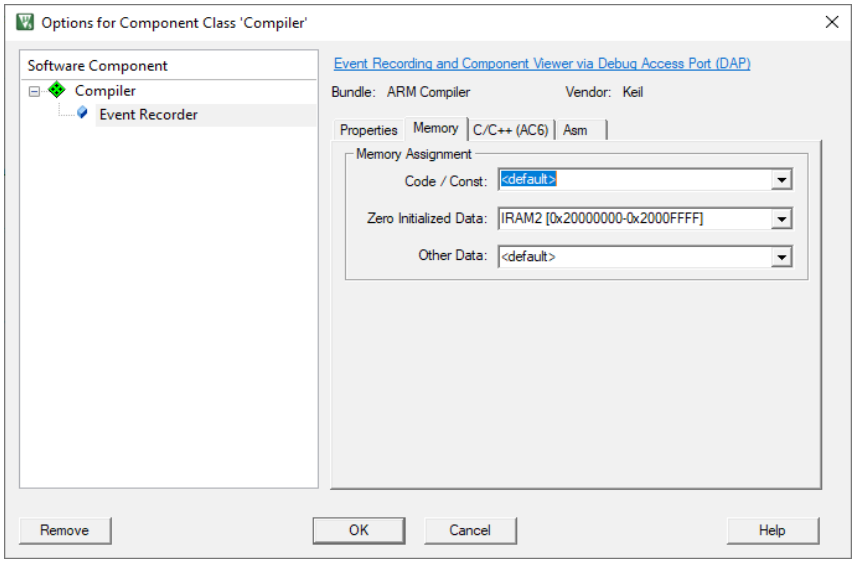


Figure 6 – Assigning Zero Initialized Data to the IRAM2 region we specified in step 2

4. Finally we need to set up the **Event Recorder Configuration** in **`RTX_Config.h`**. Here we want the event recorder to be initialised during **`osKernelInitialize`** and then to start recording. We can also specify what RTOS related events we want to record (though the default settings seem to work reasonably well). These are shown in Figure 7, below.

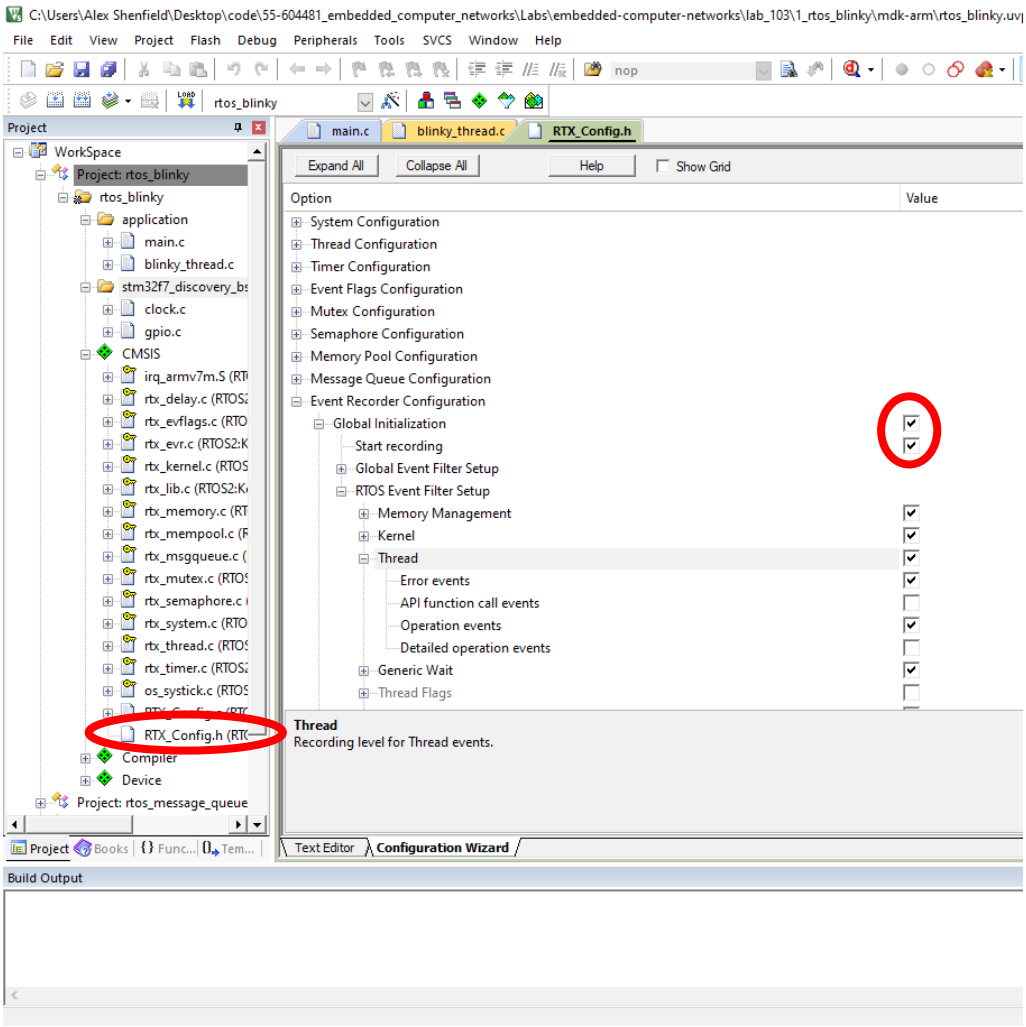


Figure 7 – Event recorder configuration for the RTOS

Now you can see which threads are in which state in the **View > Watch Windows > RTX RTOS** window (see Figure 8). You can also use the **View > Analysis Windows > System Analyzer** to see the time spent in each thread and the actual time slices between them (see Figure 9). As all three of our LED threads are of the same priority, you can see that the RTOS scheduler evenly alternates between them.

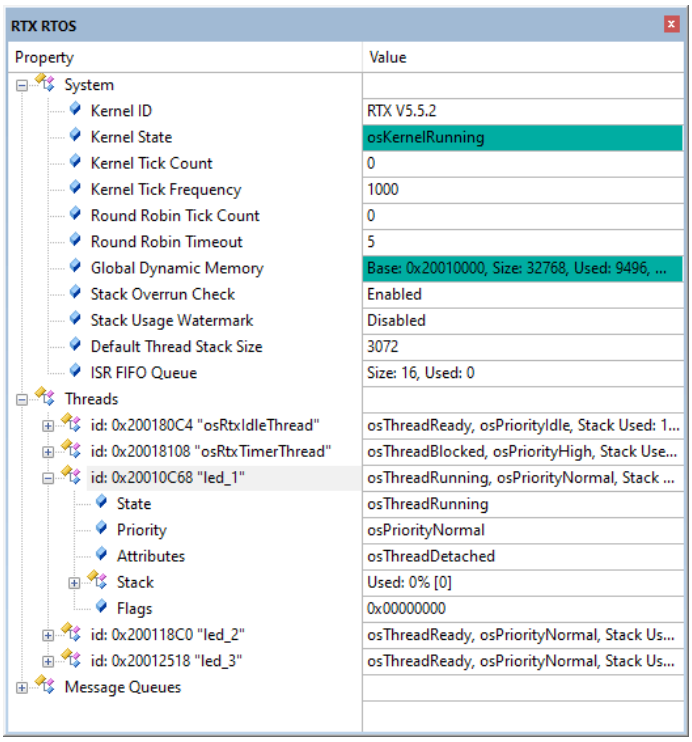


Figure 8 – The RTX RTOS watch window

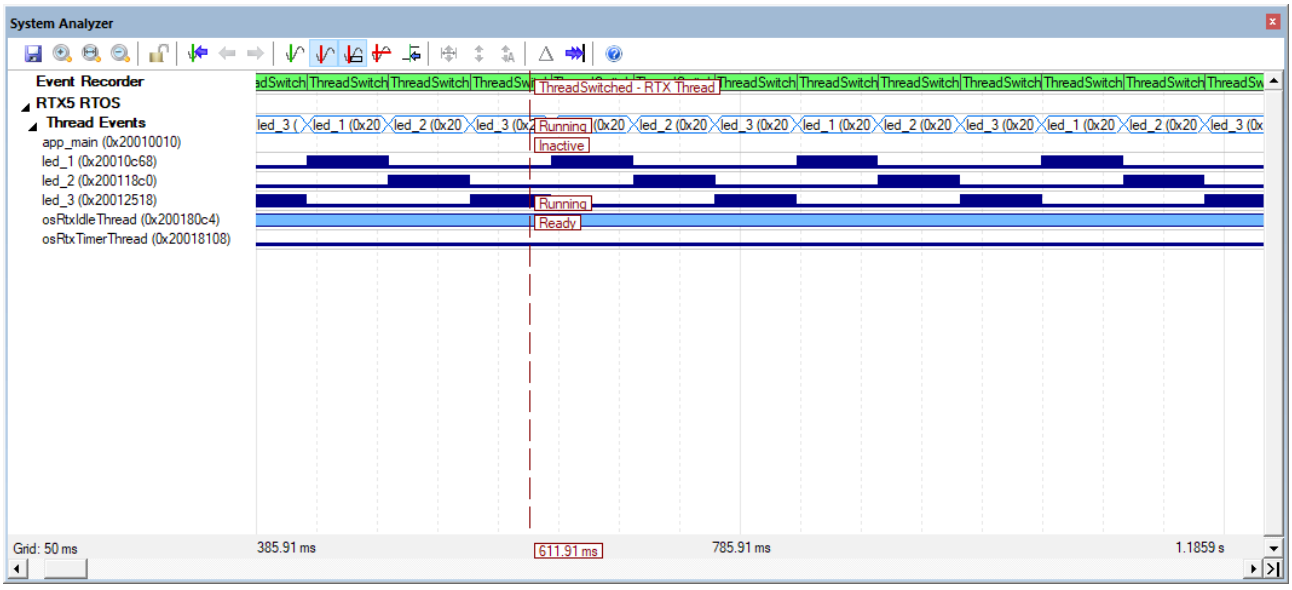


Figure 9 – The System Analyzer window

You can alter the priorities of the threads in the **`blinky_thread.c`** file. Try changing **`led_3_thread()`** to **`osPriorityAboveNormal`** in its thread attributes:

```
// led thread 3 attributes
static const osThreadAttr_t thread_3_attr =
{
  .name = "led_3",
  .priority = osPriorityAboveNormal,
};
```

What happens to the LEDs?

Why?

You should be able to see that, in the RTX RTOS watch window shown in Figure 8, the osRtxIdleThread never runs. This is because there is no point in our application (that is, 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 osRtxIdleThread now?

What about the trace in the System Analyzer window?

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 10).

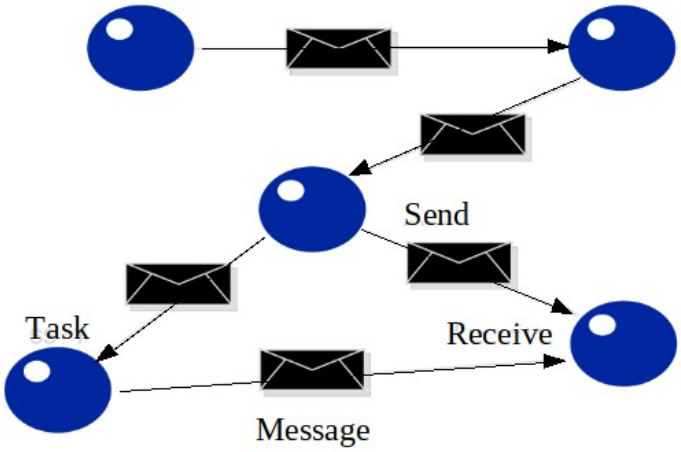


Figure 10 – Threaded application design

One of the powerful features of the CMSIS RTOS2 abstraction layer is its 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-RTOS2 provides a message queue construct to allow data to be transferred asynchronously between threads (see Figure 11). These can be used as either simple message queues or extended message queues. The key difference is that simple message queues are designed to transfer integer values (or pointers), whilst extended message queues can be easily used with larger blocks of data (such as structs).

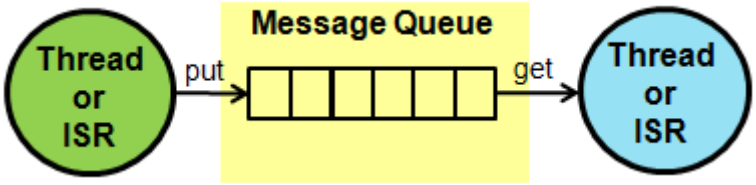


Figure 11 – Message queue

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 message queue. Figure 12 shows our application architecture.

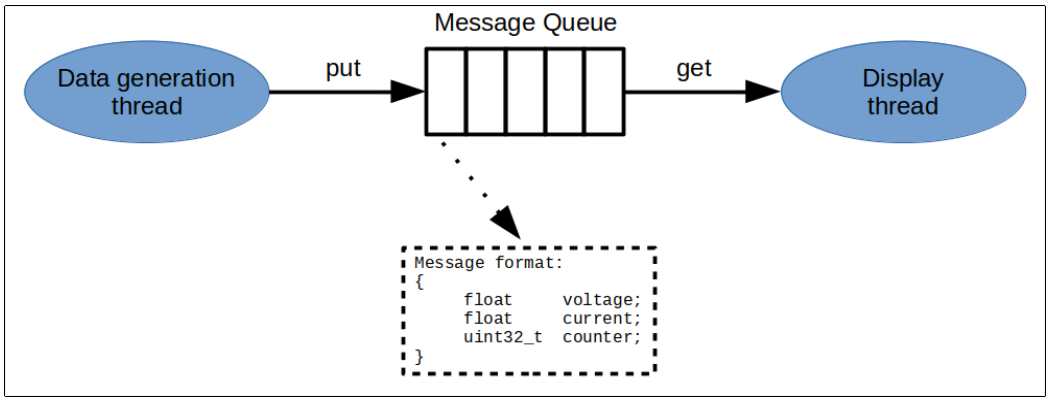


Figure 12 – Our application architecture

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 providing an external **`app_initialise()`** function rather than pulling in the **`app_main`** thread – this allows us to improve the code separation).

Code listing 3:

```
/*
* main.c
*
* this is the main rtos application
*
* this is deliberately written to be as generic as possible so we can reuse
* it - we will have all our thread specific stuff in a separate file
*
* author:    Dr. Alex Shenfield
* date:      22/10/2021
* purpose:   55-604481 embedded computer networks - lab 103
*/

// include the basic headers for the hal drivers and cmsis-rtos2
#include "stm32f7xx_hal.h"
#include "cmsis_os2.h"

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

// EXTERNAL DECLARATIONS

// this is our main application thread initialisation function - it is defined
// elsewhere (in our app_main.c file)
extern void app_initialise(void);

// CODE

// this is the main method
int main()
{
  // 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

  // initialise the real time kernel
  osKernelInitialize();

  // create application main thread (defined elsewhere)
  app_initialise();

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

Code listing 4 shows our **`app_main.c`** file. This is where we are creating our main application thread and then using it to spawn our worker threads. We also create the message queue object in our main application thread to allow us to pass data between the worker threads.

Code listing 4:

```
/*
* app_main.c
*
* this is where we create the main rtos application thread and kick everything
* off
*
* author:    Dr. Alex Shenfield
* date:      22/10/2021
* purpose:   55-604481 embedded computer networks - lab 103
*/

// include the c standard io library
#include <stdio.h>

// include the basic headers for the hal drivers and cmsis-rtos2
#include "stm32f7xx_hal.h"
#include "cmsis_os2.h"

// include my rtos objects
#include "rtos_objects.h"

// RTOS DEFINES

// define the externed function prototypes, thread ids and attributes for each
// of our worker threads

// data generation thread
osThreadId_t gen_thread;
extern void data_generation(void *argument);
static const osThreadAttr_t data_gen_thread_attr =
{
  .name = "data_generation",
  .priority = osPriorityNormal,
};

// data display thread
osThreadId_t display_thread;
extern void data_display(void *argument);
static const osThreadAttr_t data_display_thread_attr =
{
  .name = "data_display",
  .priority = osPriorityNormal,
};

// define the message queue id and attributes

// my message queue
osMessageQueueId_t m_messages;
static const osMessageQueueAttr_t msq_q_attr =
{
  .name = "my_messages",
};
```

```
// OTHER DEFINES

// stdout_init is defined in our configuration file (as part of the ARM
// compiler user code template
extern int stdout_init(void);

// APPLICATION MAIN THREAD

// provide an increased stack size to the main application thread (to help deal
// with the uart printf redirection)
static const osThreadAttr_t app_main_attr =
{
  .stack_size = 8192U
};

// application main thread - initialise general peripherals, create the message
// queue, and start the worker threads
void app_main(void *argument)
{
  // initialise the uart
  stdout_init();

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

  // create the message queue
  m_messages = osMessageQueueNew(16, sizeof(message_t), &msq_q_attr);

  // create the threads
  gen_thread = osThreadNew(data_generation, NULL, &data_gen_thread_attr);
  display_thread = osThreadNew(data_display, NULL, &data_display_thread_attr);
}

// initialise the application (by creating the main thread)
void app_initialise(void)
{
  osThreadNew(app_main, NULL, &app_main_attr);
}
```

Code listing 5 shows the **`rtos_objects.h`** header file for our application. This is where we specify the type definition for our **`message_t`** data structure and the id for the message queue. This data structure is what we are going to use to store our data in and what we are going to stuff into the message queue to exchange this data between threads.

Code listing 5:

```
/*
* rtos_objects.h
*
* this is the header file containing the rtos objects for the rtos mail queue
* application
*
* author:    Dr. Alex Shenfield
* date:      22/10/2021
* purpose:   55-604481 embedded computer networks - lab 103
*/

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

// include the header file for basic data types and the cmsis-rtos2 api
#include <stdint.h>
#include "cmsis_os2.h"

// create the objects to use in our rtos applications to pass data

// my message queue
extern osMessageQueueId_t m_messages;

// message queue data structure
typedef struct
{
  float     voltage;
  float     current;
  uint32_t  counter;
}
message_t;

#endif // RTOS_OBJ_H
```

Code listing 6 shows our data generation thread. This **`data_thread`** function generates the fake data, fills the **`message_t`** data structure, and then puts it into the message queue. This is then picked up by our display thread (see code listing 7).

Code listing 6:

```
/*
* 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:      22/10/2021
* purpose:   55-604481 embedded computer networks - lab 103
*/

// include the c standard io library
#include <stdio.h>

// include cmsis_os for the rtos api
#include "cmsis_os2.h"

// include my rtos objects
#include "rtos_objects.h"

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

// HARDWARE DEFINES

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

```
// ACTUAL THREAD

// data generation thread - create some random data and stuff it in a message
// queue
void data_generation(void const *argument)
{
  // print a status message
  printf("data generation thread up and running ...\r\n");

  // note - generally this (using a shared resource - i.e. the usart for printf
  // - from multiple threads) is a really bad idea due to race conditions etc.
  // however, we can (probably) get away with it here as we are only doing it
  // once before the data display thread is properly running :)
  //
  // really we should protect access to the usart here with a mutex around the
  // printf statement(s)

  // set up the gpio for the led and initialise the random number generator
  init_gpio(led1, OUTPUT);
  init_random();

  // set up our counter and initialise a message container
  uint32_t i = 0;
  message_t msg;

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

    // toggle led
    toggle_gpio(led1);

    // generate our fake data
    i++;
    msg.counter = i;
    msg.current = (1.0f / (random * i));
    msg.voltage = (5.0f / (random * i));

    // put the data in the message queue and wait for one second (note:
    // osWaitForever here means "wait as long as it takes until there is space
    // in the message queue)
    osMessageQueuePut(m_messages, &msg, osPriorityNormal, osWaitForever);
    osDelay(1000);
  }
}
```

Code listing 7 shows our data display thread. The **`data_thread`** function from **`data_generation_thread.c`** generates some fake data and puts it in the message queue, the **`display_thread`** function then grabs the data from the message queue (when available) and prints it to the UART. In this example we are using the UART to send the data over the VCOM serial port to our terminal program (e.g. TeraTerm).

Code listing 7:

```
/*
 * 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:      22/10/2021
 * purpose:   55-604481 embedded computer networks - lab 103
 */

// include the c standard io library
#include <stdio.h>

// include cmsis_os for the rtos api
#include "cmsis_os2.h"

// include my rtos objects
#include "rtos_objects.h"

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

```
// ACTUAL THREADS

// data display thread - pull the data out of the mail queue and print it to
// the uart
void data_display(void const *argument)
{
  // print a status message
  printf("display thread up and running ...\r\n");

  // initialise our message object and the message priority
  message_t msg;
  uint8_t   priority;

  // infinite loop getting out fake data ...
  while(1)
  {
    // get the next message in the queue
    osStatus_t status = osMessageQueueGet(m_messages, &msg, &priority, osWaitForever);
    if(status == osOK)
    {
      // print it to the serial terminal
      printf("\r\n");
      printf("Voltage: %.2f V\r\n", msg.voltage);
      printf("Current: %.2f A\r\n", msg.current);
      printf("Number of cycles: %u\r\n", msg.counter);
    }
  }
}
```

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 e.g. using TeraTerm). You should see output that looks like Figure 13.

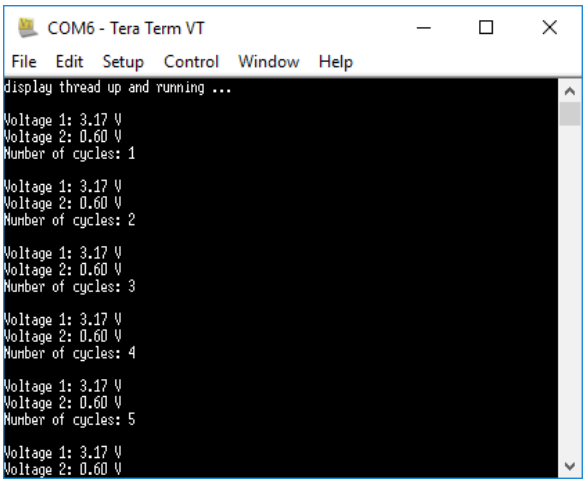


Figure 13 – Teraterm output from out RTOS message queue program

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 **`message_t`** type to simulate sending additional data between threads.

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 14) and adapting the project from the previous example.

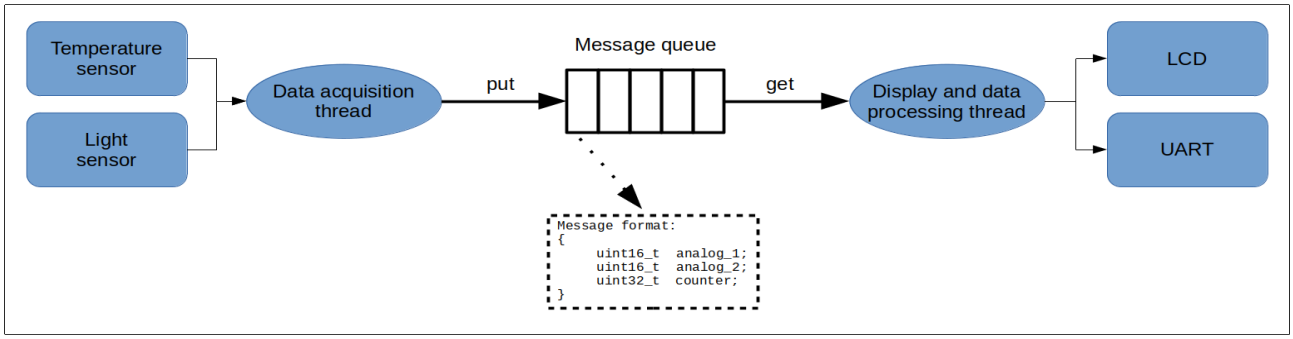


Figure 14 – Our application architecture for the real-world data acquisition and display example

Figure 15 shows an illustration of what this system may look like[3].

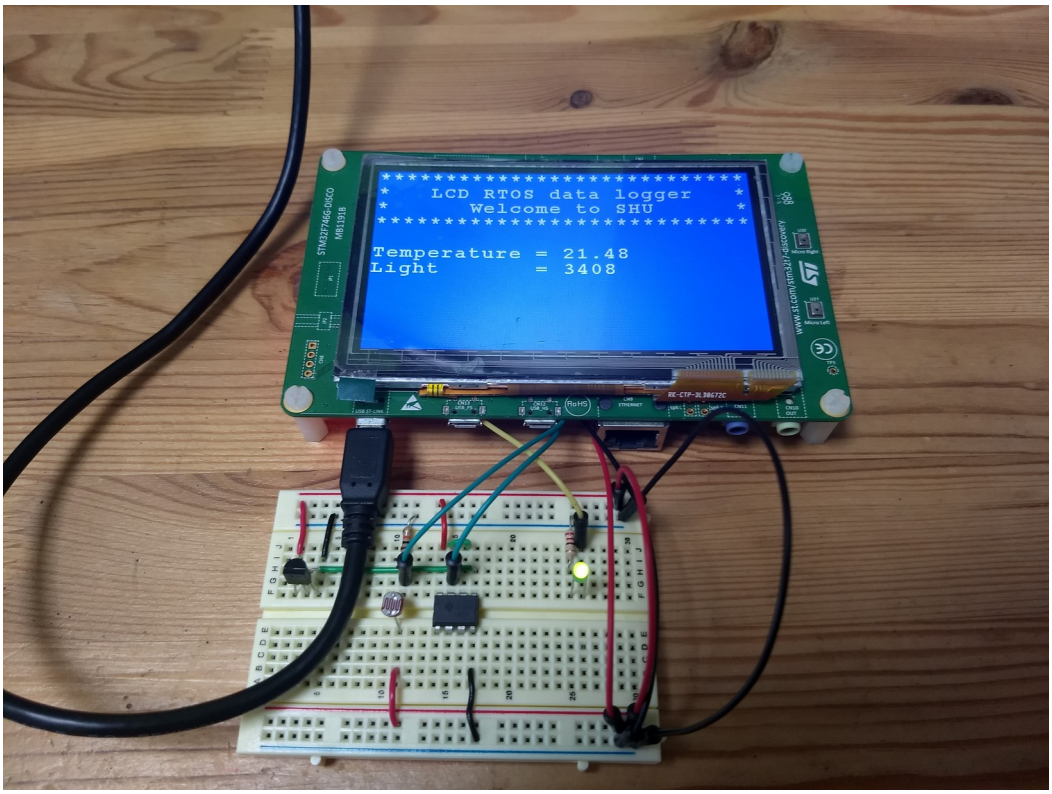


Figure 15 – Working data acquisition and display system

3 Note, you can see that this circuit is using the op amp buffer described in lab 102 to improve the ADC response.

I recommend developing this application using the following steps:

1. Make a clone of the previous project by copying the **`inc/`**, **`mdk-arm/`**, and **`src/`** folders from the **`2_message_queues/`** folder and pasting them into the **`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`**). You can remove the **`random_numbers.c`** file if you want!

3. You now need to add the drivers for the LCD screen. The simplest way to do this is through the RTE manager (as described in lab 101 – task 5, where we used it to add support for sending data over the UART to a serial terminal). Figure 16 shows the selection of components from the Board Support pack in the RTE manager.

   Note:
   a) the selected board (STM32F746-Discovery)
   b) that the validation output window is reporting additional software components required – fortunately we can just click **Resolve** (highlighted), and the RTE manager will include these for us.

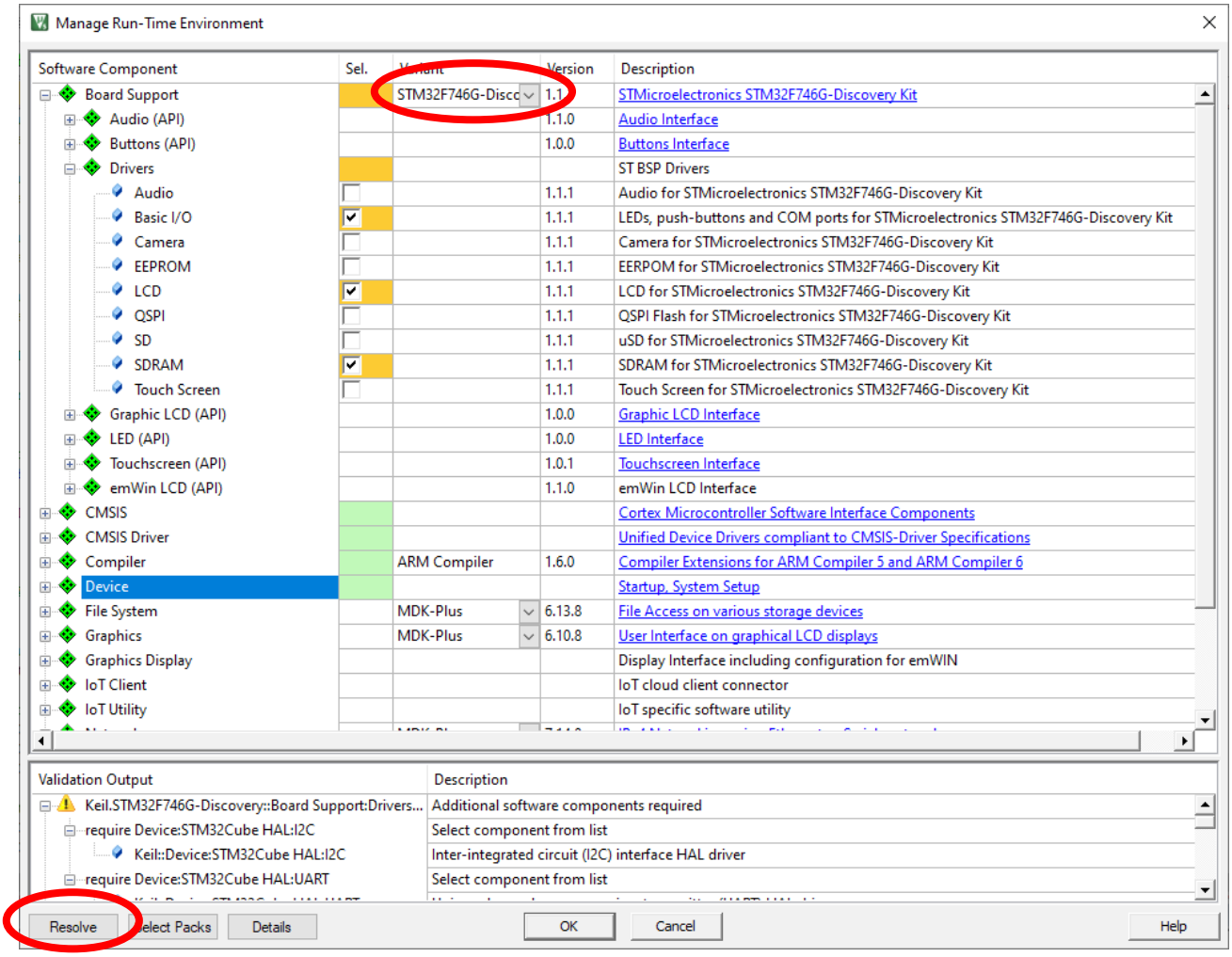


Figure 16 – Using the RTE manager to install board support packages for the LCD

4. Adapt the code from lab 102, task 2 to initialise the LCD screen, read the ADC values and convert them into formatted strings for display on the LCD. Then integrate this into the data display thread.

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 **`app_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`**).

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 create a simple read-eval-print loop (REPL) terminal application 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 use the USART driver from the CMSIS Driver pack which implements non-blocking functions to handle data transfer. When data transfer is complete, the driver triggers a callback function (with specific event codes) that can be used to handle deciding what to do next (e.g. sending more data or processing the data that has been received). Figure 17 shows the basic communication flow of sending data.

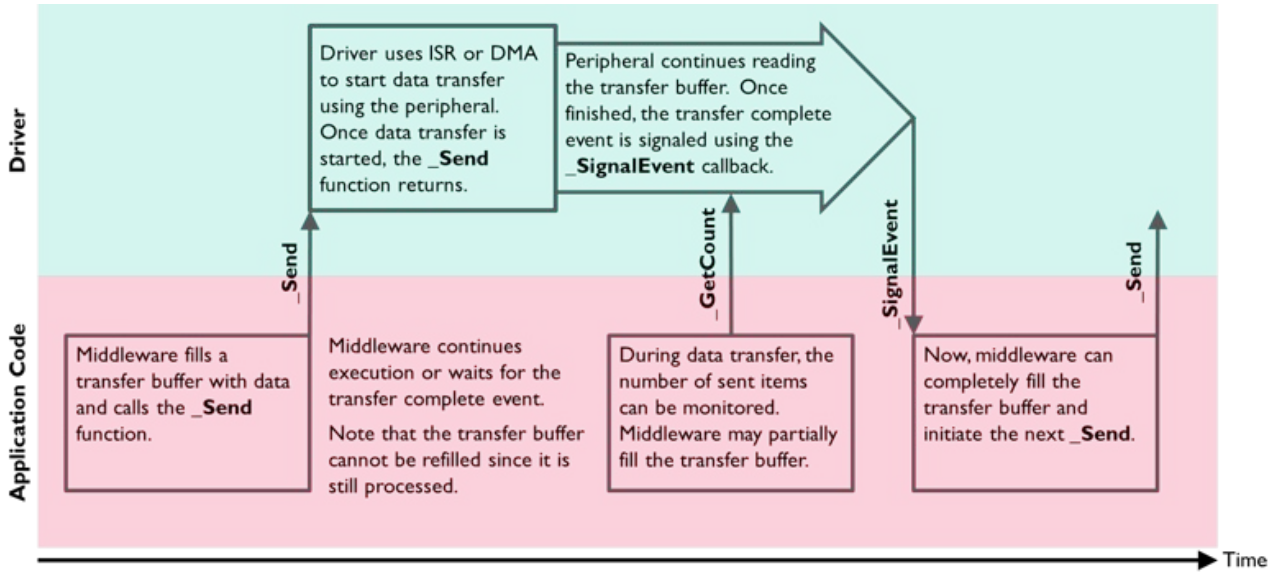


Figure 17 – CMSIS Driver communication flow for sending data[4]

4 Taken from https://www.keil.com/pack/doc/CMSIS/Driver/html/theoryOperation.html

Figure 18 shows a high level block diagram of this RTOS UART application. As you can see from Figure 17, the CMSIS driver code uses some sort of interrupt to trigger the callback function when data transfer is complete or data is available. In our application this callback function then uses event signal flags to signal the REPL thread to let it know that either data has been received or that the data was sent OK. It can then progress to the next part of the loop.

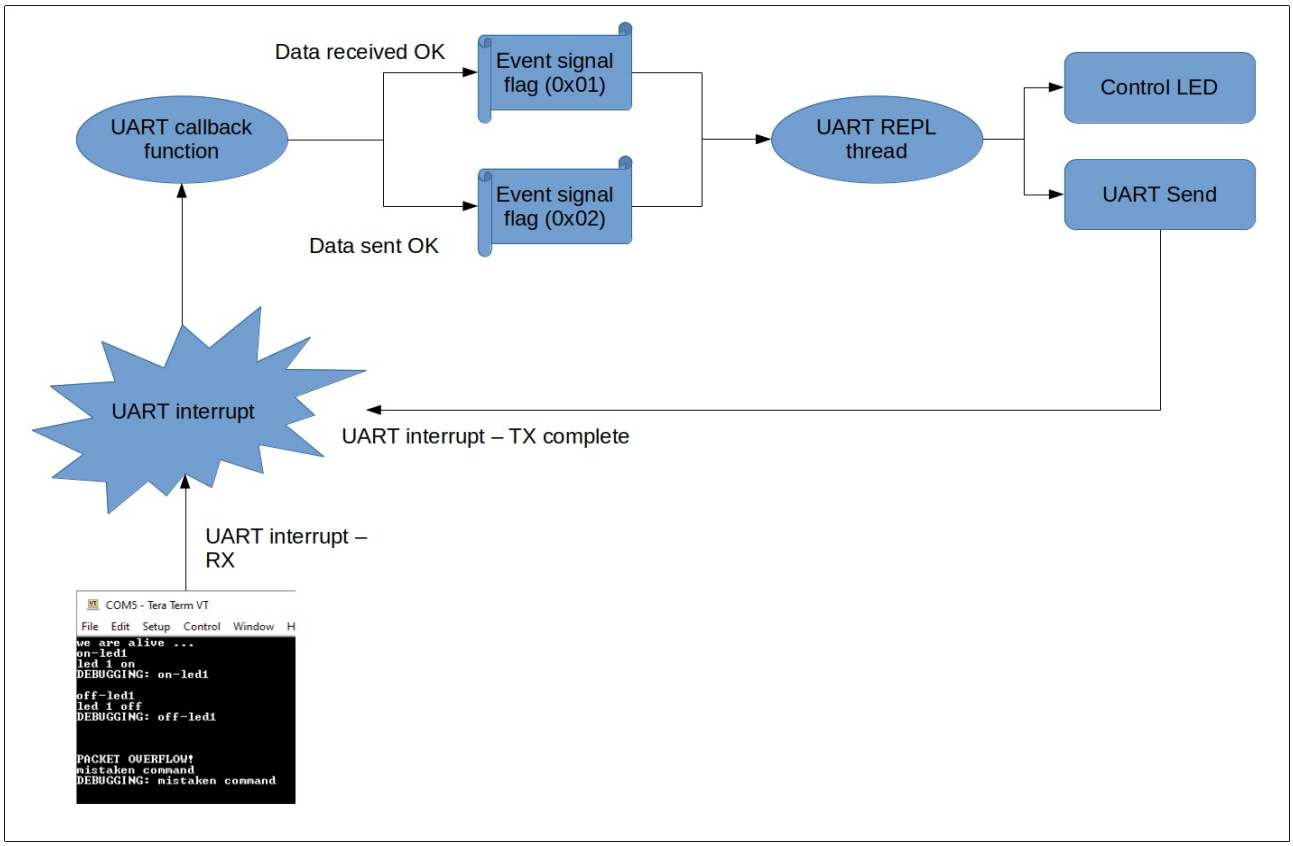


Figure 18 – The UART application architecture

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

Code listing 7 shows part of the app_main.c file where we define an event flag to use in notifying our main worker thread that events have occurred. The rest of the app_main.c file is much the same as in the previous examples.

Code listing 7

```
// define the event flag id and attributes

// we are enabling an event flag for notification of usart events
osEventFlagsId_t usart_flag;
static const osEventFlagsAttr_t usart_flag_attr =
{
  .name = "USART_event",
};

// <snip>

// application main thread - create the event flag for uart events and start
// the worker thread
void app_main(void *argument)
{
  // create the usart event flag
  usart_flag = osEventFlagsNew(&usart_flag_attr);

  // create the thread
  usart_thread = osThreadNew(my_usart_thread, NULL, &usart_thread_attr);
}

// <snip>
```

In much the same way as we did for the message queue id in the message queue example (in task 2), we have defined the osEventFlagsId_t object in our rtos_objects.h file so that we can pull it in from our repl_thread.c file and use it in the worker thread.

Code listing 8 shows our repl_**thread.c** file. This contains our my_usart_thread**()** function (which does all the data processing) and our my_usart_callback() function (which is triggered by the CMSIS USART driver in response to an interrupt being generated).

Code listing 8

```
/*
* repl_thread.c
*
* this is where we create and configure the main worker thread(s)
*
* in this application we are going to use the uart connection (via a terminal
* program such as teraterm) to implement a simple read-eval-print loop (repl)
* shell to accept and process a set of commands
*
* author:    Dr. Alex Shenfield
* date:      22/10/2021
* 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 basic headers for the hal drivers and cmsis-rtos2
#include "arm_compat.h"
#include "stm32f7xx_hal.h"
#include "cmsis_os2.h"

// include the CMSIS Driver for code for the USART
#include "Driver_USART.h"

// include my rtos objects
#include "rtos_objects.h"

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

// HARDWARE DEFINES

// define some leds that we can use to indicate system status
gpio_pin_t data_led    = {PB_14, GPIOB, GPIO_PIN_14};
gpio_pin_t error_led   = {PB_15, GPIOB, GPIO_PIN_15};
gpio_pin_t control_led = {PI_1,  GPIOI, GPIO_PIN_1};

// USART DEFINES

// we are using usart 1 which is defined elsewhere
extern ARM_DRIVER_USART Driver_USART1;
```

```
// USART CALLBACK HANDLER

// this is the callback which is triggered when a usart event occurs
void my_usart_callback(uint32_t event)
{
  // receive success
  uint32_t recv_mask = ARM_USART_EVENT_RECEIVE_COMPLETE  |
                       ARM_USART_EVENT_TRANSFER_COMPLETE ;

  // send success
  uint32_t send_mask = ARM_USART_EVENT_TRANSFER_COMPLETE |
                       ARM_USART_EVENT_SEND_COMPLETE;

  // if our usart receive was a success
  if(event & recv_mask)
  {
    toggle_gpio(data_led);
    osEventFlagsSet(usart_flag, 0x01);
  }

  // if our usart send / transmit was a success
  if(event & send_mask)
  {
    toggle_gpio(data_led);
    osEventFlagsSet(usart_flag, 0x02);
  }

  //
  // error handling ...
  //

  // rx overflow / tx underflow error
  if(event & (ARM_USART_EVENT_RX_OVERFLOW | ARM_USART_EVENT_TX_UNDERFLOW))
  {
    write_gpio(error_led, HIGH);
    __breakpoint(0);
  }
}
```

```
// WORKER THREAD

// usart thread function
void my_usart_thread(void *argument)
{
  // create a packet array to use as a buffer to build up our command string
  uint8_t packet[128];
  int i = 0;

  // initialise our leds
  init_gpio(data_led, OUTPUT);
  init_gpio(error_led, OUTPUT);
  init_gpio(control_led, OUTPUT);

  // usart configuration ...

  // get a pointer to the usart driver object
  static ARM_DRIVER_USART * usart_driver = &Driver_USART1;

  // initialise and configure the usart driver (running at 9600bps)
  usart_driver->Initialize(my_usart_callback);
  usart_driver->PowerControl(ARM_POWER_FULL);
  usart_driver->Control(ARM_USART_MODE_ASYNCHRONOUS |
                        ARM_USART_DATA_BITS_8 |
                        ARM_USART_PARITY_NONE |
                        ARM_USART_STOP_BITS_1 |
                        ARM_USART_FLOW_CONTROL_NONE, 9600);

  // set up rx and tx lines
  usart_driver->Control(ARM_USART_CONTROL_TX, 1);
  usart_driver->Control(ARM_USART_CONTROL_RX, 1);

  // initial welcome message ...

  // print a status message
  usart_driver->Send("we are alive ...\r\n", 19);

  // wait for the usart_flag event flag 0x02 (i.e. we have sent the data
  // successfully)
  osEventFlagsWait(usart_flag, 0x02, osFlagsWaitAny, osWaitForever);

  // main thread logic ...

  // main thread loop
  while(1)
  {
    // read a character from the usart and wait for the recv operation to
    // complete successfully (i.e. we get the usart_event flag 0x01)
    char c;
    usart_driver->Receive(&c, 1);
    osEventFlagsWait(usart_flag, 0x01, osFlagsWaitAny, osWaitForever);
```

```
    // if the last character we read was a newline, then process the contents
    // of the packet buffer
    if(c == '\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)
      {
        // turn the control led on
        write_gpio(control_led, HIGH);

        // display a status message
        usart_driver->Send("led 1 on\r\n", 11);
        osEventFlagsWait(usart_flag, 0x02, osFlagsWaitAny, osWaitForever);
      }
      if(strcmp((char*)packet, "off-led1\r") == 0)
      {
        // turn the control led on
        write_gpio(control_led, LOW);

        // display a status message
        usart_driver->Send("led 1 off\r\n", 12);
        osEventFlagsWait(usart_flag, 0x02, osFlagsWaitAny, osWaitForever);
      }

      // zero the packet index
      i = 0;
    }
    else
    {
      // if there is space left in the buffer, then add the character -
      // otherwise print an error message and reset the packet index to start
      // again
      if(i < 128)
      {
        packet[i] = c;
        i++;
      }
      else
      {
        usart_driver->Send("\r\nPACKET OVERFLOW!\r\n", 21);
        osEventFlagsWait(usart_flag, 0x02, osFlagsWaitAny, osWaitForever);
        i = 0;
      }
    }
  }
}
```

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 19.

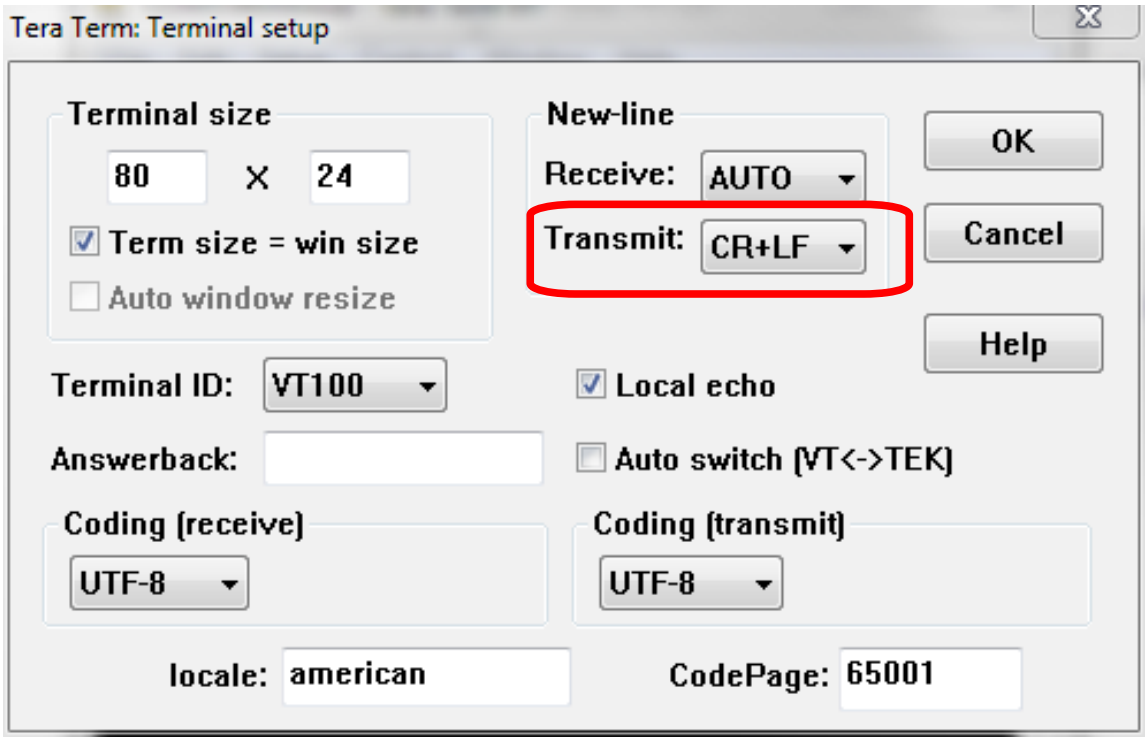


Figure 19 – Teraterm configuration

All being well, you should see a welcome message and be able to send commands to the system (see Figure 20).

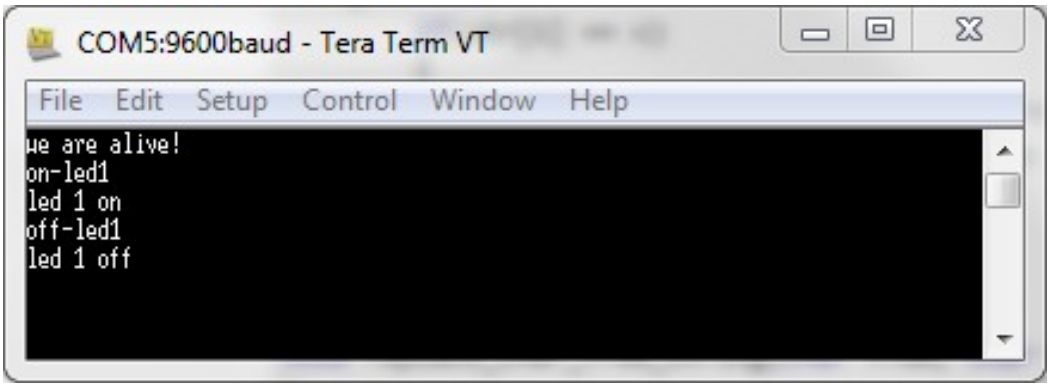
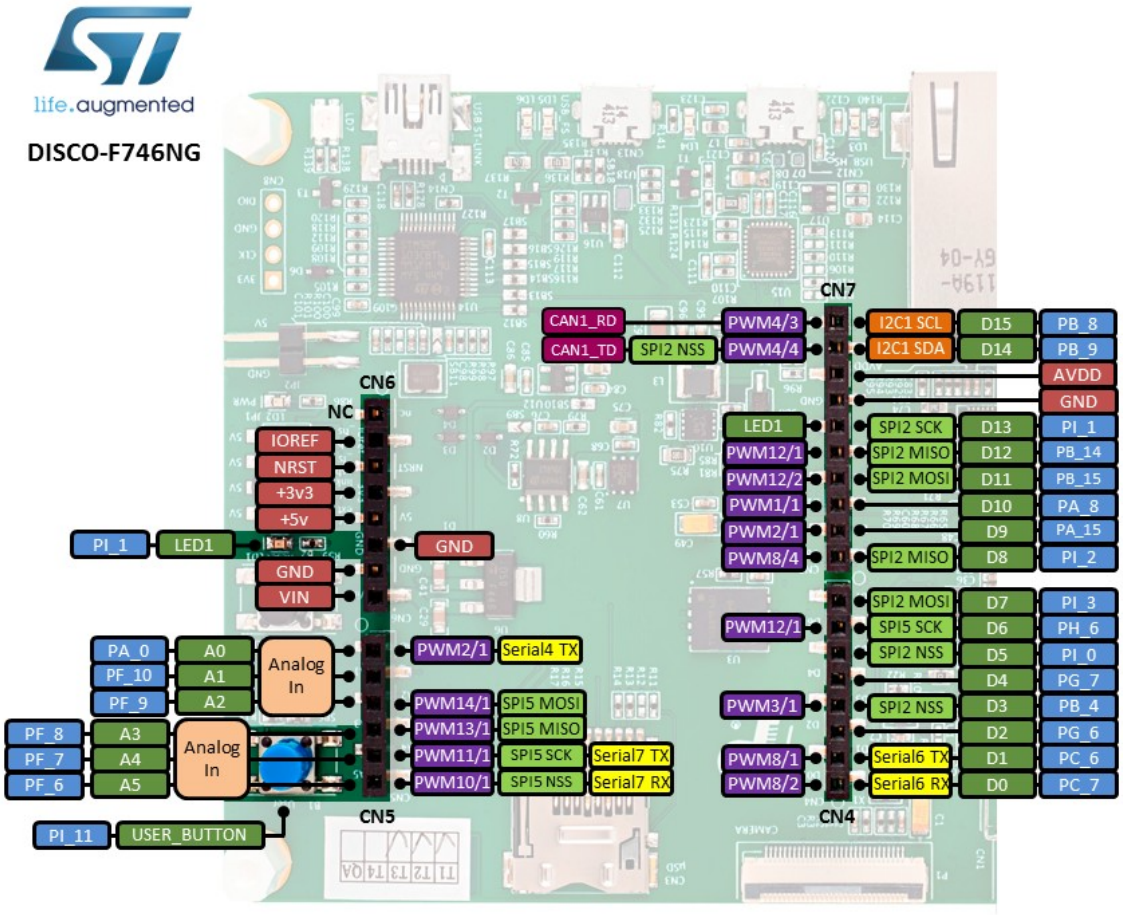


Figure 20 – Teraterm output

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