Sheffield Hallam University
Department of Engineering
BEng (Hons) Computer Systems Engineering
BEng (Hons) Electrical and Electronic Engineering

Activity ID		Activity Title			Laboratory Room No.	Level
Lab 101		A gentle introduction to the STM32F7 discovery board			4302	6
Term	**Duration [hrs]**	**Group Size**	**Max Total Students**	**Date of approval/ review**	**Lead Academic**	
1	4	1	25	09-21	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

A gentle introduction to the STM32F7 discovery board and expansion kit

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 aims to introduce you both to some basic electronics principles and to the STM32F7 discovery board (as well as the ST HAL driver library and the rest of the hardware we will be using throughout this year).

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

Task 1

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

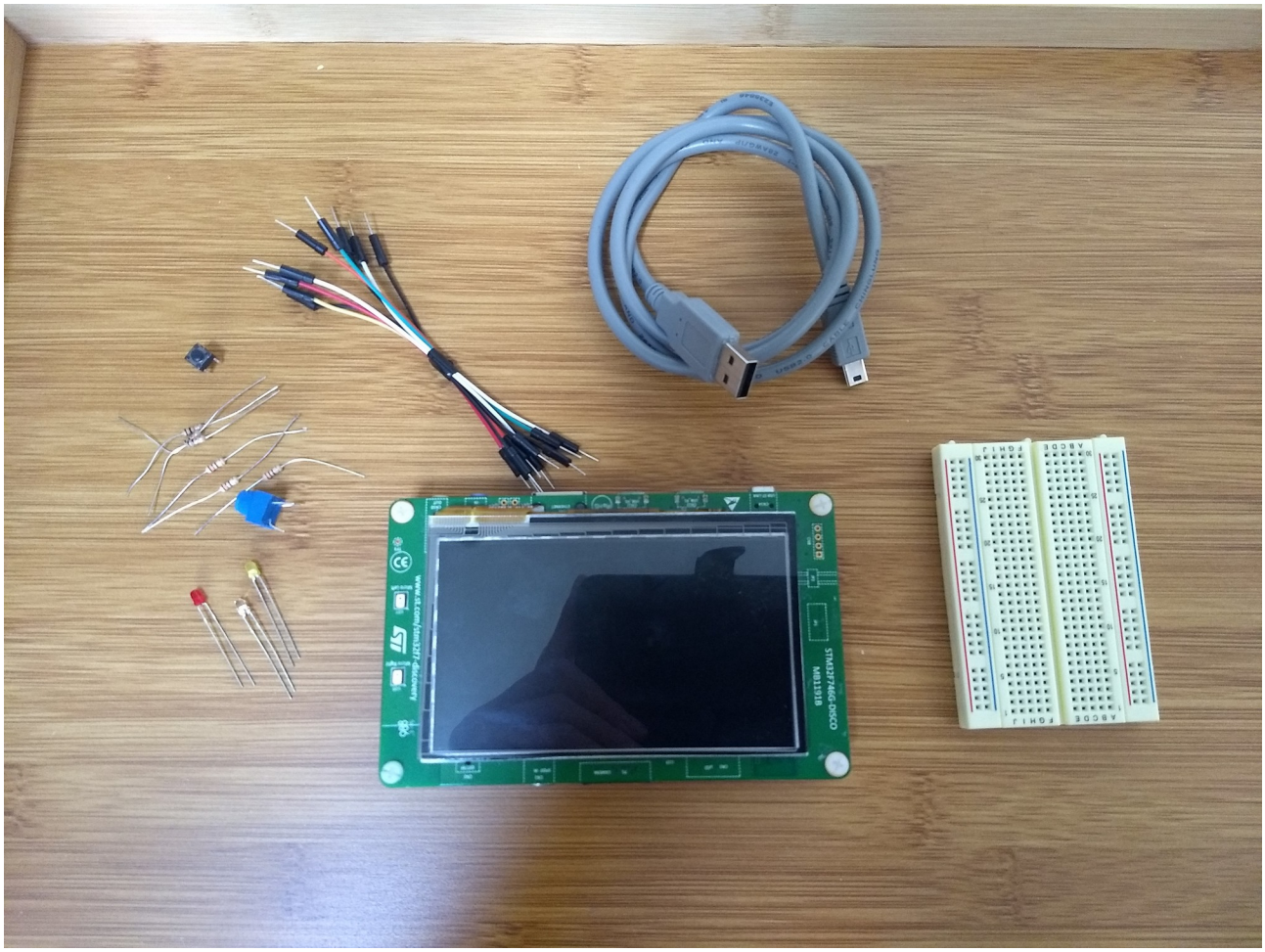


Figure 1 – The necessary equipment for this lab (don't worry if you have LEDs of a different colour or different potentiometers)

2. To connect the breadboard to the STM32F7 discovery board you will need some male to male jumper wires. Figure 2 shows some solid core jumper wire.

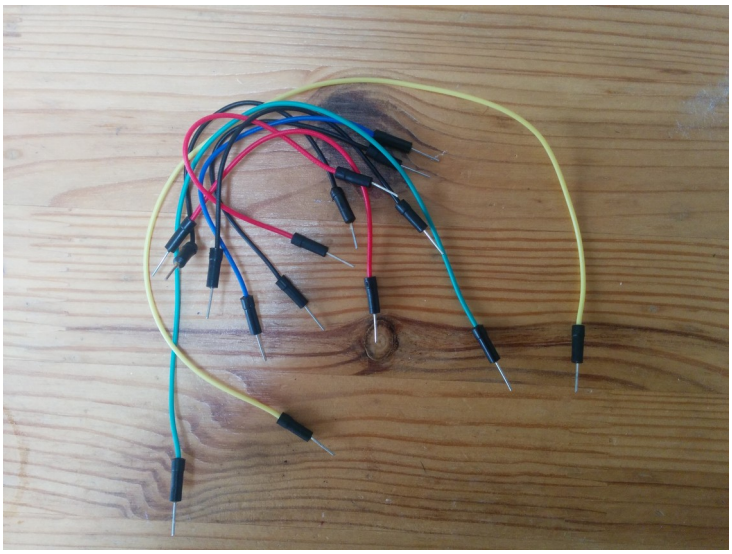


Figure 2 – An example of jumper wires

3. Once you have made a few 3 inch jumper wires it is time to wire up the breadboard. We will connect a 220 ohm resistor from the anode (long leg) of the LED to the positive supply rail on the breadboard, and connect the positive supply rail on the breadboard to PI_1 on the STM32F7 discovery board (marked as D13 – see Appendix A for the STM32F7 discovery board pin outs). We will then wire up the cathode (short leg) on the LED to the negative supply rail on the breadboard and the negative supply on the breadboard to the ground pin on the STM32F7 discovery board. Figure 3 shows the completed and wired up circuit – if you have any questions about this circuit ask a member of staff.

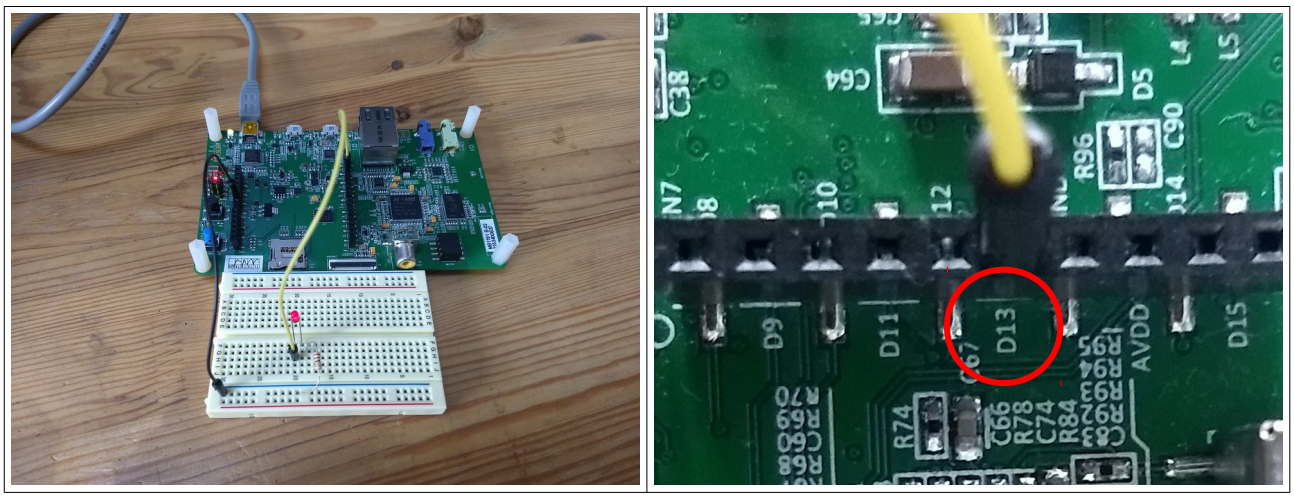


Figure 3 – The completed circuit

4. We now need to write some code to flash this LED. Figure 4 shows the program logic in pseudo-code, and code listing 1 shows the complete program. A Keil uVision 5 project is available to download from GitHub.

```
Procedure BLINK:

    while not ready:
        initialise GPIO pin (PI_1)
    end while

    toggle LED pin (PI_1)
    wait 1 second
    repeat

end Procedure BLINK
```

Figure 4 – Pseudo code for the “blink” program

We are going to use this breadboard method of prototyping throughout the rest of the laboratory exercises this year, and for the circuits you will design for your assignment.

Code listing 1:

```
/*
 * main.c
 *
 * this is the main blinky application with no external dependencies apart from
 * the stm32f7xx hal libraries
 *
 * author:  Dr. Alex Shenfield
 * date:    28/09/2020
 * purpose: 55-604481 embedded computer networks : lab 101
 */

// include the hal drivers
#include "stm32f7xx_hal.h"

// map the led to GPIO PI1 (this is the inbuilt led by the reset button)
#define LED_PIN      GPIO_PIN_1
#define LED_PORT     GPIOI
#define LED_CLK_ENABLE() __GPIOI_CLK_ENABLE()

// declare our utility functions
void init_sysclk_216MHz(void);
void configure_gpio(void);

// 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();

  // initialise the gpio pins
  configure_gpio();

  // loop forever ...
  while(1)
  {
    // toggle the led on the gpio pin
    HAL_GPIO_TogglePin(LED_PORT, LED_PIN);

    // wait for 1 second
    HAL_Delay(1000);
  }
}
```

```
// CLOCK

// configure the stm32f7 discovery board to operate at 216MHz
void init_sysclk_216MHz(void)
{
  // set up initialisation structures for the oscillator and the clock
  RCC_OscInitTypeDef rcc_oscillator_config;
  RCC_ClkInitTypeDef rcc_clock_config;

  // call macros to configure the correct power settings on the chip
  __HAL_RCC_PWR_CLK_ENABLE();
  __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);

  // configure the oscillator and phase-locked loops so the chip runs at 168MHz
  rcc_oscillator_config.OscillatorType = RCC_OSCILLATORTYPE_HSE;
  rcc_oscillator_config.HSEState       = RCC_HSE_ON;
  rcc_oscillator_config.HSIState       = RCC_HSI_OFF;
  rcc_oscillator_config.PLL.PLLState   = RCC_PLL_ON;
  rcc_oscillator_config.PLL.PLLSource  = RCC_PLLSOURCE_HSE;
  rcc_oscillator_config.PLL.PLLM       = 25;
  rcc_oscillator_config.PLL.PLLN       = 432;
  rcc_oscillator_config.PLL.PLLP       = RCC_PLLP_DIV2;
  rcc_oscillator_config.PLL.PLLQ       = 9;
  HAL_RCC_OscConfig(&rcc_oscillator_config);

  // activate the OverDrive to reach the 216 MHz frequency
  HAL_PWREx_EnableOverDrive();

  // configure the appropriate clock dividers
  rcc_clock_config.ClockType      = RCC_CLOCKTYPE_HCLK
                                  |RCC_CLOCKTYPE_SYSCLK
                                  |RCC_CLOCKTYPE_PCLK1
                                  |RCC_CLOCKTYPE_PCLK2;
  rcc_clock_config.SYSCLKSource   = RCC_SYSCLKSOURCE_PLLCLK;
  rcc_clock_config.AHBCLKDivider  = RCC_SYSCLK_DIV1;
  rcc_clock_config.APB1CLKDivider = RCC_HCLK_DIV4;
  rcc_clock_config.APB2CLKDivider = RCC_HCLK_DIV2;
  HAL_RCC_ClockConfig(&rcc_clock_config, FLASH_LATENCY_7);
}

// HARDWARE

// configure the gpio
void configure_gpio()
{
  // declare a gpio typedef structure that basically contains all the gpio
  // settings and properties you can use
  GPIO_InitTypeDef gpio_init_structure;

  // enable the clock for the led
  LED_CLK_ENABLE();

  // configure the gpio output on the led pin
  gpio_init_structure.Pin  = LED_PIN;
  gpio_init_structure.Mode = GPIO_MODE_OUTPUT_PP;
  gpio_init_structure.Pull = GPIO_PULLUP;
  gpio_init_structure.Speed = GPIO_SPEED_FAST;

  // complete initialisation
  HAL_GPIO_Init(LED_PORT, &gpio_init_structure);
  HAL_GPIO_WritePin(LED_PORT, LED_PIN, GPIO_PIN_RESET);
}
```

Task 2

Whilst the code we wrote in task 1 above works fine, we can see that some functions (such as the clock initialisation function to set up the system clock) we are going to be using in pretty much every project we write! So, to aid re-usability and efficiency, I have abstracted some of these commonly used functions into a board support package for some of the kit we have here at Sheffield Hallam University.

For the simple "blink" example in task 1 the use of libraries doesn't make a huge difference (as you can see from the provided project and the code in code listing 2, below) but, as code size and complexity increases, the use of separate source files and libraries to segregate functionality becomes very useful in creating readable and maintainable code (as we'll see in future weeks).

Code listing 2:

```
/*
 * main.c
 *
 * this is the main blinky application (this code is dependent on the
 * extra shu libraries such as the pinmappings list and the clock configuration
 * library)
 *
 * author:  Dr. Alex Shenfield
 * date:    28/09/2020
 * purpose: 55-604481 embedded computer networks : lab 101
 */

// include the hal drivers
#include "stm32f7xx_hal.h"

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

// map the led to GPIO PI1 (again, this is the inbuilt led)
gpio_pin_t led = {PI_1, GPIOI, GPIO_PIN_1};

// 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();

  // initialise the gpio pins
  init_gpio(led, OUTPUT);

  // loop forever ...
  while(1)
  {
    // toggle the led on the gpio pin
    toggle_gpio(led);

    // wait for 1 second
    HAL_Delay(1000);
  }
}
```

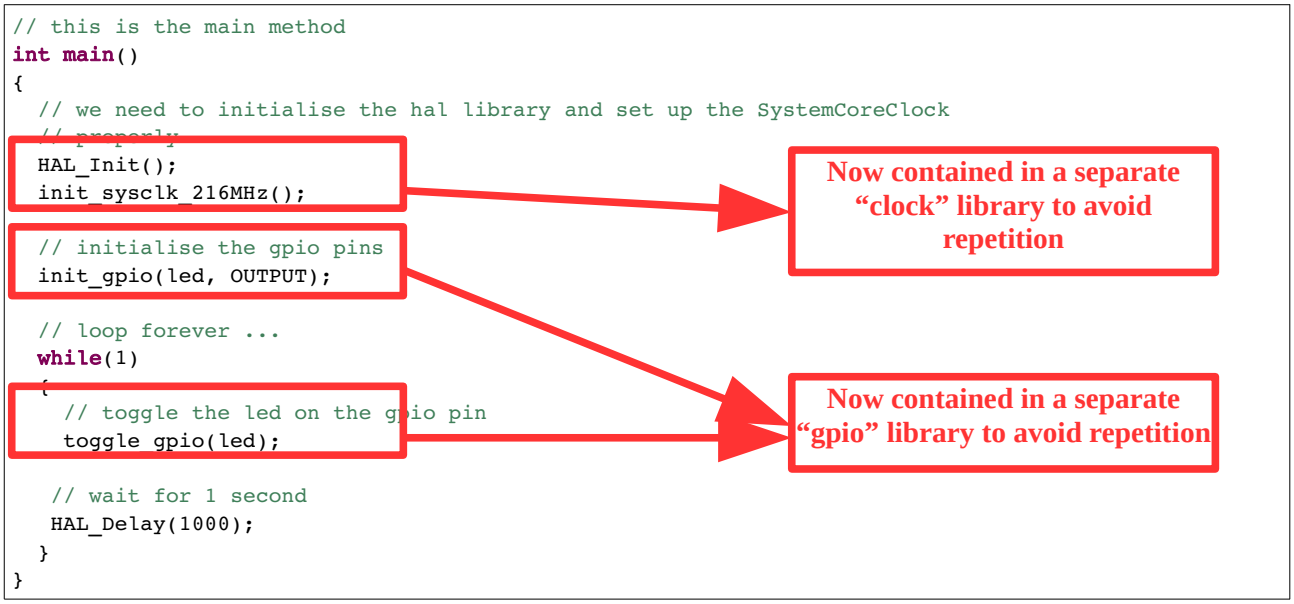
The only real difference between this code and the code from task 1 is that the commonly used functions (such as initialising the clock and initialising and using the gpio pins) have been abstracted out into libraries (so we can reuse them easily across multiple projects). In the Keil uVision 5 project (available from GitHub), these are contained in the **stm32f7_discovery_bsp_shu_kit** group. The code below illustrates this point:

```
// 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();

  // initialise the gpio pins
  init_gpio(led, OUTPUT);

  // loop forever ...
  while(1)
  {
    // toggle the led on the gpio pin
    toggle_gpio(led);

   // wait for 1 second
   HAL_Delay(1000);
  }
}
```



Test this code!

Task 3

In this section we are going to experiment with programming the STM32F7 discovery board to produce different effects with LEDs. There are two skeleton files provided in the multi-project workspace (under the "exercises" project which includes all the appropriate libraries). Note that you cannot build a project that contains two main() methods so we have to tell uVision to only include one of these source files at a time in the build process.

To do this we need to right click on the c file we want to exclude, and go to "Options for exercises – File ...". This will bring up the window shown in Figure 5 (below). To exclude the file we need to uncheck the "Include in Target Build" and "Always Build" boxes – highlighted in Figure 5.

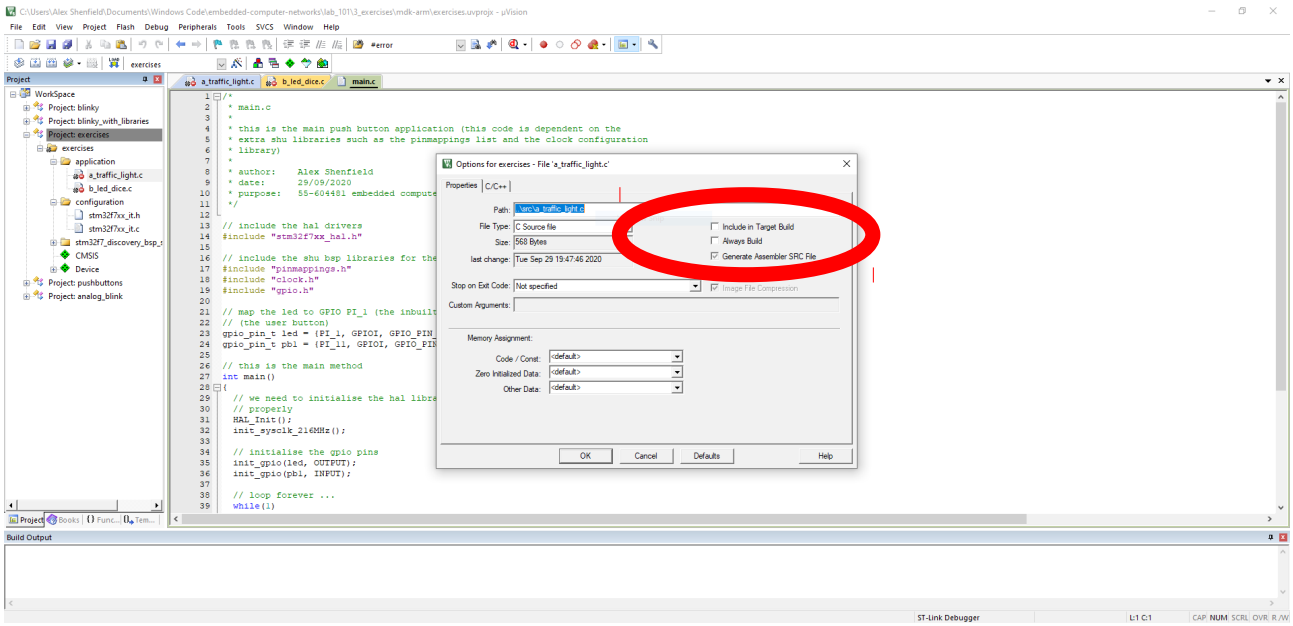


Figure 5 – Excluding a source file from the build process

Then create the circuit shown in Figure 6.

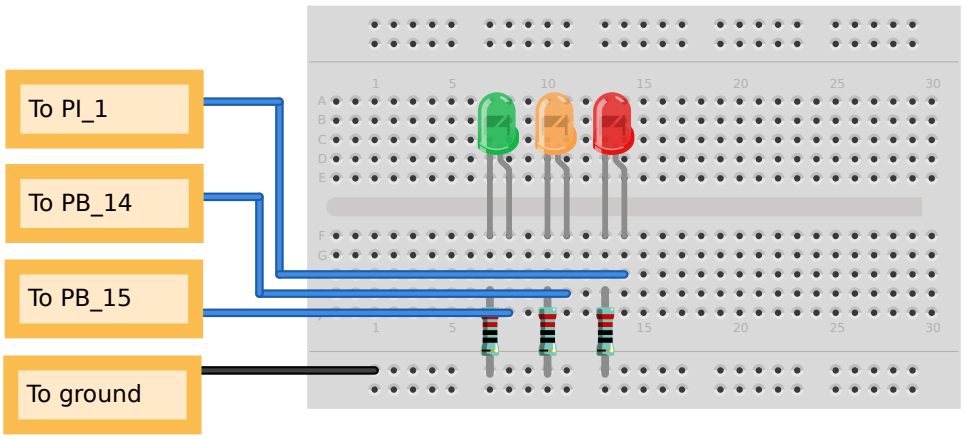


Figure 6 – Controllable LEDs

a) Now adapt the code from Task 2 to replicate the sequence of a single set of traffic lights. Make sure you tweak the timings of the LEDs so that the delays are as realistic as possible.

b) Now try and create a single LED dice (see Figure 7).

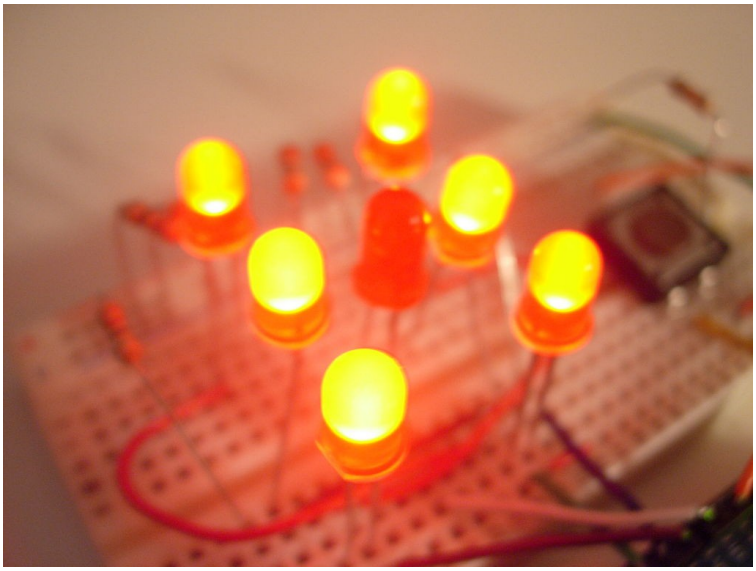


Figure 7– An LED dice

Within the SHU board support package provided, there is a set of functions to deal with the initialisation of the random number generator as well as generate the random numbers. After including the "random_numbers.h" header file and initialising the random number generator[2], you can use the code:

```
uint32_t rnd = (get_random_int() % 6) + 1;
```

to generate the random roll of the dice.

2 Details as to all the provided functions can be found in the appropriate header files. In this case, to initialise the random number generator we need to call `init_random();`

Task 4

In this task we are going to use a simple pushbutton switch to control an LED.

Firstly create the circuit shown in Figure 8, below. This wires up the two lower rails of the breadboard to the +5V and ground pins on the STM32F7 discovery board (+3.3V to the lower rail and ground to the top rail) allowing us to easily access the +3.3V supply and ground. The pushbutton switch then has one leg connected to +3.3V and the other leg connected to ground via a pull-down resistor (in this case a 10K Ohm one). The other side of this leg is connected to GPIO pin PA_8 on our STM32F7 discovery board. We then connect GPIO pin PI_1 to the LED.

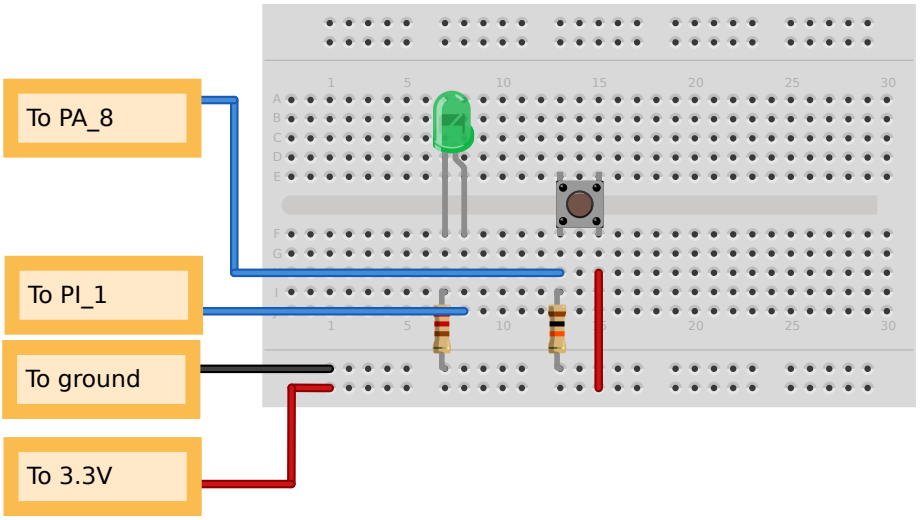


Figure 8 – The switch circuit

Q1 What is a “pull-down” resistor?

Q2 We could also have a “pull-up” resistor. What is the difference?

Q3 Why do we need “pull-up” or “pull-down” resistors?

Now we wish to write some code to control this circuit. When the pushbutton is held down we want the LED to turn on, and then when the pushbutton is released we want the LED to turn off again. See code listing 3 for a simple program to accomplish this[3].

3 Note that a Keil uVision 5 project for this exercise is available to download from GitHub.

Code listing 3:

```
/*
 * main.c
 *
 * this is the main push button application (this code is dependent on the
 * extra shu libraries such as the pinmappings list and the clock configuration
 * library)
 *
 * author:  Dr. Alex Shenfield
 * date:    29/09/2020
 * purpose: 55-604481 embedded computer networks : lab 101
 */

// include the hal drivers
#include "stm32f7xx_hal.h"

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

// map the led to GPIO PI_1 (the inbuilt led) and the push button to PA_8
// (the user supplied button)
gpio_pin_t led = {PI_1, GPIOI, GPIO_PIN_1};
gpio_pin_t pb1 = {PA_8, GPIOA, GPIO_PIN_8};

// 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();

  // initialise the gpio pins
  init_gpio(led, OUTPUT);
  init_gpio(pb1, INPUT);

  // loop forever ...
  while(1)
  {
    // if the button is pressed ...
    if(read_gpio(pb1))
    {
      // turn the led on on the gpio pin
      write_gpio(led, HIGH);
    }
    else
    {
      // turn the led off on the gpio pin
      write_gpio(led, LOW);
    }
  }
}
```

Enter this program and make sure everything works. If there are any problems contact a member of staff.

Now you are going to make some modifications to the program to add functionality:

1. Alter the above program to “latch” the button state (i.e. pressing the button turns the LED on, pressing it again turns the LED off).

2. You may have noticed that, in your “latched” button state program, the LED sometimes doesn't behave as expected – this is because the mechanical contacts on the switch may “bounce”, triggering the latch in quick succession. You should add “debouncing” to your program (which means checking the button state twice in a short period of time to ensure that the button was actually pressed and it wasn't just the contacts “bouncing”). Ask a member of staff is this is unclear.

3. Wire up two other LEDs of different colours. Adapt your program to cycle through the LEDs when the button is pressed.

Task 5

In this task we are going to use a potentiometer to control the blinking rate of an LED. To do this you should wire up the circuit shown in Figure 9.

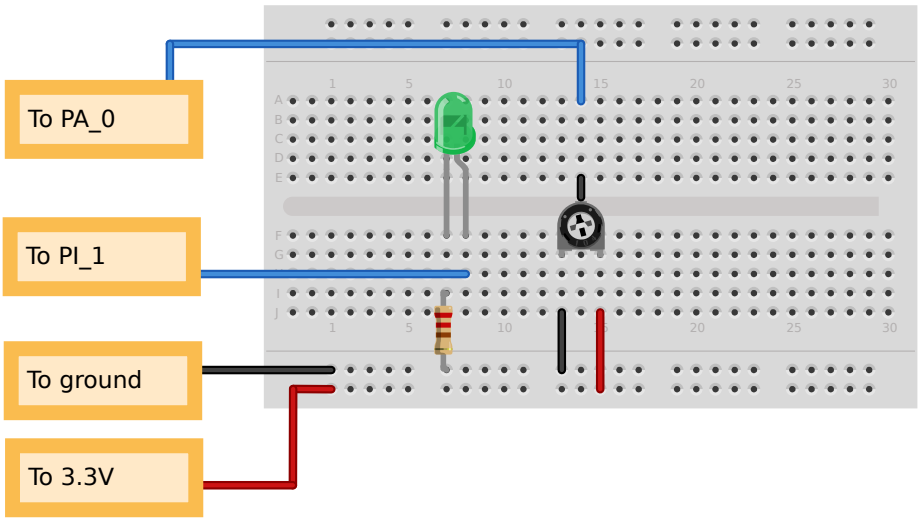


Figure 9 – The blink rate circuit

This circuit has the two rails of the breadboard[4] wired up to the +3.3V and ground pins on the STM32F7 discovery board (+3.3V to the lower rail and ground to the upper rail) allowing us to easily access the +3.3v supply and ground. The potentiometer then has one leg connected to +3V another leg connected to ground, and the centre leg (the "wiper") attached to GPIO pin PA_0. We then connect the LED to GPIO pin PI_1.

We then need to write our program to control the LED. First we have to read the value of the potentiometer via the ADC on the STM32F7 discovery board and then use this value to control the blink rate of the LED. Figure 9 shows the program logic for this example in pseudo-code.

4 Note that the breadboards on the current version of the discovery kit are slightly different to this image and so you will have to adjust the circuit accordingly

```
Procedure ADJUSTABLE_BLINK:

    while not ready:
        initialise UART 1
        initialise ADC pin (PA_0)
        initialise GPIO pin (PI_1)
    end while

    toggle LED pin (PI_1)
    delay = read POT pin (PA_0)
    print “delay” value to the terminal
    wait “delay” seconds
    repeat

end Procedure ADJUSTABLE_BLINK
```

Figure 9 – Pseudo code for the “adjustable_blink” program

As you can see, the logic for this program is fairly straightforward; however, the STM32F7 chip is complex and there are many different options available for configuring the ADC. We are only interested in a small subset of these, so I have provided a set of library functions that simplify the configuration process for the ADC.

Note that in this example we are also going to send the value of the potentiometer to a terminal via a UART (the virtual COM port on the USB port in this case). You will find this functionality extremely helpful in debugging programs of even moderate complexity!

The latest versions of Keil uVision make this process extremely easy by using the CMSIS Driver architecture and the Keil pack structure. Firstly we use the run-time environment manager to add the appropriate CMSIS Driver code to control the UART. Figure 10 shows where the run-time environment manager button is located, and Figure 11 shows the options for adding the USART driver.

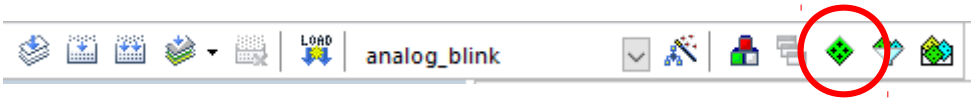


Figure 10 – The RTE button on the uVision toolbar

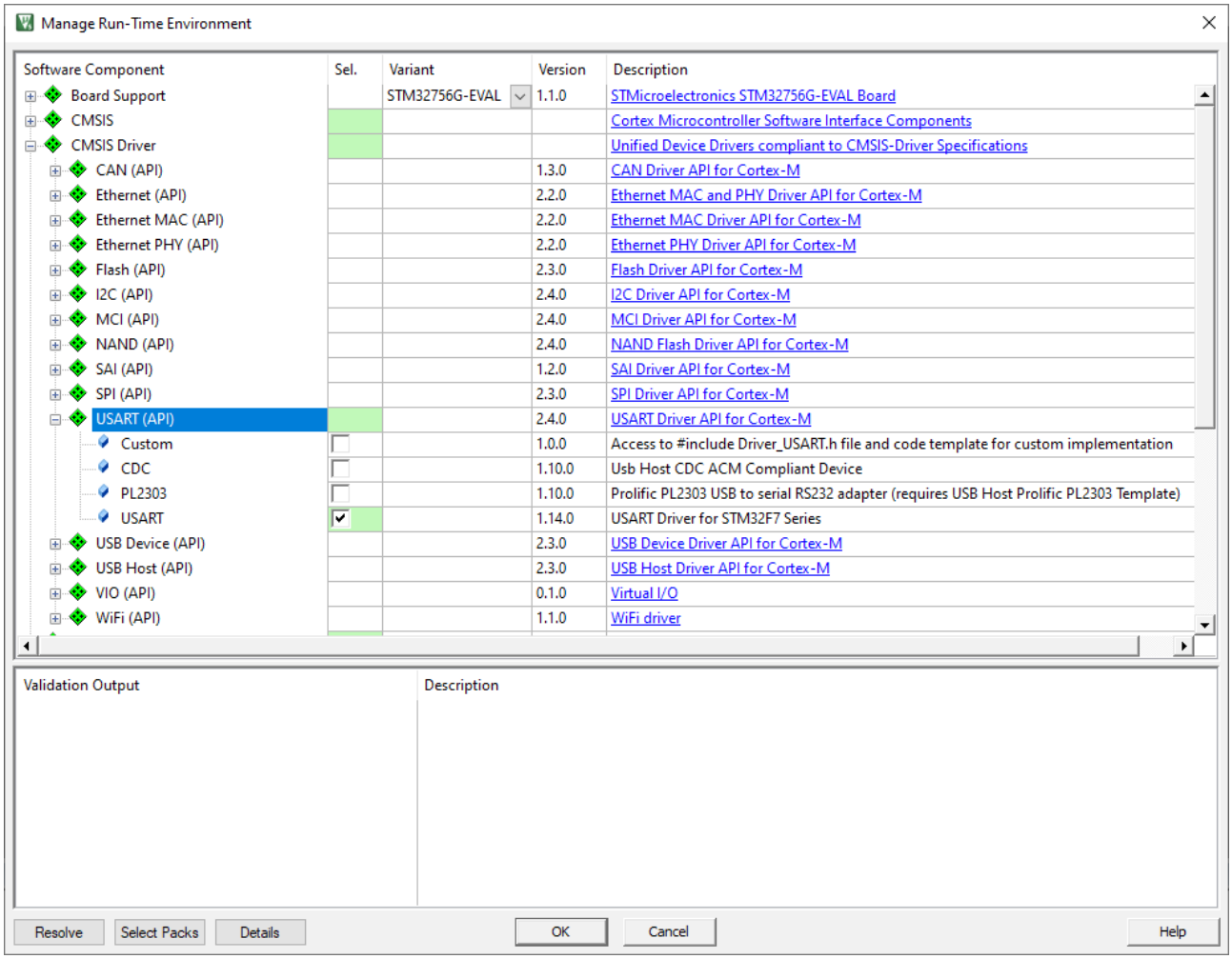


Figure 11 – The CMSIS Driver USART (API) options

After adding the CMSIS Driver USART code we then need to set up redirection of stdout to a user defined output target (e.g. the USART driver we included in the previous step). Figure 12 shows these options under the Compiler software component.

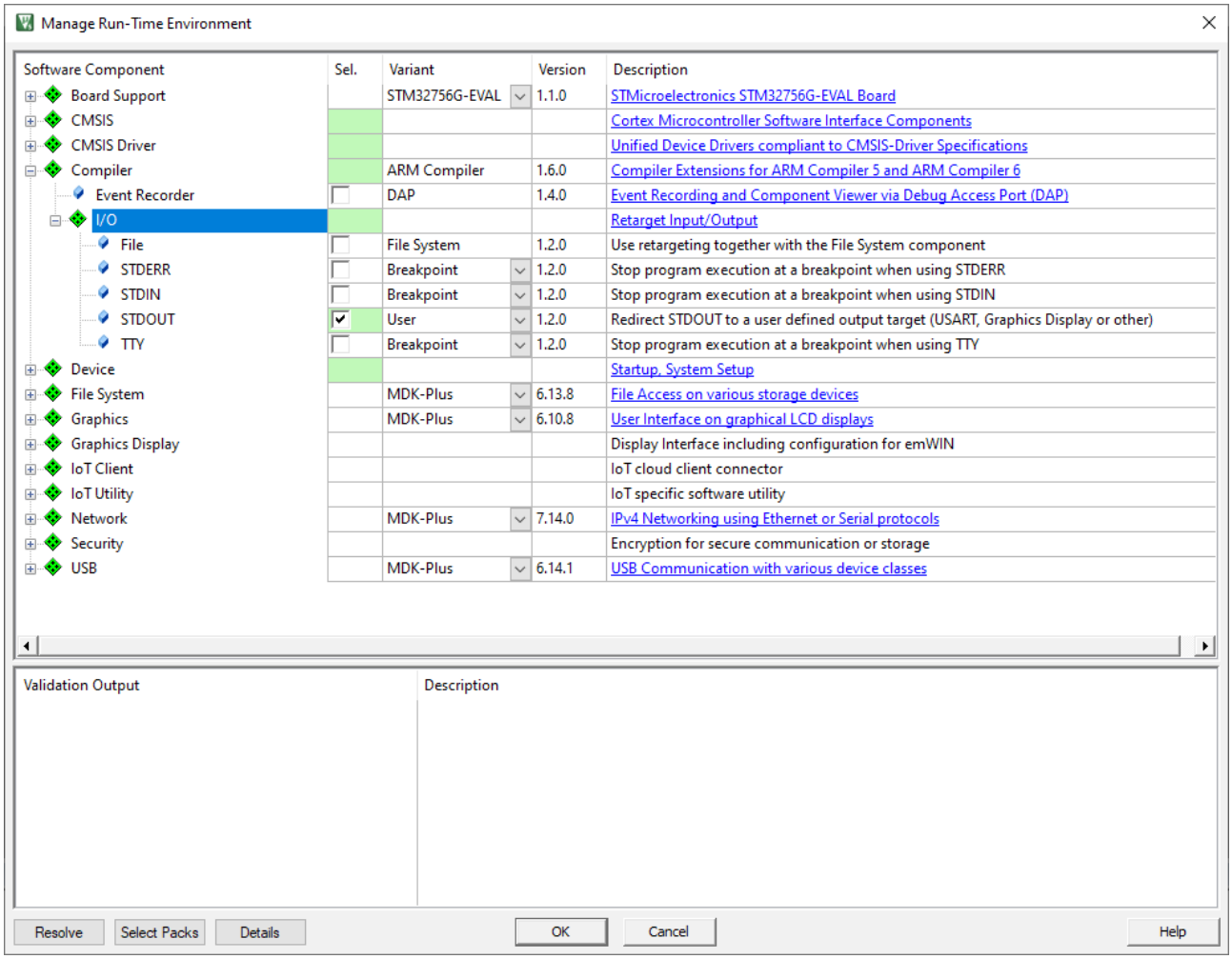


Figure 12 – STDOUT redirection using the ARM Compiler software component

Now we've set the run-time environment up with all the necessary drivers, we need to configure the appropriate UART on the board. We want to set up the UART that is associated with the virtual COM port (VCP) on the STM32F7 discovery board, as this will let us send message via the USB connection (this simplified both the hardware needed – as we don't need a separate cable to connect to the computer – and reduces the number of the broken out pins we are using). On the STM32F746 discovery board we are using in this module this is USART1 and uses pins PA9 (for TX) and PB7 (for RX). Figure 13 shows how we can enable and configure this USART using the '*RTE_Device.h*' header file (via the configuration wizard).

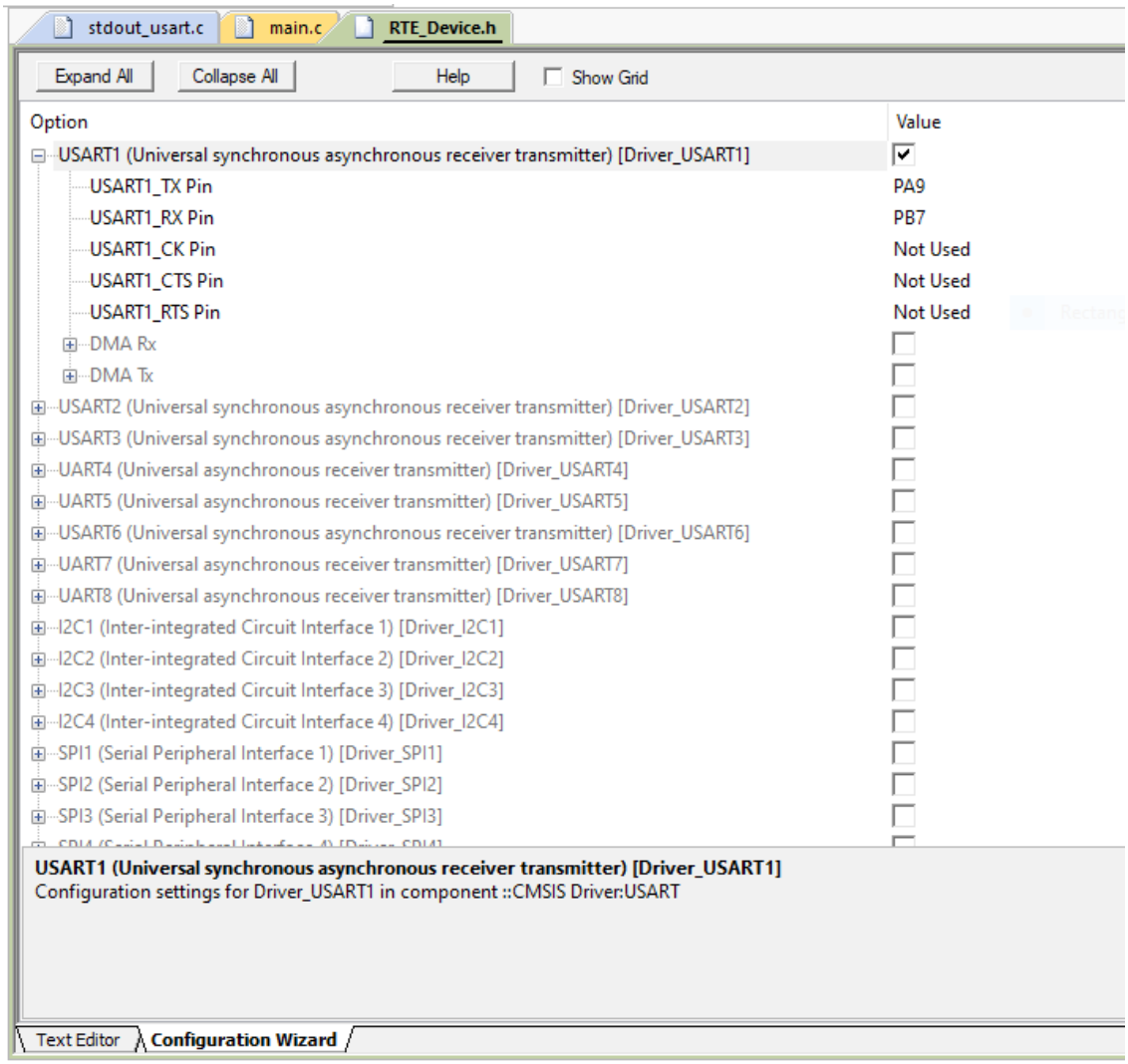


Figure 13 – Device configuration

Finally we need to create the stdout redirection file using the provided user code template (see Figure 14) and use the configuration wizard to point it at the correct serial port (see Figure 15).

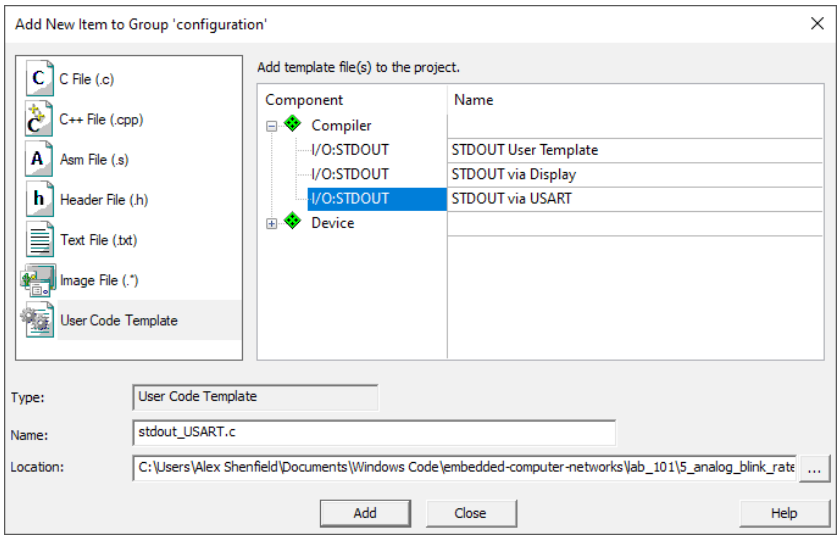


Figure 14 – Using the user code template to add a file for stdout redirection

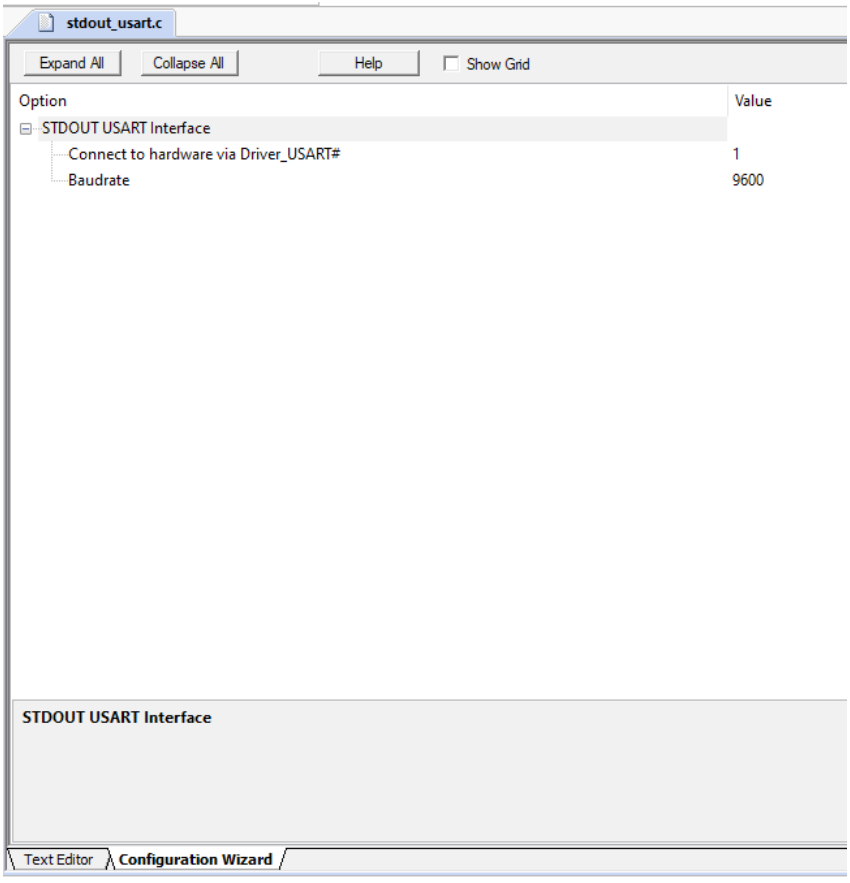


Figure 15 – Using the configuration wizard to point the stdout redirection at the appropriate USART (note also that we have set the baudrate here to 9600bps – you will need to know this when connecting TeraTerm)

The final project will end up looking like Figure 16, below.

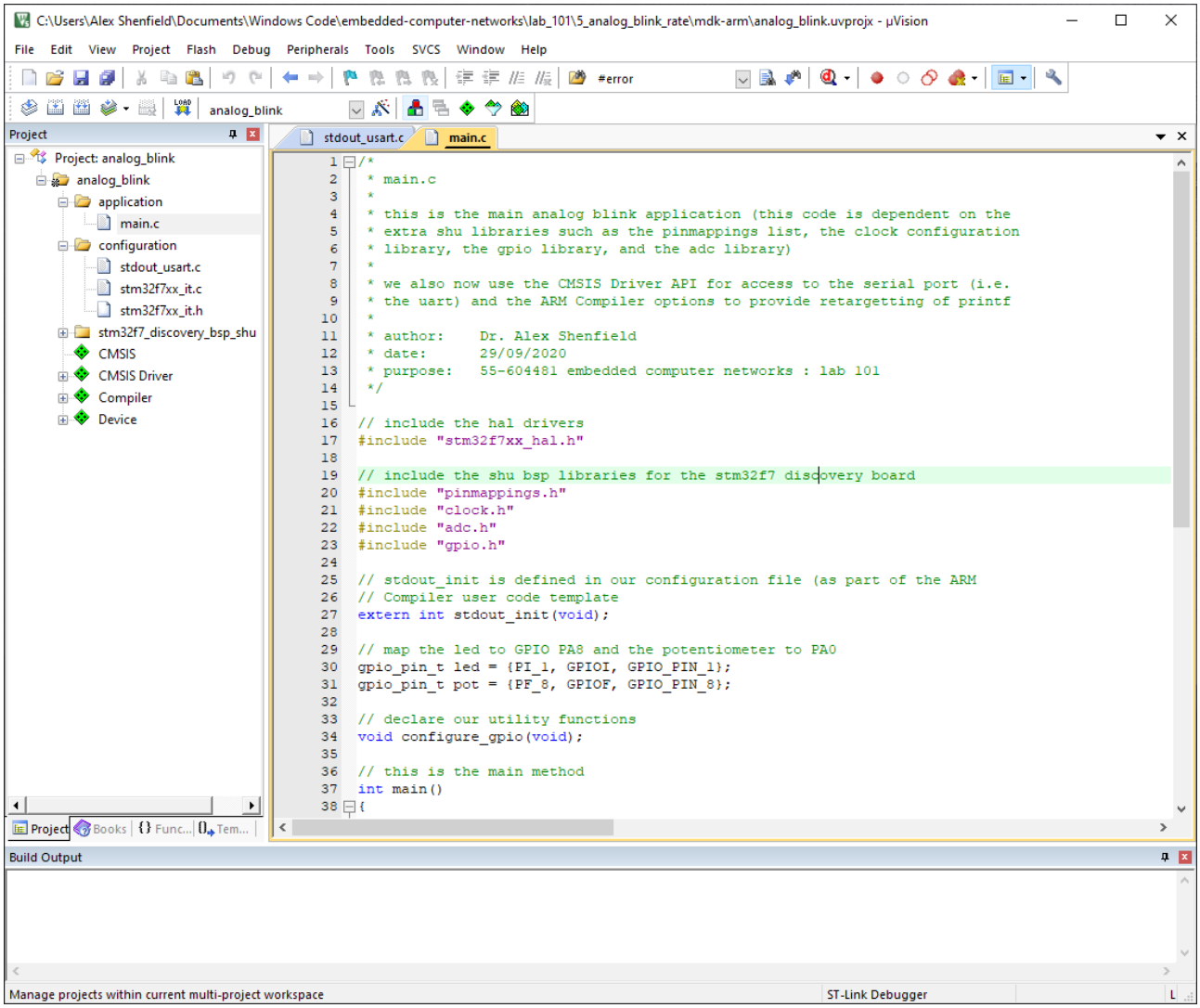


Figure 16 – The final configured project

We then need to implement the logic from our pseudo-code in Figure 9. The completed program is shown in code listing 4.

A Keil uVision 5 project (containing all the appropriate libraries) is available to download from GitHub.

Code listing 4:

```
/*
 * main.c
 *
 * this is the main analog blink application (this code is dependent on the
 * extra shu libraries such as the pinmappings list, the clock configuration
 * library, the gpio library, and the adc library)
 *
 * author:  Dr. Alex Shenfield
 * date:    29/09/2020
 * purpose: 55-604481 embedded computer networks : lab 101
 */

// include the hal drivers
#include "stm32f7xx_hal.h"

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

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

// map the led to GPIO PI1 and the potentiometer to PA0
gpio_pin_t led = {PI_1, GPIOI, GPIO_PIN_1};
gpio_pin_t pot = {PA_0, GPIOA, GPIO_PIN_0};
```

```
// 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();

   // initialise the uart, adc and gpio pins
   stdout_init();
   init_adc(pot);
   init_gpio(led, OUTPUT);

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

   // loop forever ...
   while(1)
   {
     // toggle the led pin
     toggle_gpio(led);

     // read the potentiometer and echo that value to the terminal
     uint16_t adc_val = read_adc(pot);
     printf("potentiometer value = %d : delay time = %d ms\r\n",
            adc_val, (adc_val/2));

     // delay for the appropriate time
     HAL_Delay(adc_val / 2);
  }
}
```

Compile and load this program and make sure everything works. If there are any problems contact a member of staff.

The debugging information is sent to a serial terminal, so you will need a terminal emulator to view this. I like TeraTerm for it's configurability and simplicity (though alternatives such as PuTTy are also available). Once you have loaded the code, you can open TeraTerm (in the SHU labs it's available in AppsAnywhere) and connect the the VCOM port as shown in Figure 17. When the program runs, you will then get the potentiometer value printed to the terminal window (as shown in Figure 18).

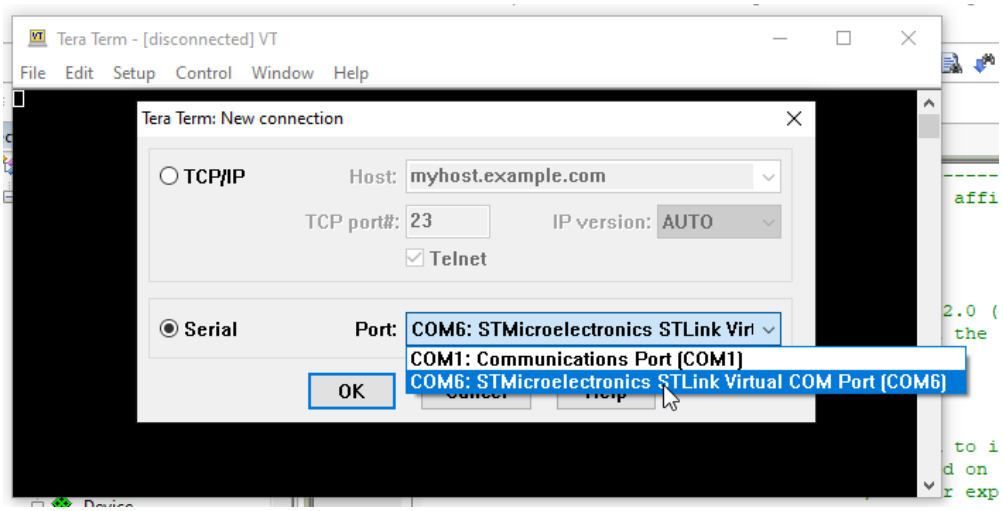


Figure 17 – Setting up TeraTerm to connect to the STLink Virtual COM Port

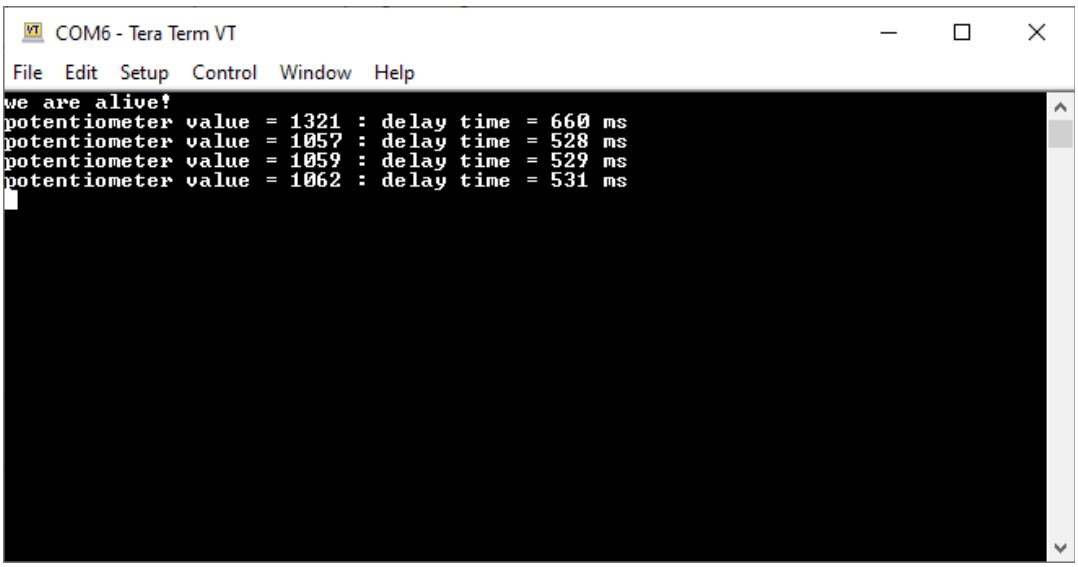
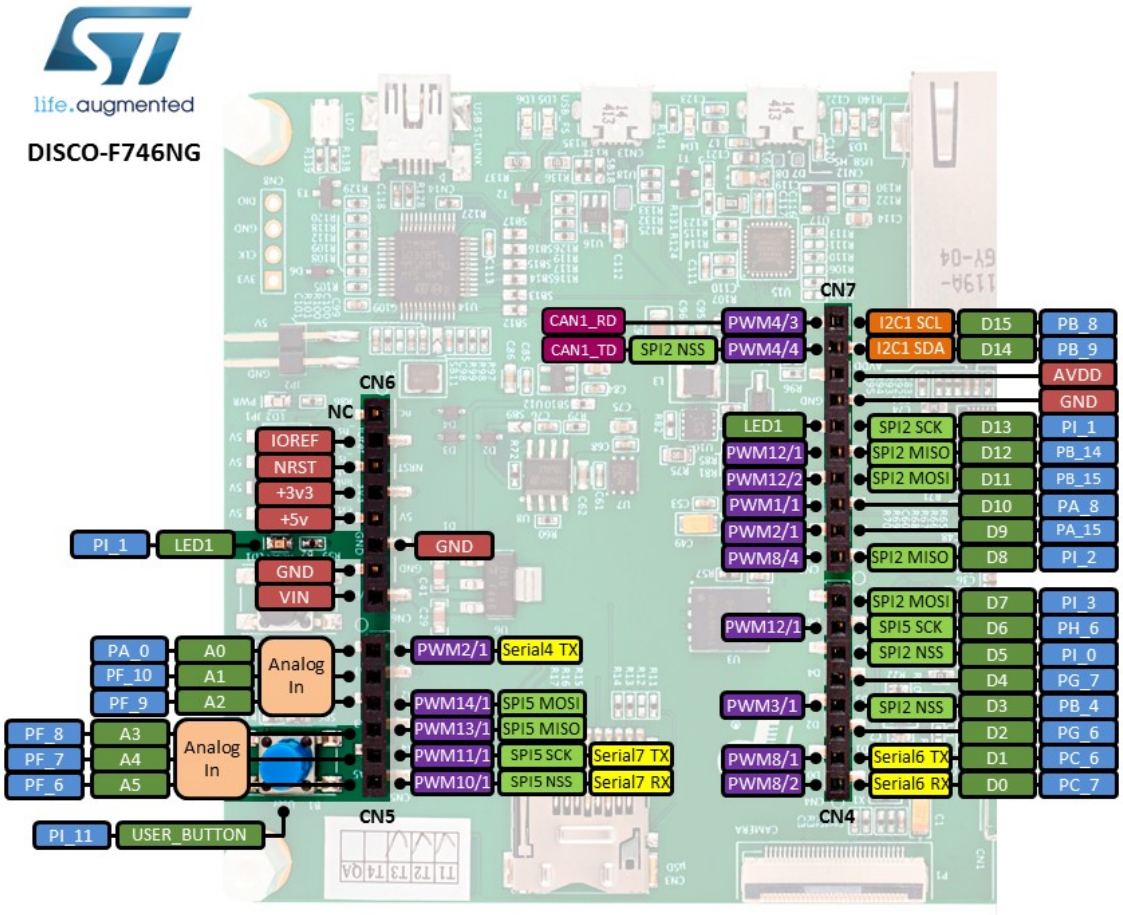


Figure 18 – TeraTerm terminal window output

Appendix A – The STM32F7 discovery board schematic



STM32F7 discovery board pin outs

Check list

Tasks	
Task 1 – Program	
Task 2 – Program	
Task 3 – Exercises a)	
Task 3 – Exercises b)	
Task 4 – Questions	
Task 4 – Program	
Task 4 – Additional exercise 1	
Task 4 – Additional exercise 2	
Task 4 – Additional exercise 3	
Task 5 – Program	

Feedback