Electronic Engineering 1Y / Microelectronics 1
Digital Electronics Laboratories:
Embedded Systems – Introduction to the microcontroller

Objectives

After completing these laboratories, you should be able to program an embedded microcontroller to,

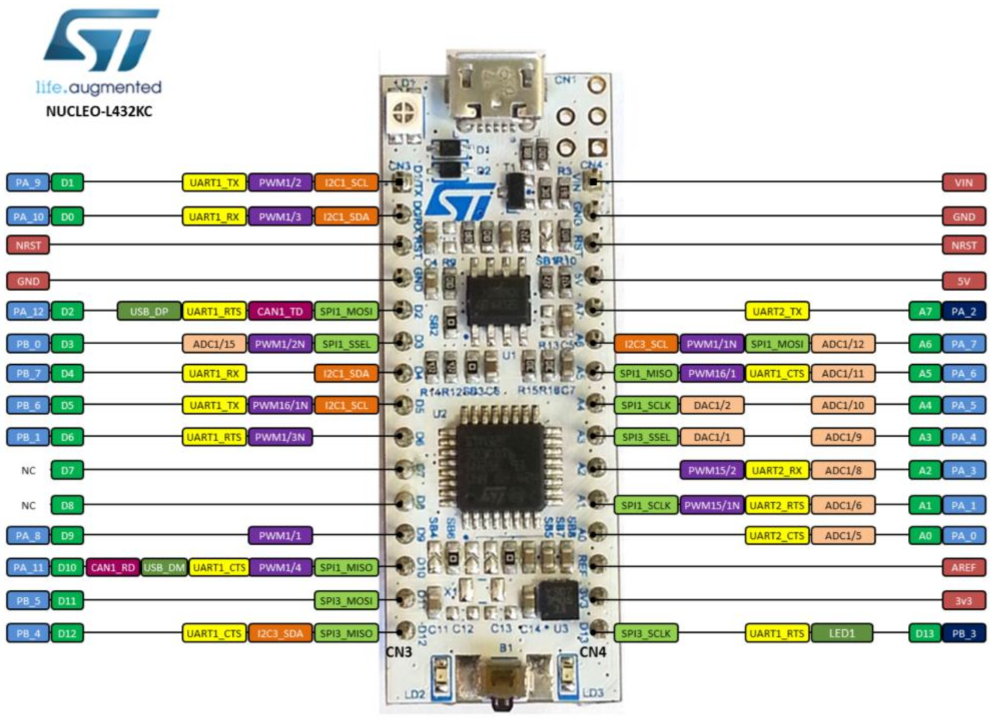
- generate digital output signals, receive and process digital input signals
- generate analogue output signals, receive and process analogue input signals
- generate pulse width modulated signals, and use a FET to drive a load
- communicate with something else through a serial link
- write to a display
- use serial communication

This set of experiments will be assessed by milestones.
You really need to get through **all the milestones**, as the follow-on lab project will require you to be proficient in getting a range of signals into and out of the microcontroller.
You should aim to complete these set of experiments within the planned lab sessions. That is up to 15 hours of lab time. I would recommend that you read the lab script beforehand and cross check through the lecture notes for the corresponding tasks. Your time in the lab should then be a bit more productive. Finally, **keep a lab book** and write things down in your lab book, especially things that were not as you thought, and hurdles you overcame.

YOUR ACTIVITIES BEGIN ON PAGE 12.

Overview of the Microcontroller

The embedded systems section of this course is based around a microcontroller development board. Each student will be given a board to keep, which hopefully you will come to cherish and use widely throughout your years of study in Glasgow. You will also be given a breadboard to enable you to easily connect components to your beloved MCU, as well as other passive components needed to accomplish your milestones. The Nucleo microcontroller development board is shown in Figure 1 below,



life.augmented
NUCLEO-L432KC
PA_9 D1 UART1_TX PWM1/2 I2C1_SCL
PA_10 D0 UART1_RX PWM1/3 I2C1_SDA
NRST
GND
PA_12 D2 USB_DP UART1_RTS CAN1_TD SPI1_MOSI
PB_0 D3 ADC1/15 PWM1/2N SPI1_SSEL
PB_7 D4 UART1_RX I2C1_SDA
PB_6 D5 UART1_TX PWM16/1N I2C1_SCL
PB_1 D6 UART1_RTS PWM1/3N
NC D7
NC D8
PA_8 D9 PWM1/1
PA_11 D10 CAN1_RD USB_DM UART1_CTS PWM1/4 SPI1_MISO
PB_5 D11 SPI3_MOSI
PB_4 D12 UART1_CTS I2C3_SDA SPI3_MISO
CN3
VIN
GND
NRST
5V
UART2_TX A7 PA_2
I2C3_SCL PWM1/1N SPI1_MOSI ADC1/12 A6 PA_7
SPI1_MISO PWM16/1 UART1_CTS ADC1/11 A5 PA_6
SPI1_SCLK DAC1/2 ADC1/10 A4 PA_5
SPI3_SSEL DAC1/1 ADC1/9 A3 PA_4
PWM15/2 UART2_RX ADC1/8 A2 PA_3
SPI1_SCLK PWM15/1N UART2_RTS ADC1/6 A1 PA_1
UART2_CTS ADC1/5 A0 PA_0
AREF
3v3
SPI3_SCLK UART1_RTS LED1 D13 PB_3
CN4

Figure 1 - Nucleo microcontroller development board and some of the pin possibilities.

The development board is based on the STM32L432KCU6 microcontroller (an integrated circuit manufactured by ST microelectronics), with a 32-bit ARM Cortex-M4 core running at 80MHz (ARM processors are used in many contemporary products).
The connections from the board that you will use are power (3.3V and GND) and pins labelled as D or A. For example, the development board has many interfaces:

- **Digital IN and OUT**: pins D0 to D6, D9 to D13 and A0 to A3 and A6 to A7
- **Analogue IN**: pins A0 to A6 and D3 can also be configured for analogue input
- **Analogue OUT**: pins A3 and A4 can be configured for analogue output
- **PWM OUT** (Pulse Width Modulation): pins D0, D1, D9, D10, A2, A5
- Various other pins can be configured for **serial communication** such as CAN, SPI, I2C and UART connection.

Potential irregularities:

- Pins A4 and A5 are only **ADC input**
- Pins D7 and D8 are not used.
- The pins marked PWMx/yN are inverted waveforms of the PWMx/y outputs.
- For PWM, the x represents a timer and y is channel, so if you want a different frequency, you must use a different timer, i.e. PWM1 and PWM16. If you are happy with the same frequency but want different duty cycles, you can use PWM1/1, PWM1/2 etc…

The real beauty of the development board is the simplicity with which the microcontroller can be programmed via the online compiler in the MBED environment. This comes with an extensive application programming interface (API), which is a large set of building blocks (libraries) which are effectively C++ utilities which allows programs to be easily, and quickly developed. Code can be generated on the web-based complier which means you can be developing software whenever you have internet access. Once compiled, programs are easily downloaded to the board via a USB connector, which can also be used to power the board from the 5V USB output.

There are 3 LEDs on the board, one for 'power on' (red LED2 next to pin D12), a dual colour LED1 for status, and a green LED3 that is connected to pin D13 digital output. LED3 can be driven to test basic functions without the need for any external component connection. However just using a microcontroller for one LED would be pretty boring, so we will want to attach other stuff…

To connect external components to the microcontroller, you will need to plug the Nucleo board into a breadboard, similar to the one shown in Figure 2 below. You should consider which way round it should go and make sure the USB lead does not take valuable space.

It is useful first step to connect GND and the 3.3V to the power rails of the breadboard since they are used a lot in circuits.

BE AWARE:

You should be aware that various input and output pins have maximum voltage and current limits, which should not be exceeded, otherwise your cherished microcontroller will be terminally damaged. **Microcontroller input pins should not see voltages in excess of 5V.**

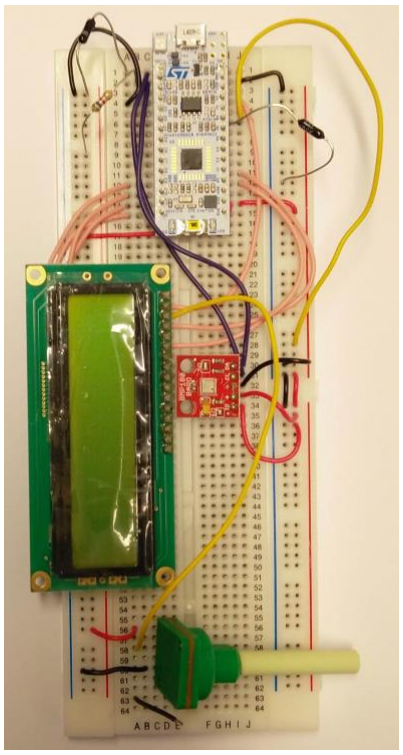


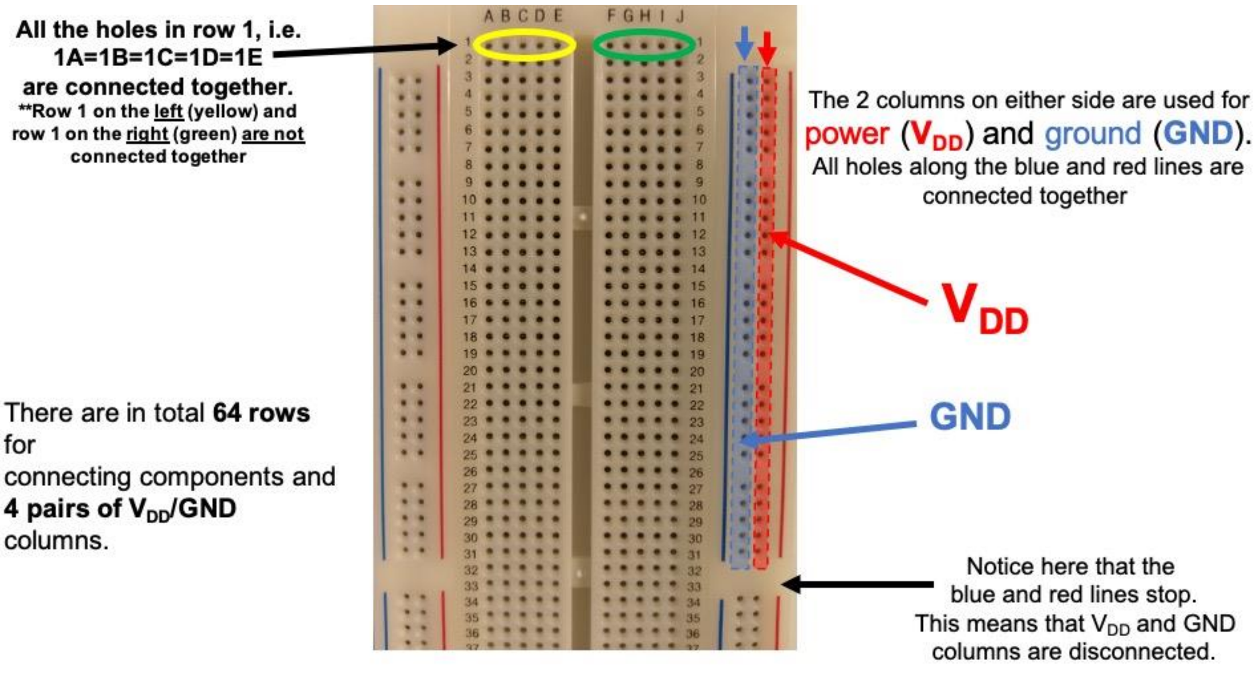
Figure 2 - an mbed board plugged into a breadboard

As you can see from Figure 1, the MCU has a number of voltage pins, including:

- **GND** : the reference potential = 0V. This should always be used as the ground for your logic circuits.
- **VIN** : 7V - 12V input if running on a battery rather than the usb
- **5V** : 5.0 V USB output that comes directly from your USB
- **3V3** : 3.3 V regulated output from the on board regulator – this is the voltage referred to as V_{DD} in the course notes, and should always be used as the +ve for your logic circuits.

Getting Started with your Development Breadboard

When you arrive at the lab on your first session, you will get a Development Breadboard (also your STM32L432KCU6 MCU). The breadboard will look like the one in the picture below, it is a standardized component and particularly useful for developing prototype circuits.



All the holes in row 1, i.e. 1A=1B=1C=1D=1E are connected together.
****Row 1 on the <u>left</u> (yellow) and row 1 on the <u>right</u> (green) <u>are not</u> connected together**

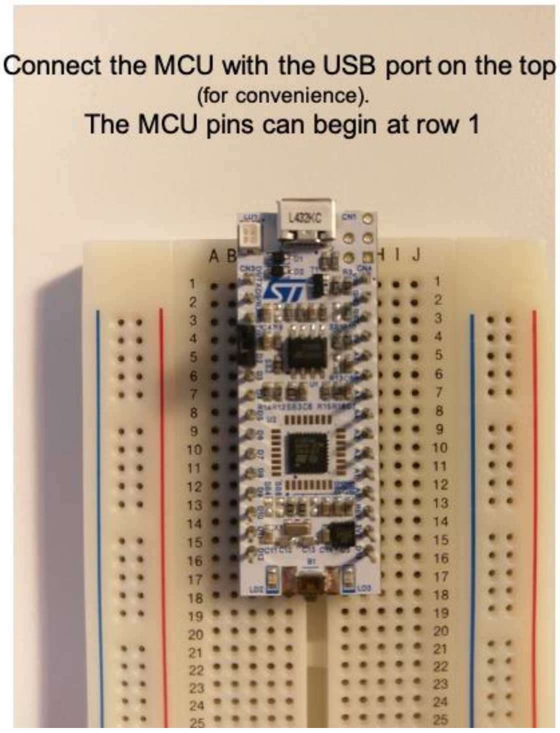
The 2 columns on either side are used for power (**V_{DD}**) and ground (**GND**). All holes along the blue and red lines are connected together

There are in total **64 rows** for connecting components and **4 pairs of V_{DD}/GND** columns.

Notice here that the blue and red lines stop. This means that V_{DD} and GND columns are disconnected.

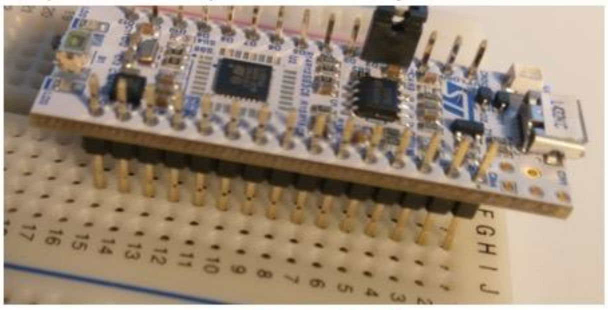
Please study the picture above and understand how a Breadboard works!!

****Mount your microcontroller like is shown in the picture below.**

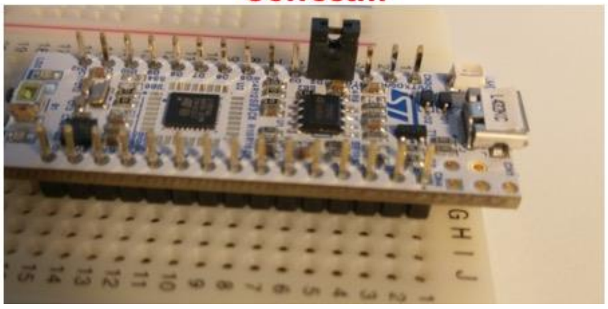


Connect the MCU with the USB port on the top (for convenience).
The MCU pins can begin at row 1

Incorrect!
The pins should be pushed all the way on the breadboard.

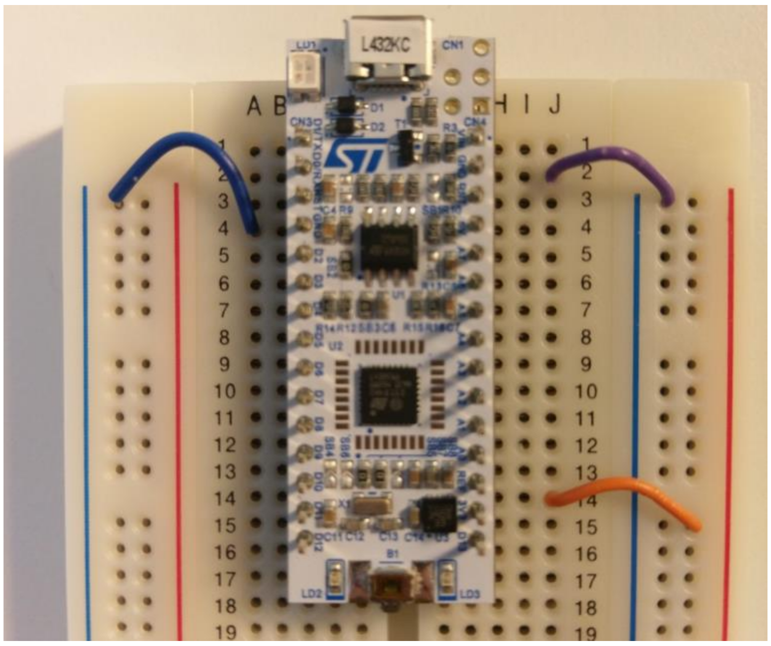


Correct!!!

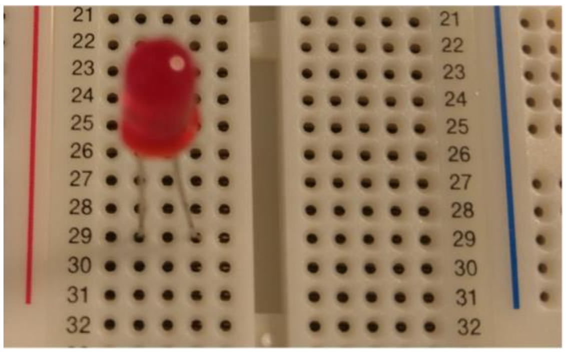


Before creating any complicated prototypes on our breadboard, the very first thing to do is to **connect the VDD (red) and GND (blue) columns with our microcontroller**. By doing so at the beginning, we can create tidy prototypes that can be easily debugged if there are any problems. On your MCU there are **two GND pins and one V_{DD} pin** (3.3V).

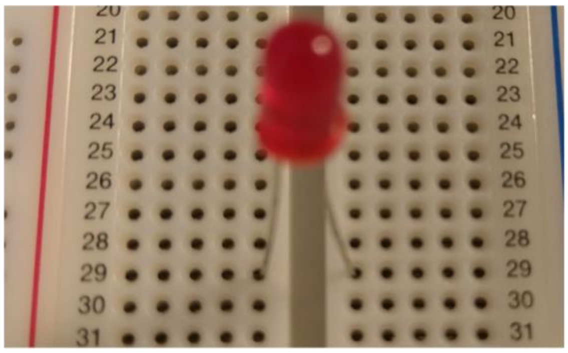
Use wires with appropriate colours to define GND and V_{DD}



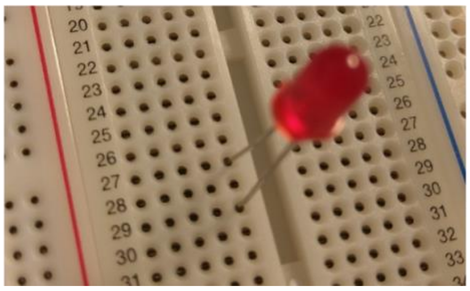
How to connect components on your Breadboard



Incorrect! Both LED legs are connected on the same row, therefore connected together

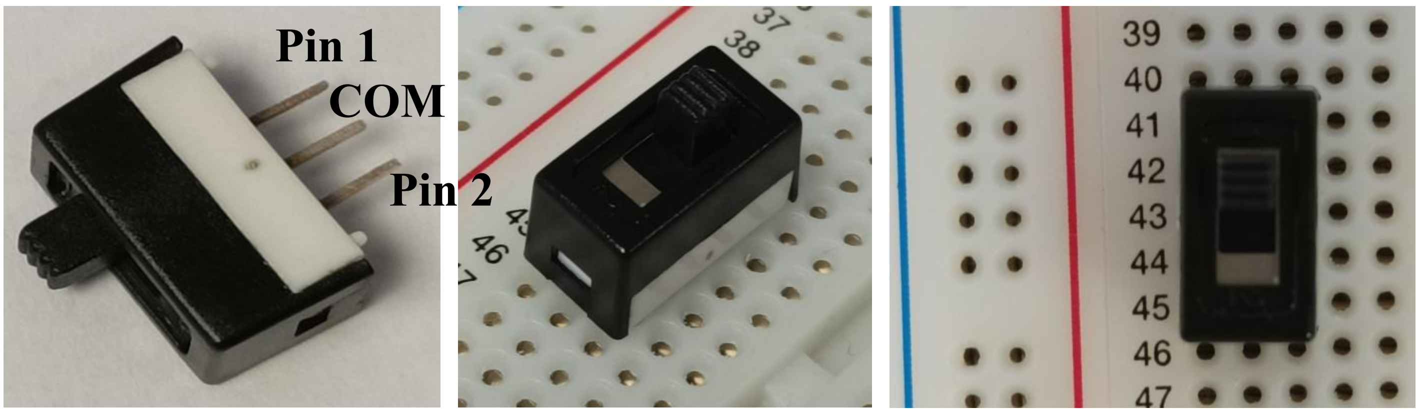


Correct! Both LED legs are connected on the same row, but on columns E and F. Therefore not connected together.

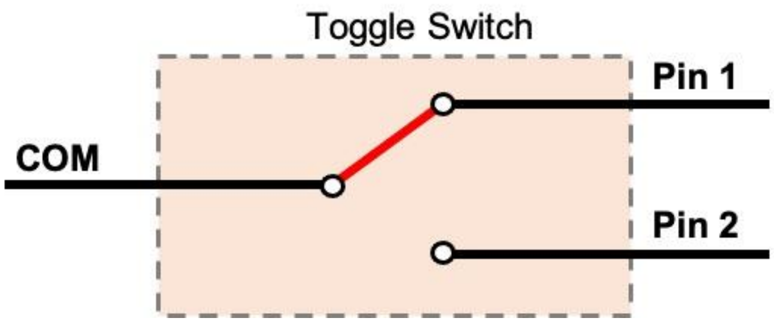


Correct! Both LED legs are connected on different rows. Therefore not connected together.

Toggle Switch (with 3 pins)



There are 3 pins. The **middle pin** is the **COM** (common) pin and the other two are **Pin 1** and **Pin 2** respectively. The wiring diagram of this switch is the following,



Getting Started with the Programming

Throughout this course we will be using the **Zephyr** platform to program your MCU. Zephyr is an open-source, cross-platform Real Time Operating System (RTOS), that you can program MCUs from different manufacturers.

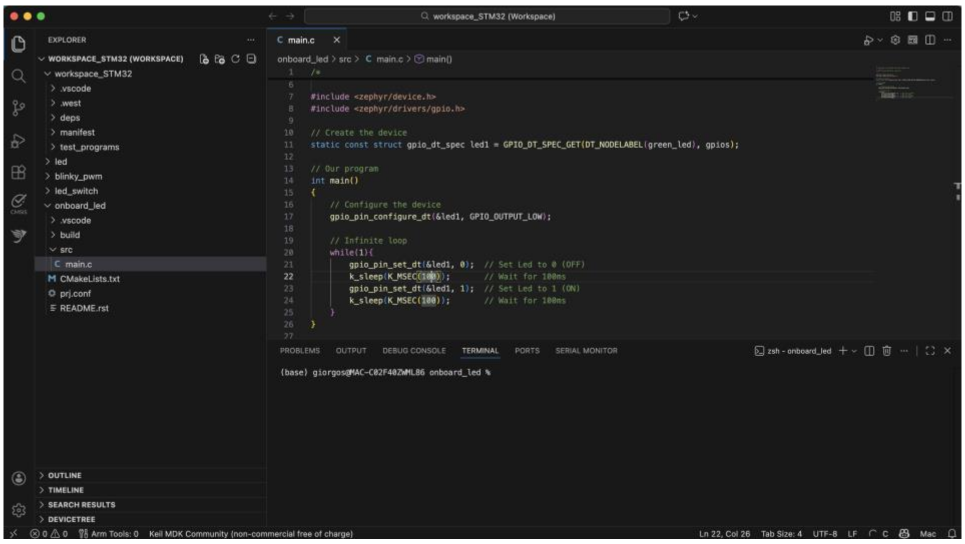
Zephyr RTOS can be accessed via Visual Studio Code (VS Code), through the Zephyr Workbench extension. More details on how to install Zephyr Workbench can be found on the video instructions uploaded on your Moodle page:

1. https://moodle.gla.ac.uk/mod/resource/view.php?id=5816079 (ENG 1022 link)
2. https://moodle.gla.ac.uk/mod/resource/view.php?id=5816082 (ENG1064 link)

The installation may take a couple of hours.

Please install Zephyr on your personal laptop, or on your University Anywhere Desktop account before the lab sessions.

VS Code and Zephyr can run on either Windows, Mac or Linux platforms. After you follow the video instructions and Zephyr Workbench is installed you will be able to see the programming environment, as shown in figure 3 below.



```c
/*

#include <zephyr/device.h>
#include <zephyr/drivers/gpio.h>

// Create the device
static const struct gpio_dt_spec led1 = GPIO_DT_SPEC_GET(DT_NODELABEL(green_led), gpios);

// Our program
int main()
{
    // Configure the device
    gpio_pin_configure_dt(&led1, GPIO_OUTPUT_LOW);

    // Infinite loop
    while(1){
        gpio_pin_set_dt(&led1, 0);  // Set Led to 0 (OFF)
        k_sleep(K_MSEC(100));       // Wait for 100ms
        gpio_pin_set_dt(&led1, 1);  // Set Led to 1 (ON)
        k_sleep(K_MSEC(100));       // Wait for 100ms
    }
}
```

```
(base) giorgos@MAC-C02F40ZWML86 onboard_led %
```

Figure 3 - Screenshot of mbed compiler environment

To run your first program, switching ON and OFF an LED (like we did in Lecture 2) follow the instructions of the videos found on:

While the program is downloading (flashed) on to the MCU memory, the status top left LED on the MCU should flash quickly. Your MCU will run the newest program you send to it.

For this particular program, you should see the green led (bottom right) flashing on and off every 0.1 seconds. By the way, the **led1** for the program is actually labelled **LD3** or **green_led** on the NUCLEO board, just to add to the confusion.

```
/*
 * Giorgos Georgiou
 * University of Glasgow
 * ENG1022 & ENG1064
 * Jan 2026
 */

#include <zephyr/device.h>
#include <zephyr/drivers/gpio.h>

// Create the device
static const struct gpio_dt_spec led1 = GPIO_DT_SPEC_GET(DT_NODELABEL(green_led), gpios);

// Our program
int main()
{
    // Configure the device
    gpio_pin_configure_dt(&led1, GPIO_OUTPUT_LOW);

    // Infinite loop
    while(1){
        gpio_pin_set_dt(&led1, 0);  // Set Led to 0 (OFF)
        k_sleep(K_MSEC(100));       // Wait for 100ms
        gpio_pin_set_dt(&led1, 1);  // Set Led to 1 (ON)
        k_sleep(K_MSEC(100));       // Wait for 100ms
    }
}
```

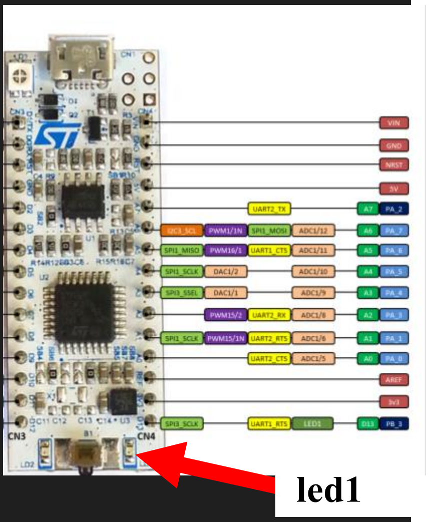
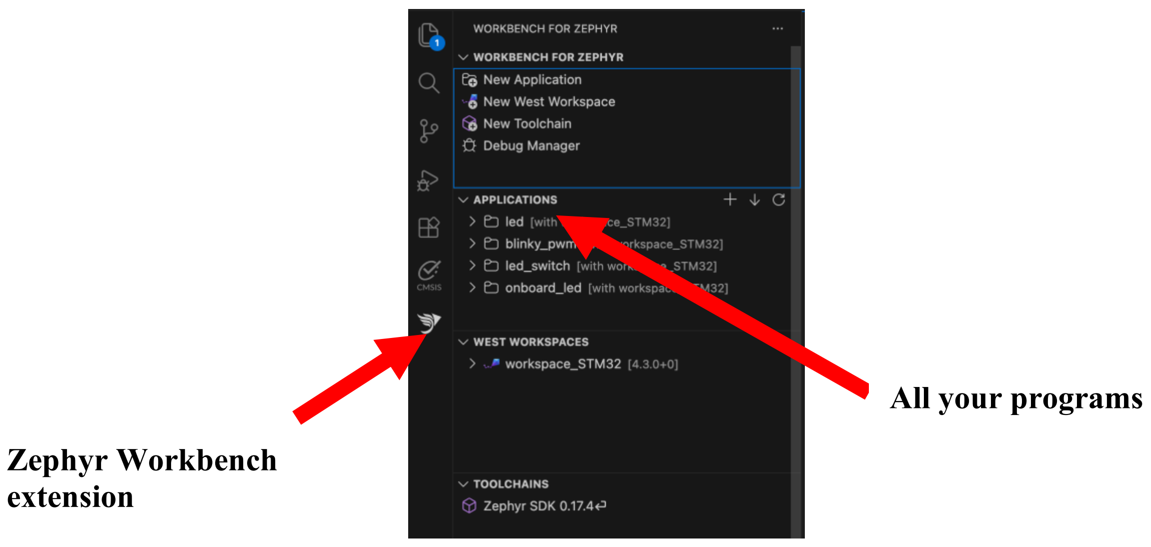


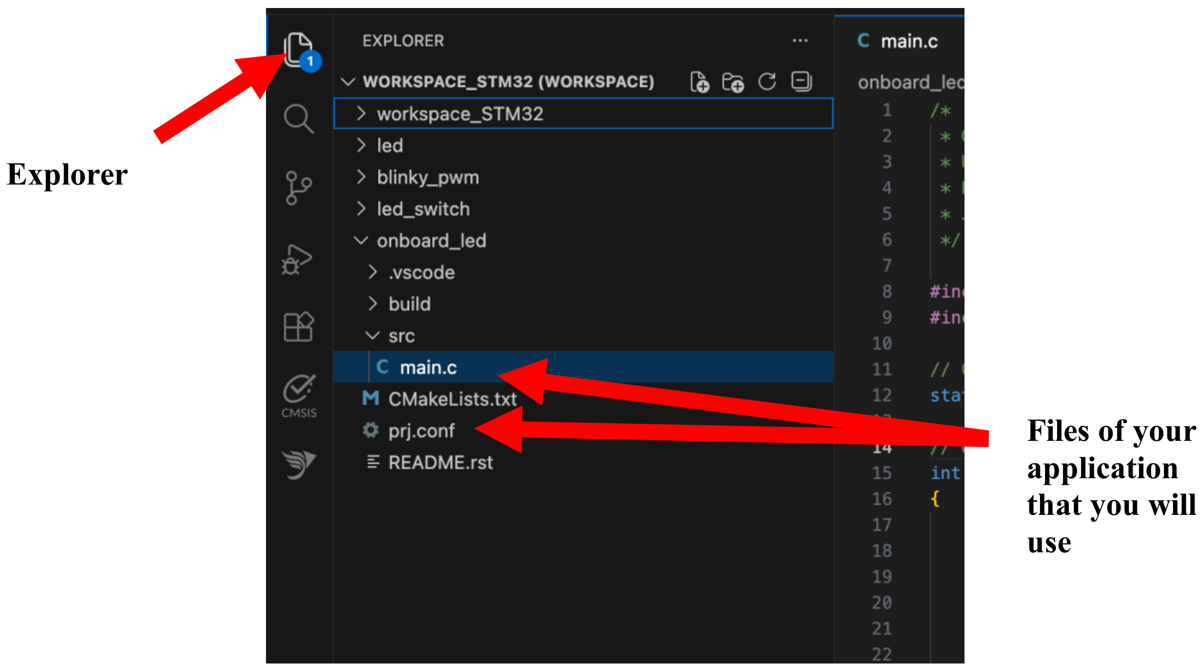
Figure 4 - The code to switch ON and OFF the on-board green led

To make more programs, follow the instructions on the videos provided.

Your newly created program will appear in the Applications Workspace on the left of VS Code.



In the Explorer you can see the contents of each application. If you expand the application by clicking on the >, you will see something called **src** and inside there a **main.c** . Double click on this, it will open your program main VS Code environment.



Once you write your program, you can compile and flash it on the MCU.

See: GPIO_output.mp4

The DigitalOut component

As you can see, it only takes a few lines of code to program the MCU. This is because of the Application Programming Interface (API) that was mentioned previously. Basically, this is a set of programming building blocks, which are C++ libraries, which allow rapid code development.
Based on this, we can explore each line of code in Figure 4.

Commenting on C++
All the lines between /* and */ are coloured in green and are just comments. The compile ignores all of this.
All the text on one line after // is comment. The compiler ignores this.
Comments are there to explain how the program is working, and help greatly when you need to submit code for assessment or when working as a team.

The program:

```
#include <zephyr/device.h>
```

This is a header that needs to be at the start of each Zephyr program – it is a link to a set of libraries specific to your hardware – in other words it tells the generic C++ compiler all the important information about the target of the program (your STM32L432KC) and contains all the 'standard' MCU programming functions.

```
#include <zephyr/drivers/gpio.h>
```

This is a Zephyr library that we need to include whenever we use General Purpose Input Outputs (GPIO). It contains useful commands that we can use to communicate with a GPIO. Similarly there are similar libraries for other peripherals, i.e. /adc.h , /pwm.h , /spi.h etc.

```
// Create the device
static const struct gpio_dt_spec led1 = GPIO_DT_SPEC_GET(DT_NODELABEL(green_led), gpios);
```

Before using any peripheral device on our MCU, we need to create that device as a variable. This variable will link to the MCU memory register that controls the specific GPIO. The above command follows Zephyr's standard library definitions, where the created device becomes a variable of type

`gpio_dt_spec`

which is a Zephyr variable for a GPIO device. In this example, we name this device `led1`.
The variable is equal to,

`GPIO_DT_SPEC_GET(DT_NODELABEL(green_led), gpios);`

This command calls a function named `GPIO_DT_SPEC_GET` to retrieve the ROM register of a GPIO device labelled **green_led**.
We will explain in more detail during our next lecture what happens with this command.

Something important I would like to highlight is that in Zephyr we will be modifying 3 files.

- The main.c program
- An overlay file
- A proj.conf file.

```
// Our program
int main()
{
```

The first action of any C++ program is contained within its **main()** function. The function definition, ie what goes on inside the function - is contained within the curly brackets, starting immediately after **main()**, and continuing until the last closing curly bracket } (line 15). On the microcontroller you have, the main function runs only once every time you press the reset button – it does not repeat. This is not particularly useful – imagine a lift that went to one floor and then stopped forever. We need its code to keep running and running…

```
  // Configure the device
  gpio_pin_configure_dt(&led1, GPIO_OUTPUT_LOW);
```

Essentially, what you are doing here is using the gpio_pin_configure_dt function (from the gpio.h library) to set **led1** as a digital output, initialised as logic 0 (low). Note the "&" character in front of the led1 variable. This is a C++ operator that points to the location of **led1** in the memory.

```
  // Infinite loop
  while(1){
```

Many embedded systems programs contain an endless loop, ie a program that just repeats for ever. In other branches of programming, this is bad practice, but for embedded systems, this is necessary. The endless loop is created using the **while** keyword; this controls the code within the curly brackets which follow. Normally, **while** is used to set up a loop, which repeats only if a certain condition is satisfied, but if we write **while(true),** or **while(1)** this will make the loop repeat endlessly. The value 1 is the same as Boolean 'true', which means that we have asked while the value is true do the stuff in brackets. Be careful, C++ is case sensitive, so true in lower case is not the same to the compiler as TRUE in capitals.

The part of the program that actually causes the LED to switch on and off is contained in the 4 lines within the **while** loop. There are two calls to the library function **k_sleep();,** and two statements **gpio_pin_set_dt();** in which the value of **led1** is changed.

```
    gpio_pin_set_dt(&led1, 0);  // Set Led to 0 (OFF)
    k_sleep(K_MSEC(100));       // Wait for 100ms
    gpio_pin_set_dt(&led1, 1);  // Set Led to 1 (ON)
    k_sleep(K_MSEC(100));       // Wait for 100ms
```

In detail,

```
gpio_pin_set_dt(&led1, 0);  // Set Led to 0 (OFF)


gpio_pin_set_dt(&led1, 1);  // Set Led to 1 (ON)
```

means that the variable **led1** is set to the value 0 or 1, whatever its previous value was. This sets a logic 0, or voltage 0V on the pin to which **led1** is connected. This will turn the **led1** off. Similarly, logic 1, or voltage of 3.3V, will cause the LED to light up.

```
k_sleep(K_MSEC(100));      // Wait for 100ms
```

This function is from the **device.h** library – the one you included at the start of your program. There are other variations of this command. You can use:

- k_sleep();   // time from functions K_MSEC(), K_USEC(), K_NSEC() etc.
- k_busy_wait(usec)   // time in microseconds

The 100 parameter in the K_MSEC() is an integer in milliseconds, and defines the delay length caused by this function. What actually happens is that the MCU will sleep, i.e. be inactive for the following 100ms. You can also use smaller units inside the k_sleep(), e.g.

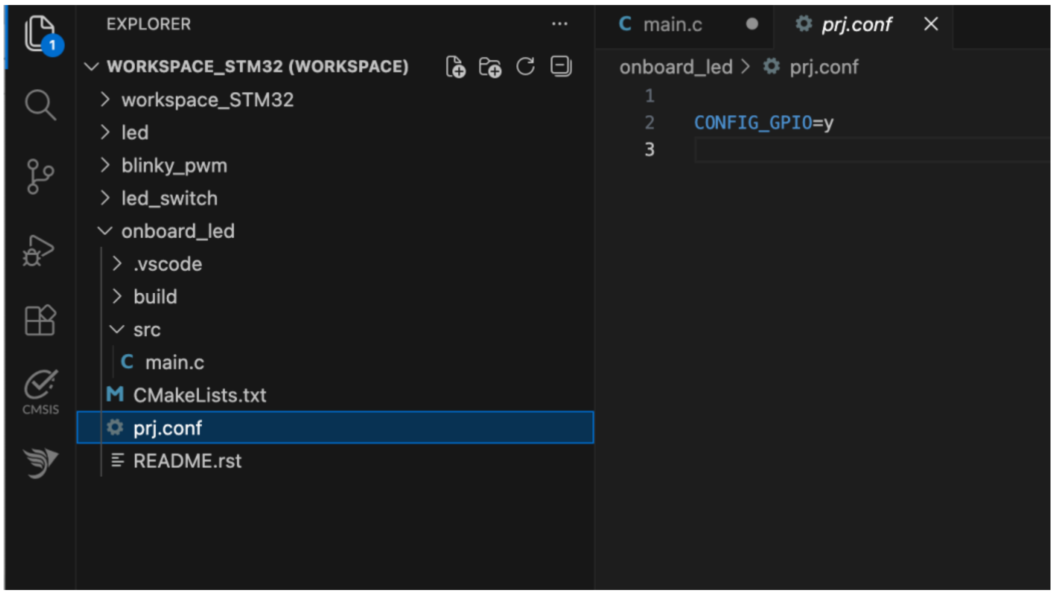
• K_NSEC( );	// delay in nanoseconds
• K_USEC( );	// delay in microseconds
• K_MSEC( );	// delay in milliseconds
• K_SECONS( );	// delay in seconds **
• K_MINUTES( );	// delay in minutes **
• K_HOURS( );	// delay in hours **
• K_FOREVER;	// infinite delay

** Note that some of the generated longer delays may not be accurate, as delays are defined by precise timers in your MCU. We will discuss how timers work in a future lecture.

The program will then reach the } on the last two lines. The commands in the while-loop are over, and so the program proceeds to start the while again, and because it is an infinite loop while(1) so the LED flashing will continue for ever…

The prj.conf

Zephyr requires that we include configu



```
CONFIG_GPIO=y
```

```
CONFIG_GPIO=y
```

This line ensures that the GPIO config files are included. Note: If this line is not included you will get an error during compilation.

Modifying the Program Code to control an external LED

See this video, GPIO_output.mp4 to help you configure pin D12 as a digital output.

Program 1: Switching on and off a pair of LEDs

Based on what you have learned above, and with reference to the Moodle video on GPIO, generate code that will flash between a red LED connected to pin **D11** and a green LED connected to pin **D12** of the MCU board. Use the development board to connect the mbed and the LEDs. The duration of illumination of each LED should be 1 second. The anode of the LED normally has the longer lead and also is the smaller piece of metal inside. The cathode also has a flat to the LED dome. BUT these are not universally true, so beware. Use an appropriate resistor to limit the current through the LED. Typically, 50-200 Ohms. Experiment with different resistance values and check that the LED brightness is indeed changed.

Learning Outcomes:

- Use an MCU GPIO and configure it as an output signal.
- Properly connect an LED on the development board
- Know the purpose and the value of a resistor connected together with an LED

Milestone 1: Show your code and the LEDs switching on and off to a demonstrator.

Program 2: Generating a square wave and observing it on an oscilloscope

Write a program to generate a 1kHz square wave, and observe it on an oscilloscope. You can choose whichever output pin you wish as the source of the square wave. Observe it on your oscilloscope.

Use a different pin and generate a second, 100kHz square wave. Observe it on your oscilloscope. Note that you will have to change the time scale on your oscilloscope.

Note: Try and use different delay commands

- k_sleep(); // time from functions K_MSEC(), K_USEC(), K_NSEC() etc.
- k_busy_wait(usec) // time in microseconds

Do you see anything different?

Learning Outcomes:

- Properly connect the oscilloscope probe on their MCU.
- Be able to use an oscilloscope – know the Vertical, Horizontal and Trigger knobs.
- Be able to read the vertical (Voltage) and horizontal (Time) scales.
- Be able to convert frequency to period.

Milestone 2: Show your code and the output waveform to a demonstrator