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,

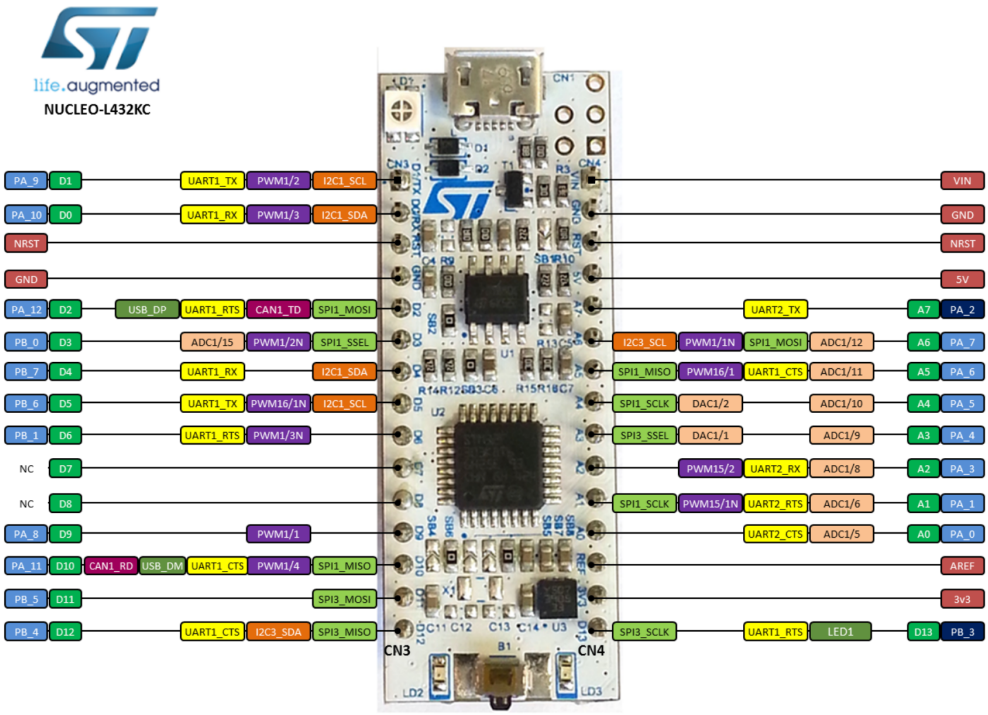


Figure 1 - Nucleo microcontroller development board and some of the pin possibilities.

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.**

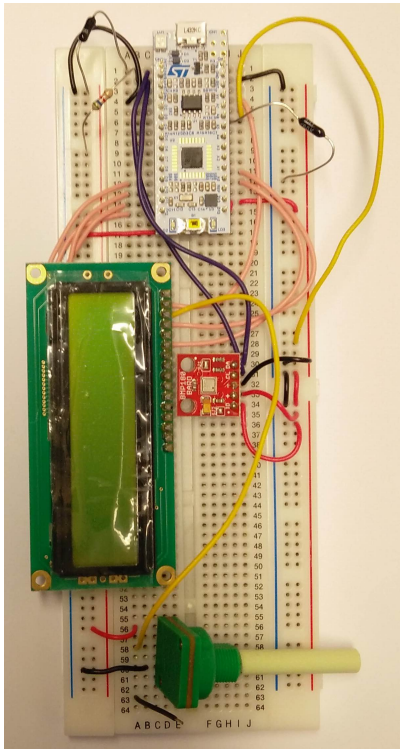


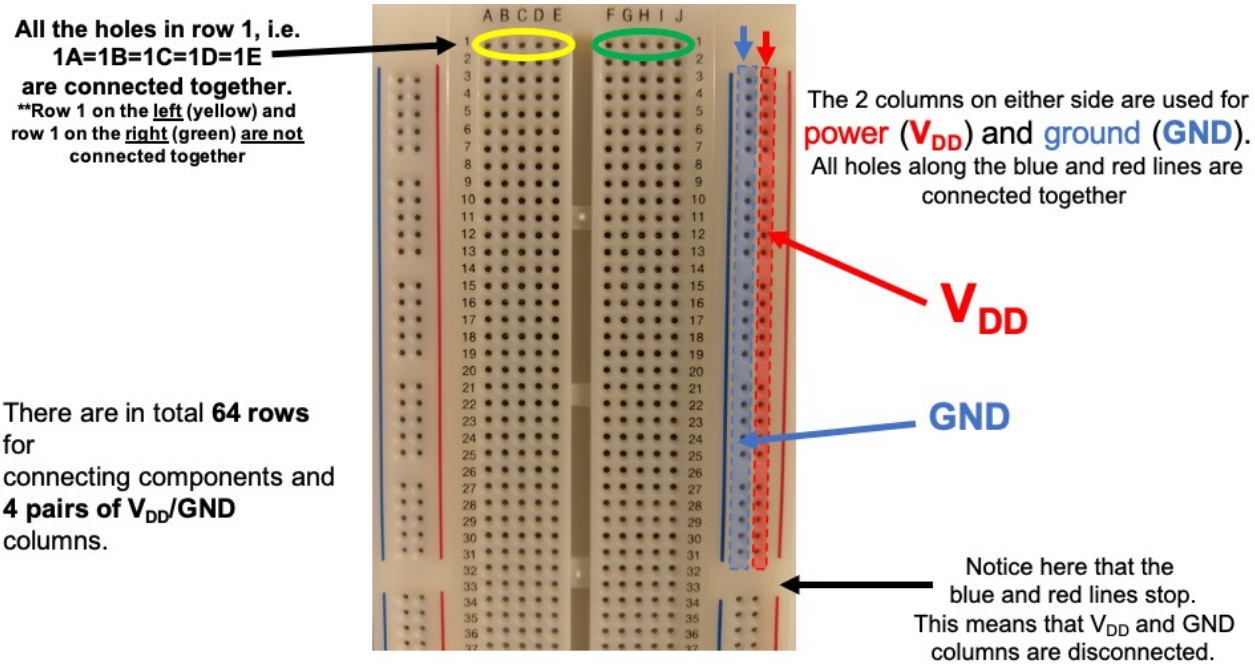
Figure 2 - an mbed board plugged into a breadboard

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.

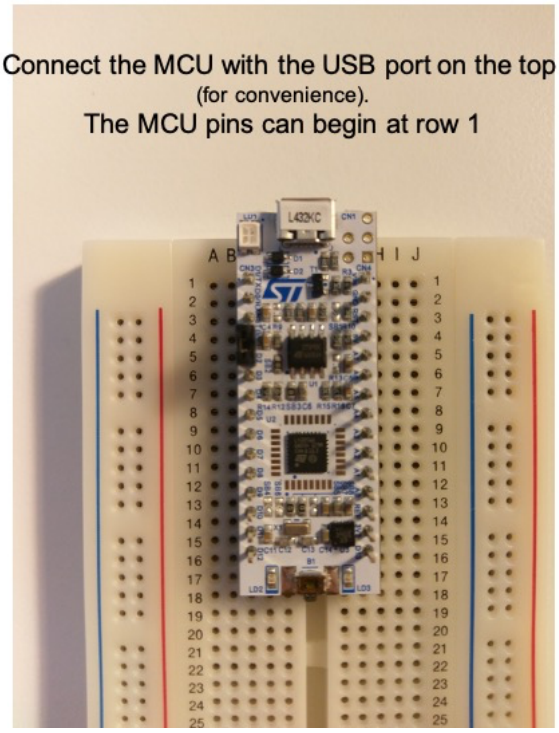
Getting Started with your Development Breadboard

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.



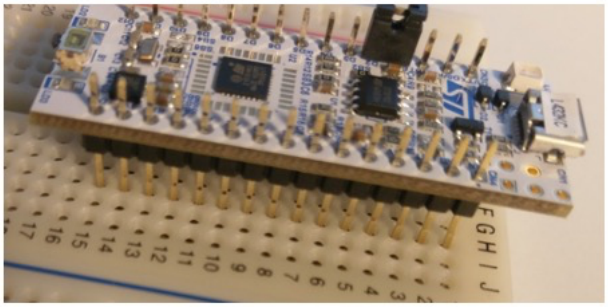
Please study the picture above and understand how a Breadboard works!!

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

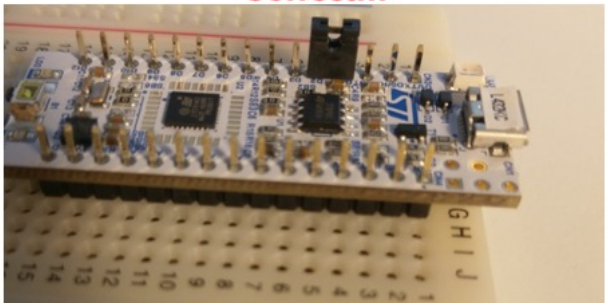


Incorrect!

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

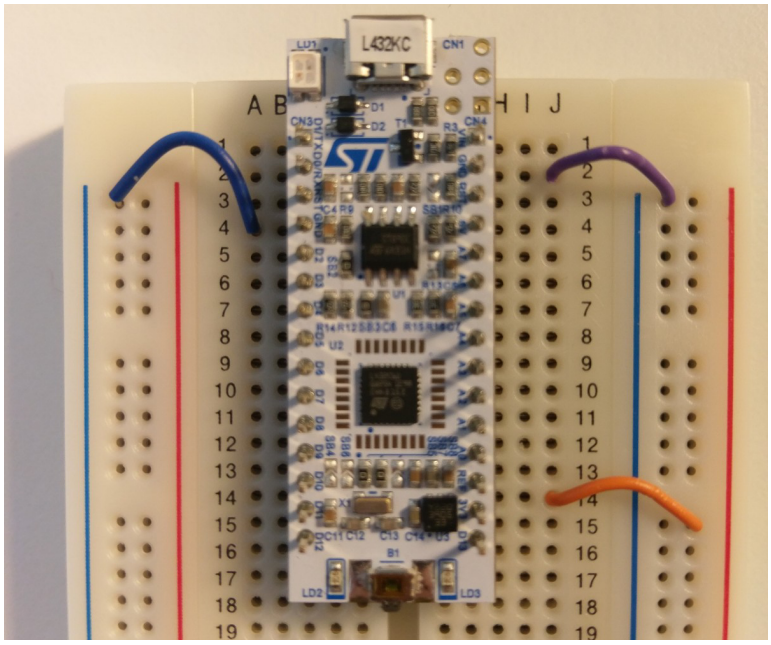


Correct!!!

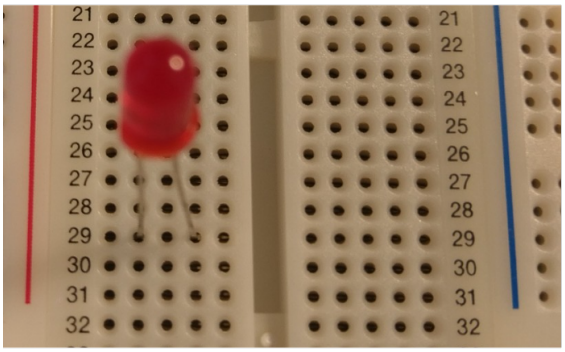


Before creating any complicated prototypes on our breadboard, the very first thing to do is to **connect the VDD (red) and GND (blue) columns with our microcontroller**. By doing so at the beginning, we can create tidy prototypes that can be easily debugged if there are any problems. On your MCU there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

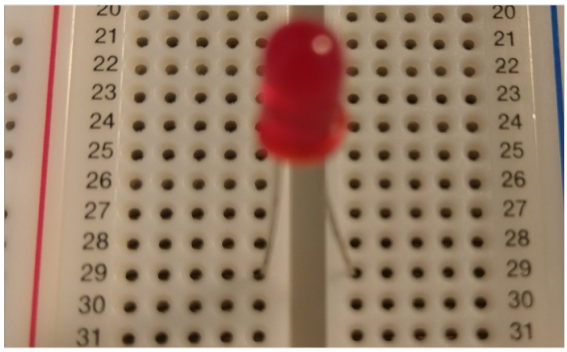
Use wires with appropriate colours to define GND and V_{DD}



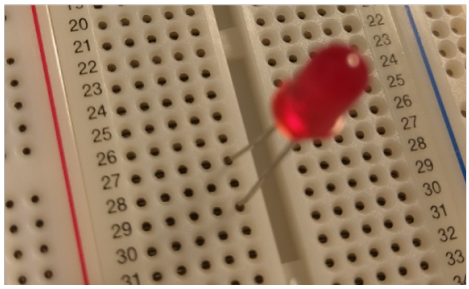
How to connect components on your Breadboard



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

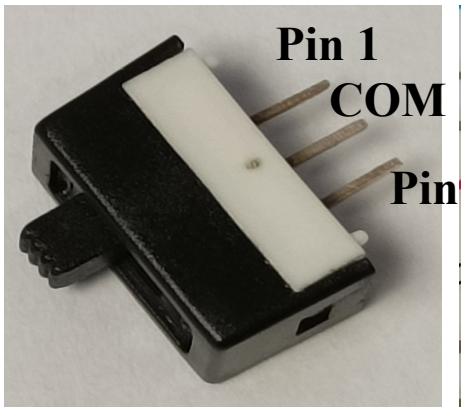


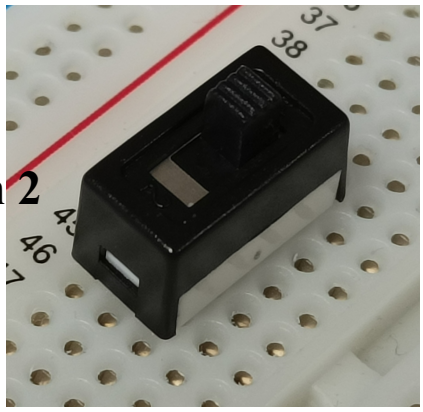
Correct! Both LED legs are connected on the same row, but on columns E and F. Therefore not connected together.

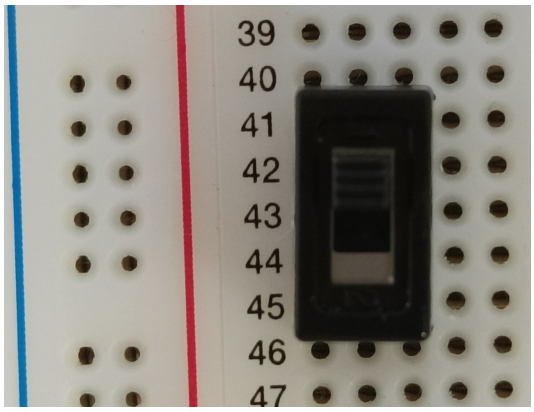


Correct! Both LED legs are connected on different rows. Therefore not connected together.

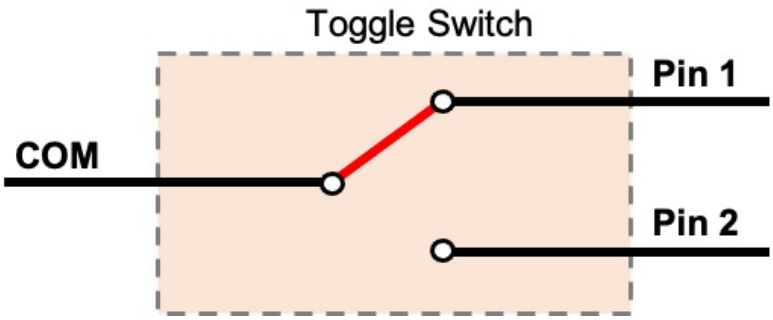
Toggle Switch (with 3 pins)







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.

You can use Zephyr on the Lab computers, or on your laptop. If you choose your laptop, please install Zephyr 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.

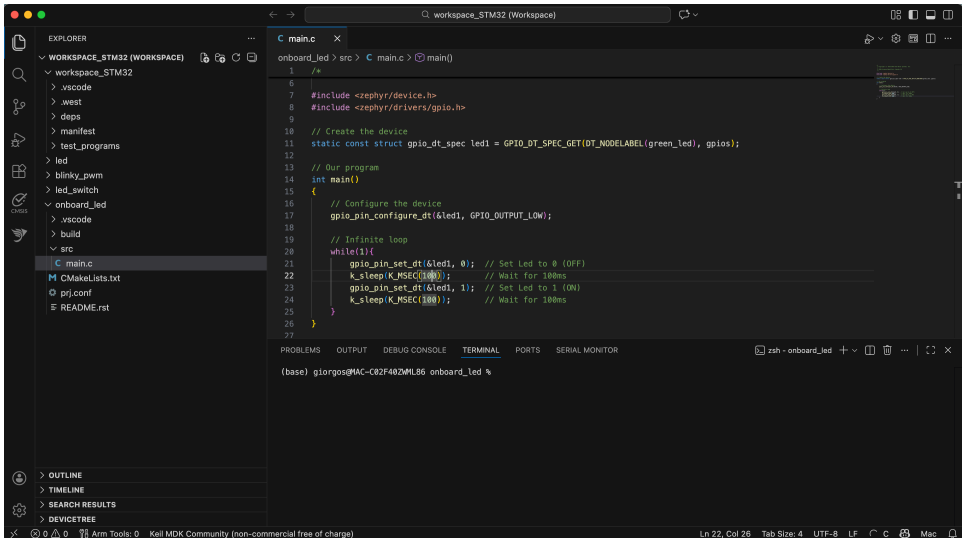


Figure 3 - Screenshot of VS Code compiler environment with Zephyr Workbench

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.

```c
/*
 * 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
    }
}
```

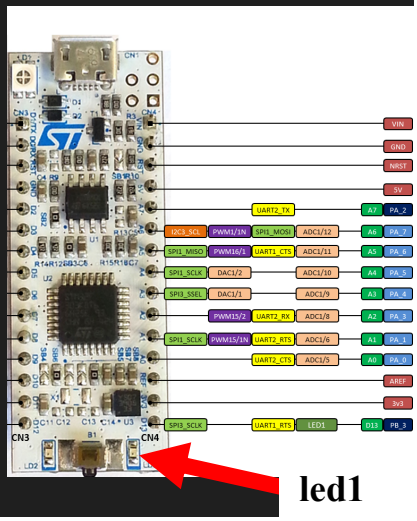
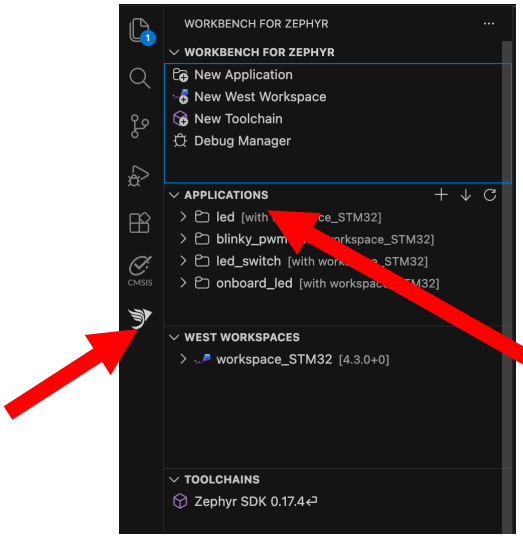


Figure 4 - The code to switch ON and OFF the on-board green led

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.

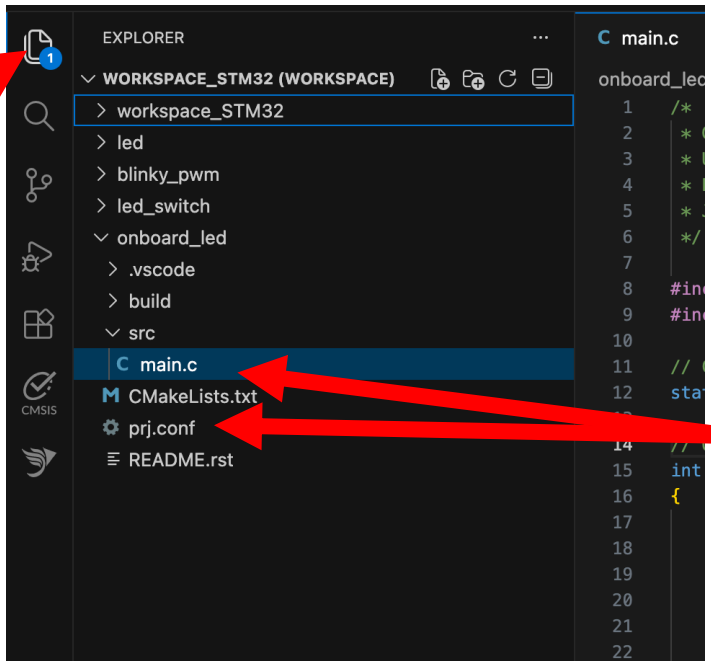


Zephyr Workbench extension

All your programs

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.

Explorer



Files of your application that you will use

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.

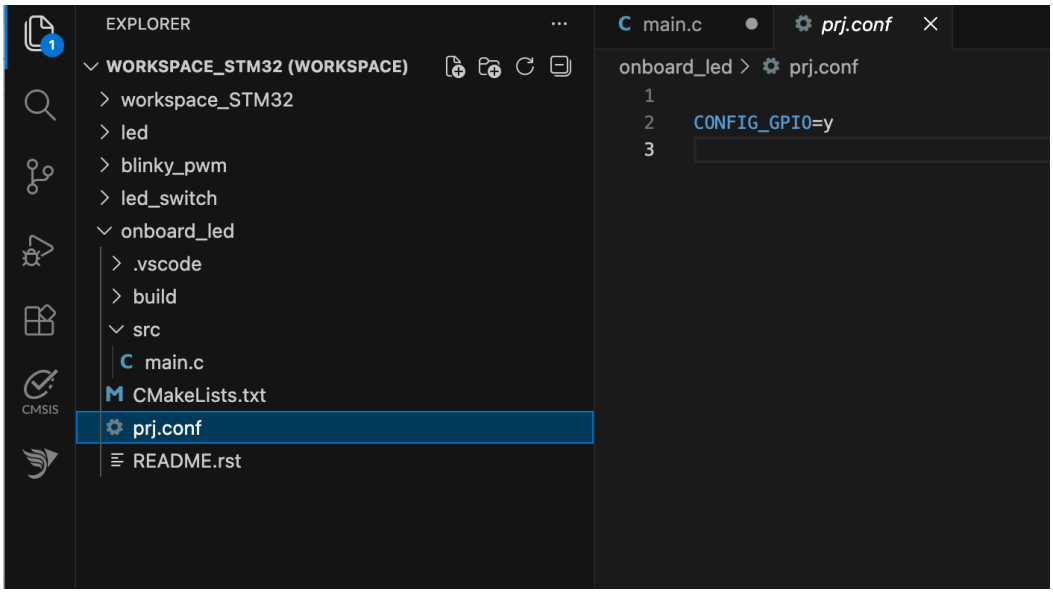
- K_NSEC( ); // delay in nanoseconds
- K_USEC( ); // delay in microseconds
- K_MSEC( ); // delay in milliseconds
- K_SECONS( ); // delay in seconds **
- K_MINUTES( ); // delay in minutes **
- K_HOURS( ); // delay in hours **
- K_FOREVER; // infinite delay

** 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
```

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

Digital Inputs

A GPIO can be also configured as a **Digital Input**. The functionality is similar to that of the digital output. As usual, we need to configure a specific MCU pin to be used as an Input. You can of course use the `compatible = "gpio-leds";` Zephyr drivers, as 'gpio-leds' can be used for both input and output. However, this is not a good practise, as the signal sent by a button to your MCU may not be clean.

We use instead `compatible = "gpio-keys";` Zephyr drivers for buttons. This driver has specific optimisation routines that cleans the signal sent by the button and as a result there are no accidental triggers.

Zephyr Overlay file: we will add the following:

```
/ {

    /* GPIO Outputs */
    // Some Inputs

    /* GPIO Inputs */
    gpio_inputs {
        compatible = "gpio-keys";

        button1: button1{
            gpios = <&gpioa 8 (GPIO_PULL_DOWN | GPIO_ACTIVE_LOW)>; // PA8 -> D9 on the
MCU and configured as GPIO INPUT with LOW
        };
    };
};
```

In the above, we have created a gpio_inputs node and inside it we have added a 'child' node with the name button1.
We use the GPIOA channel 8, which is linked to the pin PA8 or D9 on your MCU board. Furthermore, we are adding a

1. Pull Down Resistor internally: GPIO_PULL_DOWN
2. Activate is with normal polarity: GPIO_ACTIVE_LOW

Main.c
As you see below, there is no difference in creating a GPIO device for an input. It is the same as the output. Just change the node name to the name of the node you have created in the overlay: 'button1'

```
static const struct gpio_dt_spec btn = GPIO_DT_SPEC_GET( DT_NODELABEL(button1),
gpios);
```

The above will create a GPIO device with the variable **btn.** The only thing different we need to do in the Main C++ programme is to initialise the pin as an input:

```
gpio_pin_configure_dt(&btn, GPIO_INPUT);            // Button
```

Below is some code that flashes one of two LEDs, depending on the state of a switch - it gives the opportunity to introduce some additional syntax.

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

// Create our GPIO device
static const struct gpio_dt_spec led1 = GPIO_DT_SPEC_GET( DT_NODELABEL(led1_red), gpios);
static const struct gpio_dt_spec led2 = GPIO_DT_SPEC_GET( DT_NODELABEL(led2_red), gpios);
static const struct gpio_dt_spec btn = GPIO_DT_SPEC_GET( DT_NODELABEL(button1), gpios);

int main() {

    // Initialise the devices
    gpio_pin_configure_dt(&led1, GPIO_OUTPUT_LOW);   // LED 1
    gpio_pin_configure_dt(&led2, GPIO_OUTPUT_LOW);   // LED 2
    gpio_pin_configure_dt(&btn, GPIO_INPUT);            // Button

    int button_state;

    while(1){

        button_state = gpio_pin_get_dt(&btn);
        if ( button_state == 0) {
            gpio_pin_set_dt(&led1, 1);   // Setting the led1 to ON
            k_sleep(K_MSEC(100));           // Wait for 100ms
            gpio_pin_set_dt(&led1, 0);   // Setting the led1 to OFF
            k_sleep(K_MSEC(100));           // Wait for 100ms
        }
        else {
            gpio_pin_set_dt(&led2, 1);   // Setting the led2 to ON
            k_sleep(K_MSEC(100));          // Wait for 100ms
            gpio_pin_set_dt(&led2, 0);   // Setting the led2 to OFF
            k_sleep(K_MSEC(100));          // Wait for 100ms
        }
    }

}
```

And the full overlay:

```
/ {
    /* GPIO Outputs */
    gpio_outputs {
        compatible = "gpio-leds";

        led1: led1{
            gpios = <&gpiob 4 GPIO_ACTIVE_LOW>; // PB4 -> D12
        };
        led2: led1{
            gpios = <&gpiob 5 GPIO_ACTIVE_LOW>; // PB5 -> D11
        };
    };
```

```
    /* GPIO Inputs */
    gpio_inputs {
        compatible = "gpio-keys";

        button1: button1{
            gpios = <&gpioa 8 (GPIO_PULL_DOWN | GPIO_ACTIVE_HIGH)>; // PA8 -> D9 on the
MCU and configured as GPIO INPUT with LOW
        };
    };
};
```

In the above, we configured the following

1. Pin D12 as led1, which is a GPIO Output
2. Pin D11 as led2, which is a GPIO Output
3. Pin D9 as button1, which is a GPIO Input

The above code also introduces the **if** and **else** keywords and the equal operator ==. This causes the block of code which follows the **if** and **else** keywords to be executed if the specific condition is met. In the case of the code above, the condition depends on the variable **button_state**. If this variable receives 0 state (GND), then pin D12 is connected is switched ON and OFF. If **button_state** is not 0 (**else** statement), it means that it will be on state 1. In that case, the LED connected to pin D11 would be switched ON and OFF.

Program 3 - Create a square wave whose frequency depends on the position of a switch

Modify Program 2 above (square wave) so that it will output two possible frequencies, 200 Hz and 500 Hz depending on the position of a switch. You may need to solder wires onto the switch leads in order to conveniently attach to your breadboard. **Refer to the lecture notes on how to connect a switch. How many wires do you actually need? Will you need a resistor as well?**

Since you are using a digital input, it needs to be clearly a 0 (0V) or a 1 (3.3V), that means you will need a pulldown resistor to GND (or you can configure the pin in the overlay with a `GPIO_PULL_DOWN` flag. The same thing can be achieved with a pull-up resistor, i.e. you can use a pullup resistor to V_{DD}, or the pin flag `GPIO_PULL_UP`

Learning Outcomes:

- Learn how to solder wires on a switch.
- Understand the purpose of a Pulldown Resistor and its value. Why is it important?
- Be able to convert frequency to period and properly read it on the oscilloscope.
- Use both Digital Input and Digital Output pins.

Milestone 3a: Show your code and that you can change the square wave frequency by using a switch to a demonstrator.

Modify your program above to output square waves with frequencies 2Hz and 10Hz, depending on the position of the switch. Instead of observing the output signal on the oscilloscope, connect your MCU output to a red LED (use resistors together with the LEDs). Observe the LED flashing at different frequencies.

Milestone 3b: Show your code and that you can change the LED blinking rate by using a switch to a demonstrator.

Playing with the ROM registers directly

The state of each pin is controlled by the memory, ROM, or also known as registers. Each pin has a specific location in the ROM that defines its state, whether this is 1 (HIGH) or 0 (LOW).

In general, the pins for MCUs are bundled into groups. For our NUCLEO L432 there are several groups of pins labelled by the manufacturer as GPIOA, GPIOB, GPIOC, GPIOD, GPIOE, GPIOH. Each of these groups is allocated a range of memory addresses (mail trays as discussed in the class), and each memory address is linked to specific properties of each GPIO.

0x4800 1C00 - 0x4800 1FFF	1 KB	GPIOH	*Section 8.4.12: GPIO register map*
0x4800 1400 - 0x4800 1BFF	2 KB	Reserved	-
0x4800 1000 - 0x4800 13FF	1 KB	GPIOE(2)	*Section 8.4.12: GPIO register map*
0x4800 0C00 - 0x4800 0FFF	1 KB	GPIOD(2)	*Section 8.4.12: GPIO register map*
0x4800 0800 - 0x4800 0BFF	1 KB	GPIOC	*Section 8.4.12: GPIO register map*
0x4800 0400 - 0x4800 07FF	1 KB	GPIOB	*Section 8.4.12: GPIO register map*
0x4800 0000 - 0x4800 03FF	1 KB	GPIOA	*Section 8.4.12: GPIO register map*

As you can see from the above, the GPIOA is allocated the addresses 0x4800 0000 to 0x4800 03FF, which is in total 16383 memory addresses/locations.
For the GPIOA, each address contains properties for the pins, like the ON/OFF state, the information whether a pin is an GPIO_OUTPUT or GPIO_INPUT etc.
The most relevant information for the ON/OFF state is the following,

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x14	**GPIOx_ODR** (where x = **A..E,H**)	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	OD15	OD14	OD13	OD12	OD11	OD10	OD9	OD8	OD7	OD6	OD5	OD4	OD3	OD2	OD1	OD0
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

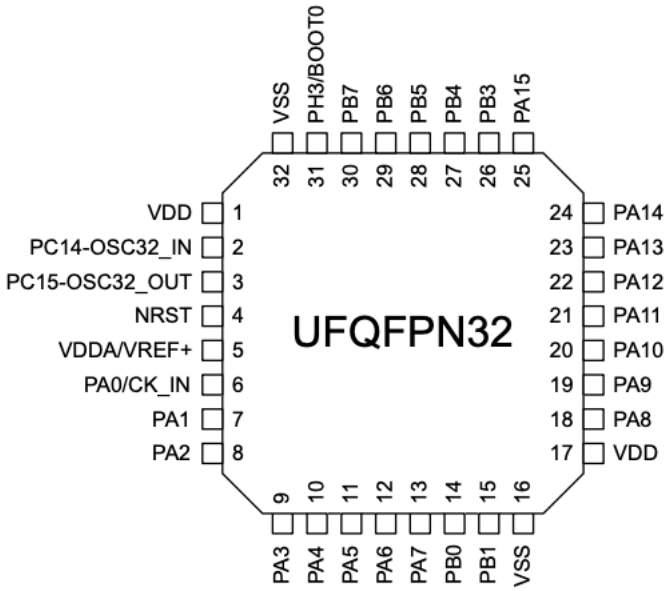
The **GPIOx_ODR** stands for **Output Data Register**, which is essentially the output state of the pin. As you can see in this register, there are in total 32 bits, however, only 16 can modify a pin, from 0-16. The offset, is an offset from the base address of the GPIO we want to access.
For example, for GPIOA the memory address/register that controls the ON/OFF state is 0x4800 0014.

The above is just the method the manufacturer uses for various types of MCUs in their product range. This doesn't mean that we have so many pins on or NUCLEO L432.

On the NUCLEO L432KC on GPIOA we have ,

- GPIOA: PA0 – PA15
- GPIOB: PB0 – PB7
- GPIOC: PC14 – PC15
- GPIOH: PH3

We will be using primarily the GPIOA and GPIOB. See image below to get an understanding of how this looks on the MCU.


VSS
PH3/BOOT0
PB7
PB6
PB5
PB4
PB3
PA15
32
31
30
29
28
27
26
25
VDD 1
PC14-OSC32_IN 2
PC15-OSC32_OUT 3
NRST 4
VDDA/VREF+ 5
PA0/CK_IN 6
PA1 7
PA2 8
UFQFPN32
24 PA14
23 PA13
22 PA12
21 PA11
20 PA10
19 PA9
18 PA8
17 VDD
9
10
11
12
13
14
15
16
PA3
PA4
PA5
PA6
PA7
PB0
PB1
VSS

Using registers to control a *Seven Segment Display*

In Program 1 you switched on and off a pair of LEDs. LEDs are often packaged together, to form patterns, digits, alphanumeric characters etc. A number of standard groupings of LEDs are available, including a seven-segment display, which is rather versatile as by lighting different combinations of the seven segments, each of which is an individual LED, all numerical digits can be displayed, along with quite a few alphanumeric characters.

Figures 5a,b below shows the seven segments labelled A-G. A decimal point (DP) is usually included in the display. The seven segment display we will use (you can get the datasheet from the course Moodle site), has 10 pins, which are connected to the seven segments as shown in Figure 5b.

The 7 LEDs have a common cathode terminal, pins 3 or 8 on the package, the other pins connect to individual segments, so segment A is connected to pin 7, segment E is connected to pin 1.

For a common cathode display, the cathode is connected to ground of the MCU, and by applying a significantly large positive voltage to the LED, as can be achieved with an MCU DigitalOut command, a given LED can be illuminated.

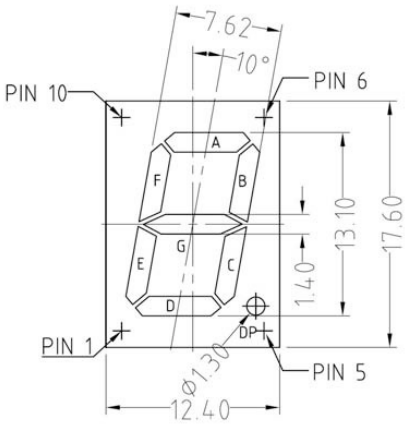


Figure 5a - Labelling of the 7 segments in a seven segment display

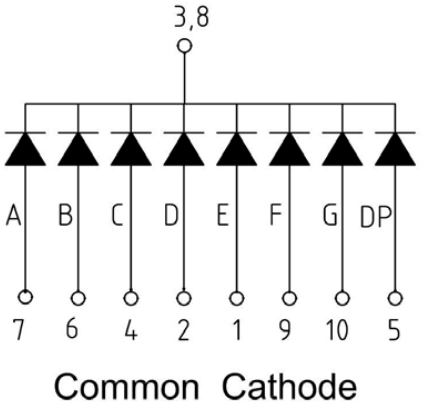


Figure 5b - the pin allocation of the Avago HDSP-C5L3 7 segment display

All the segments can be written in a single byte (8 bits), for example in the sequence

(MSB) DP G F E D C B A (LSB)

If you want to represent the digit "0", it would be necessary to illuminate segments A, B, C, D, E, F. Using the above approach, this could be represented by the byte 00111111 (ie DP and segment G not illuminated, all other segments illuminated). This can be represented in hexadecimal notation as 0x3F (0x tells us it is a hexademical notation, 3F is the representation for 0011 1111).

The advantage of this approach is that we can connect the 7 segment display to 7 digital output pins of the MCU, and then use a new mbed API class **BusOut**, which allows you to group a set of digital outputs together, and to write a single word direct to it.

To demonstrate this, using the breadboard connect pin 3 or 8 of the seven-segment display to the Gnd connection of the MCU, and the other pins of the seven segment display to MCU pins that can be configured as digital outputs. To have a direct control of the 7 segment LED through the memory registers, we will connect the LED to pins that belong to ONLY 1 GPIO, i.e GPIOA, or GPIOB. Try and do this in a reasonably logical way, such as that outlined in the table below.

MCU pin	GND	PA0	PA1	PA2	PA3	PA4	PA5	PA6	PA7
Display pin	3 or 8	7	6	4	2	1	9	10	5
Display segment	-	A	B	C	D	E	F	G	DP

Produce a table in your lab book like that below which shows the hexadecimal number you will write to the 8 MCU pins to generate the display values 0-9 (0 and 1 are provided to get you started)

Display Value	0	1	2	3	4	5	6	7	8	9
Segment drive in digital	0011 1111	0000 0110								
Segment drive in hex	0x3F	0x06								

Having worked out which values to write to the MCU pins, the issue of coding this can be considered with reference to the program below.

Since we are controlling the LEDs directly through the memory registers, **we do not need an Overlay file**.

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

// Create our GPIO PORT device
// We will directly talk to the port memory register
static const struct device *const led7 = DEVICE_DT_GET( DT_NODELABEL(gpioa) );

int main() {

    // In this case we only need to configure/initialise 8 pins from that port
    // The gpioa port has 16 pins. We only need 8 of them
    for (int pin = 0; pin < 8; pin++) {
        gpio_pin_configure(red_ led7, pin, GPIO_OUTPUT_LOW);
    }

    // Create an array of HEX values
    static const uint8_t digits[10] = {0x3F, 0x06, *ADD HERE THE REST* };

    while(1){
        for (int i = 0; i<10; i++){
            gpio_port_set_masked(led7, 0xFF, digits[i]);
            k_sleep(K_MSEC(1000));
        }
    }
}
```

The main command we use here to talk to the memory registers is:

```
gpio_port_set_masked(led7, 0xFF, digits[i]);
```

The function `gpio_port_set_masked()` gets 3 input values:

1. The device variable, which is the GPIOA that was defined at the start as 'led7'
2. The bits we need to change in the memory address linked to **GPIOA**. This is called **bit-mask**. As we saw above, there are 16 bits in total there, but **we want to only change the first 8**. Therefore, we use 0xFF which is equivalent to the binary number **0000 0000 1111 1111 = 0xFF**
3. And lastly the value we want to change the bits to. Here we have created an array called `static const uint8_t digits[10]` where we have stored in advance all 10 LED digits in hex number. A for-loop iterates through all these digits writing on the memory address directly the value we asked for.

Question: If we wanted to work with pins PA8-PA15, how can we do it?

Program 4 - Drive a seven-segment display to display 0-9 repeatedly

Based on your table to determine the hexadecimal values required to produce the digital 0-9 on the seven-segment display, modify the code above to count from 0-9 repeatedly.

Learning Outcomes:

- Properly connect a 7-segment display to their MCU. **Remember**, this display is made out of LEDs and a suitable resistor needs to be connected in series with each LED.
- Understand memory registers and how they are linked to the pins.
- Be able to convert a number from Decimal to Binary to Hex.

Milestone 4: Show your code and the working display to a demonstrator.

Program 5 - Display the letters H E L L O in turn on a seven-segment display

Modify your code from Program 4 to have the seven-segment display flash the letters H E L L O in turn.

Learning Outcomes:

- Properly connect a 7-segment display to their MCU. **Remember**, this display is made out of LEDs and a suitable resistor needs to be connected in series with each LED.
- Understand memory registers and how they are linked to the pins.
- Be able to convert a number from Decimal to Binary to Hex.

Milestone 5: Show your code and the working display to a demonstrator.

Analogue Output

As you can see from Figure 1 of this sheet, the MCU board can be configured to generate a number of outputs, including analogue outputs on pin A3 (PA4) or A4 (PA5).

Following the previous section, you may have noticed the PA5 and PA5 pins are configured by default and linked to GPIOA. However, as we discussed in the class, each MCU pin can be configured to do other things as well (with a few exceptions).

As you can see in the Datasheet https://www.st.com/resource/en/datasheet/stm32l432kc.pdf , on Table 14 (page 52) the PA5 and PA5 have **Alternate and Additional functionalities.** One of them is **DAC1_OUT1 and DAC1_OUT2** respectively.

Pin Number	Pin name (function after reset)	Pin type	I/O structure	Notes	Pin functions	
UFQFPN32					Alternate functions	Additional functions
9	PA3	I/O	TT_a	-	TIM2_CH4, USART2_RX, LPUART1_RX, QUADSPI_CLK, SAI1_MCLK_A, TIM15_CH2, EVENTOUT	OPAMP1_VOUT, COMP2_INP, ADC1_IN8
10	PA4	I/O	TT_a	-	SPI1_NSS, SPI3_NSS, USART2_CK, SAI1_FS_B, LPTIM2_OUT, EVENTOUT	COMP1_INM, COMP2_INM, ADC1_IN9, DAC1_OUT1
11	PA5	I/O	TT_a	-	TIM2_CH1, TIM2_ETR, SPI1_SCK, LPTIM2_ETR, EVENTOUT	COMP1_INM, COMP2_INM, ADC1_IN10, DAC1_OUT2
12	PA6	I/O	FT_a	-	TIM1_BKIN, SPI1_MISO, COMP1_OUT, USART3_CTS, LPUART1_CTS, QUADSPI_BK1_IO3, TIM1_BKIN_COMP2, TIM16_CH1, EVENTOUT	ADC1_IN11

The default settings that convert the PA4 and PA5 into a Digital to Analogue (DAC) converter, are already preprogramed in the Zephyr drivers for the NUCLEO L432KC, but they are not activated. Therefore, to activate the DAC we simply need to add in the Overlay file:

Overlay

```
/ {

    /* THIS IS THE OVERLAY MAIN NODE */
    /* HERE YOU ADD USER CUSTOM DEVICES */
    /* LIKE LEDS OR BUTTONS ETC, */
    /* SIMILAR TO WHAT WE DID BEFORE */

};

/* HERE YOU CAN MODIFY EXISTING SYSTEM DEVICES */
&dac1 {
    status = "okay";       // This “okay” enables the &dac1 node device.
};
```

In the Overlay above, all we do is modifying the existing `&dac1` system node. Our NUCLEO L432KC is set to `status = "disabled";` by default, as its main functionality is GPIO. In Zephyr, the user's Overlay file has priority over the system's overlay file. This gives us the flexibility to modify system parameters.

What happens internally is that, when we enable `&dac1`, the GPIO functionality is disconnected from pin PA4 and PA5 and the DAC functionality is connected instead. As you may notice from the Table above, these pins can be also configured as serial communication pins (SPI, USART) or timers (TIM).

Prj.conf
In the config, to use the DAC we need to add the DAC drivers.

```
CONFIG_DAC=y
```

Program to Control the output of PA4
Below, you will find a program that is used for sending a DC Analogue signal to the pin PA4 (A3).

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

// Create our DAC device
static const struct device *const dac_dev = DEVICE_DT_GET( DT_NODELABEL(dac1));
static const struct dac_channel_cfg dac_ch_cfg = {
        .channel_id  = 1,
        .resolution  = 12
};

int main() {

    // Initialise the devices
    dac_channel_setup(dac_dev, &dac_ch_cfg);   // DAC device

    int V1 = 2.5/3.3 * 4095;    // Variable to output 2.5 V on the DAC
    while(1){
        dac_write_value(dac_dev, 1, V1);
        k_sleep(K_MSEC(100));           // Wait for 100ms
    }
}
```

The `dac_write_value` takes 3 input values:

1. Device variable: `dac_dev`
2. DAC Channel: `1`
3. Output Voltage: `V1` Note: This is an integer number with values from 0 to 2^N-1, where N is the DAC's number of bits. Value of 0 represents 0V and 2^{12}-1 = 4095 represents 3.3V.

Program 6 - Creating output voltages and waves

First confirm the output voltage in the above code are as what you would expect. Then, modify the code, why adding the `while(1){ }` loop to output constant voltages of 0.5 V, 1.0 V, 2.0 V and 2.5 V. Use an oscilloscope to observe your output signal.

Learning Outcomes:

- Learn to use the Digital to Analogue functionality of MCU.
- Configuring Zephyr to switch pin functionality from GPIO to DAC.
- Be able to observe the output voltages on an oscilloscope.

Milestone 6A: Show your code and the output voltages to a demonstrator.

Using a **for** loop, generate a **100 Hz sawtooth** waveform with **minimum and maximum voltages of 0 V and 3 V** respectively. You will have to work out how many "steps" to have in your sawtooth and then figure out their duration so that the frequency is correct.

Learning Outcomes:

- Learn to use the Digital to Analogue functionality of MCU.
- Use the Digital to Analogue to create a sequence of output voltages.
- Create a sequence with the right frequency and max voltage.
- Be able to observe the output wave on an oscilloscope.

Milestone 6B: Show your code and the resulting waveform on an oscilloscope to a demonstrator.