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 14.

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,

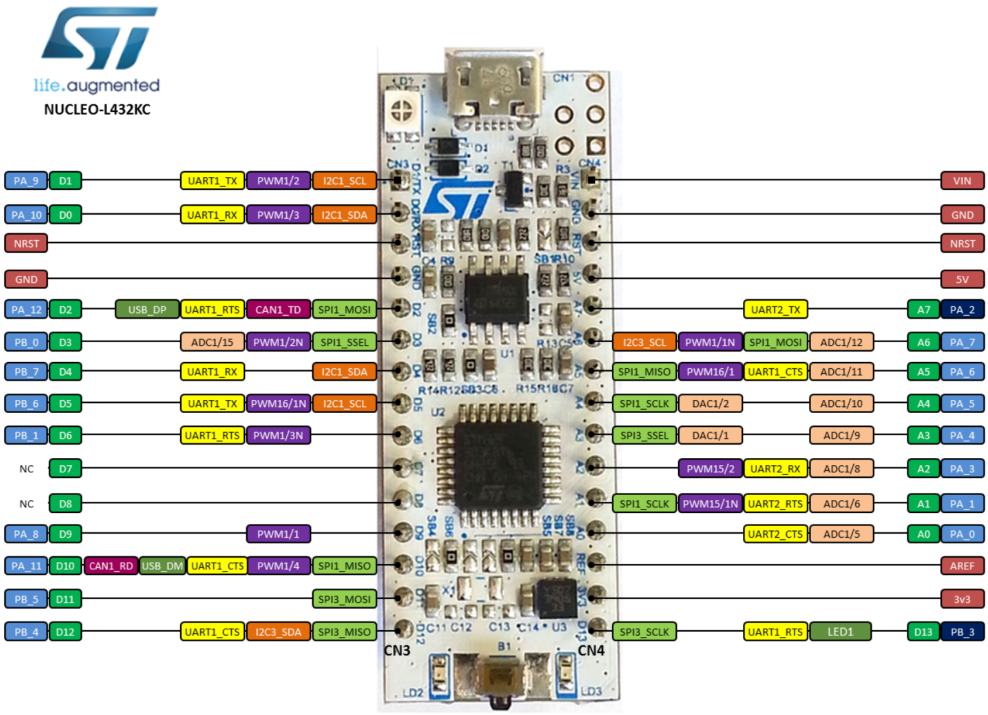


Figure 1 - Nucleo microcontroller development board and some of the pin possibilities.

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 open source compiler environments like **Zephyr** or **FreeRTOS**. These come with 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. Once compiled, programs are easily flashed 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.**

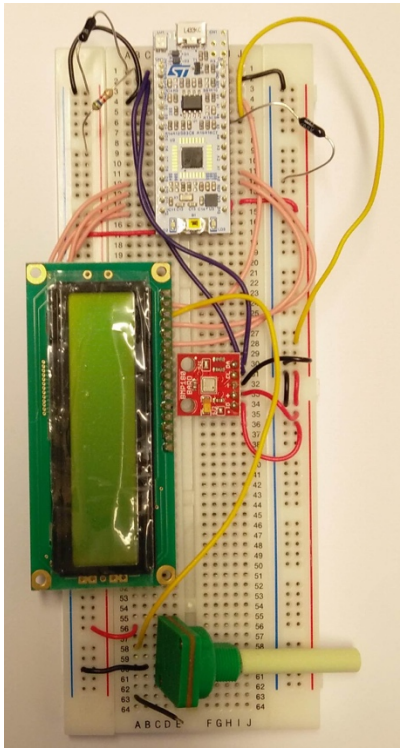


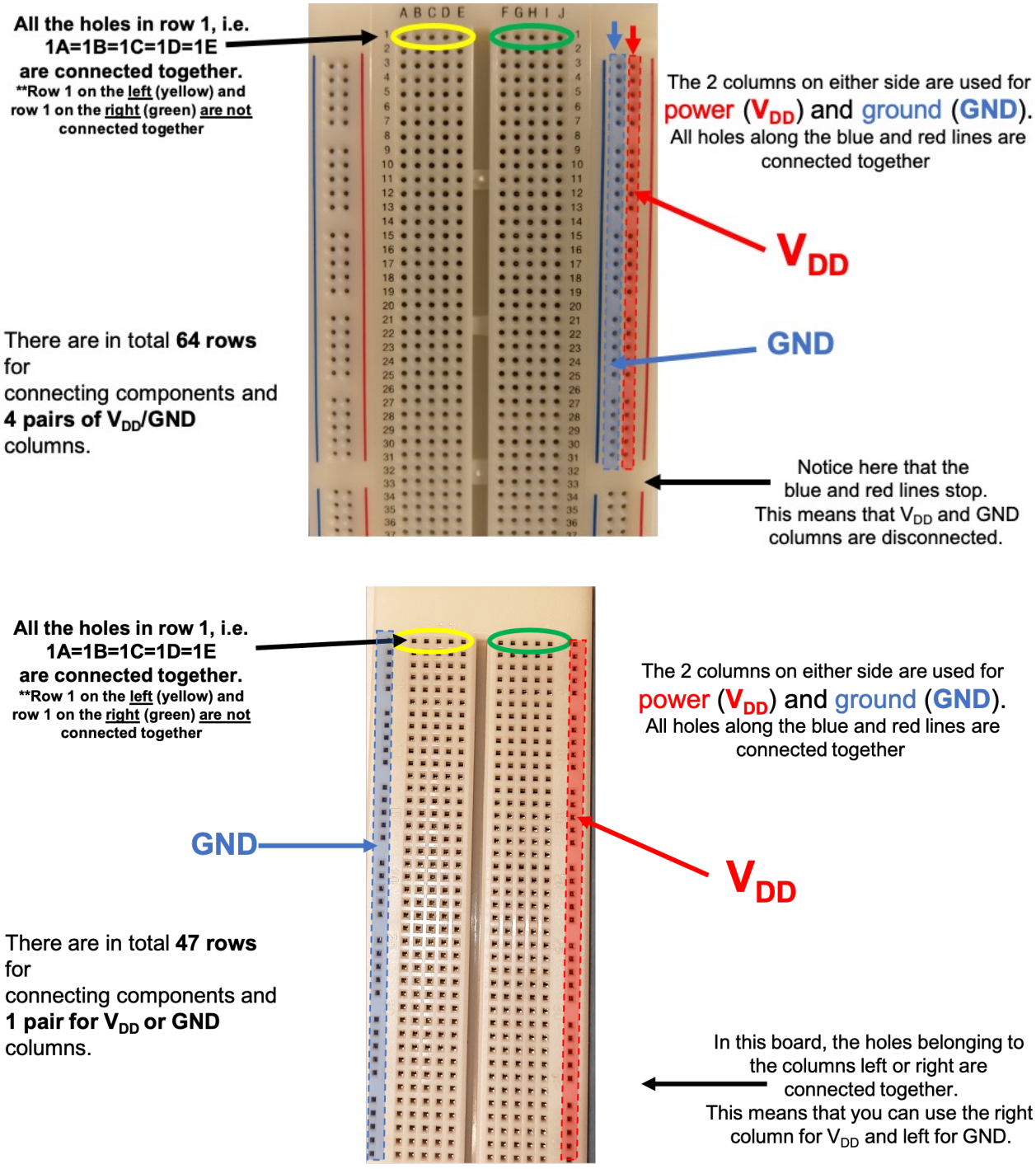
Figure 2 - an mbed board plugged into a breadboard

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.

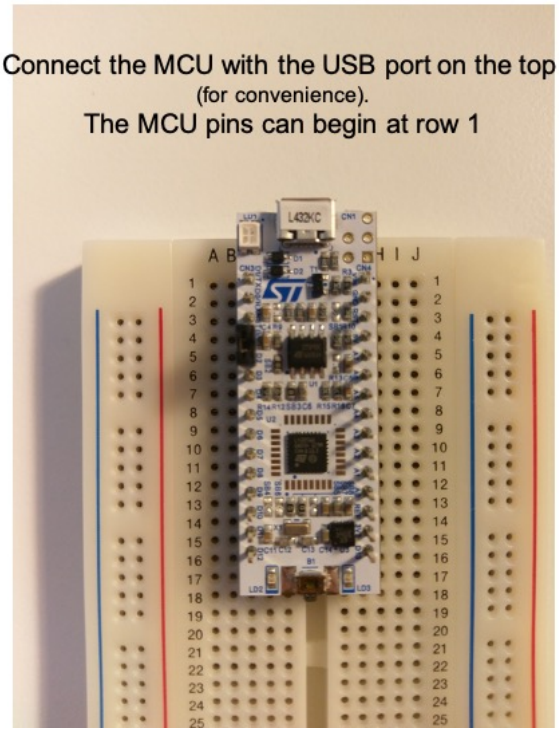
Getting Started with your Development Breadboard

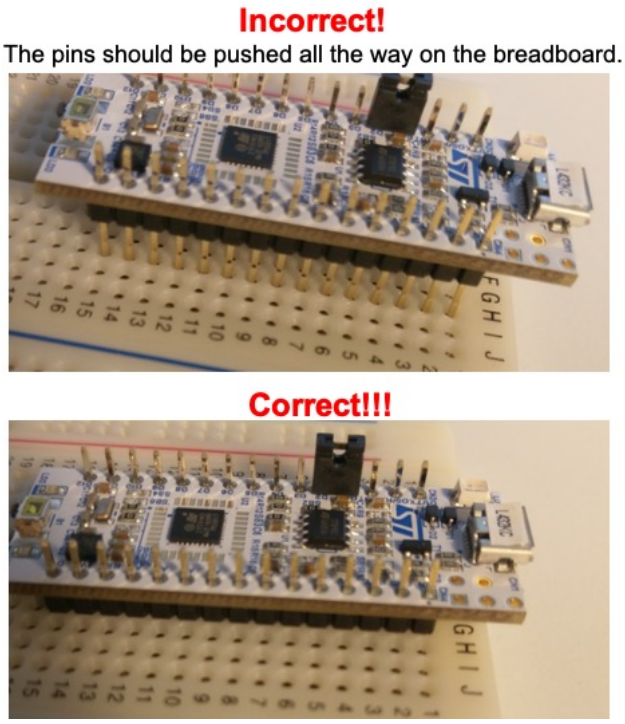
When you arrive at the lab on your first session, you will get a Development Breadboard (also your STM32L432KCU6 MCU). Your breadboard will look like one from the pictures below, it is a standardized component and particularly useful for developing prototype circuits.



Please study the picture above that corresponds to your breadboard and understand how it works!!

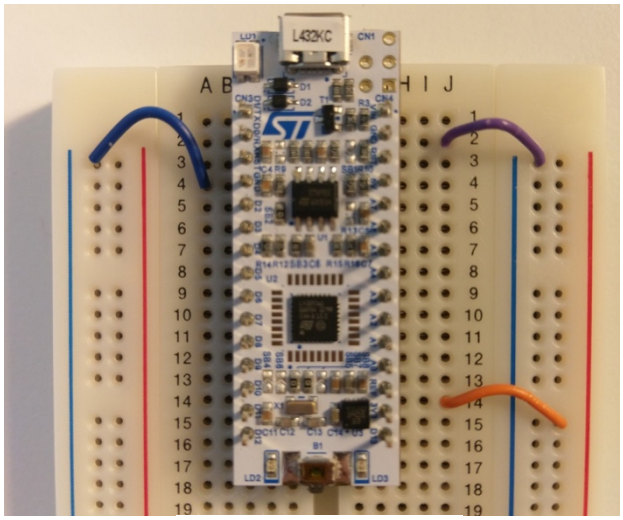
****Mount your microcontroller like is shown in the picture below.**



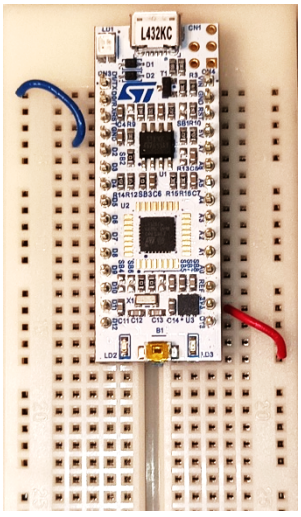


Before creating any complicated prototypes on our breadboard, the very first thing to do is to **connect the VDD (red) and GND (blue) columns with our microcontroller**. By doing so at the beginning, we can create tidy prototypes that can be easily debugged if there are any problems. On your MCU there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

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

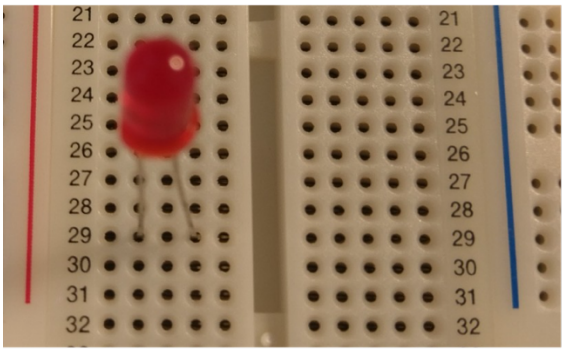


Breadboard A

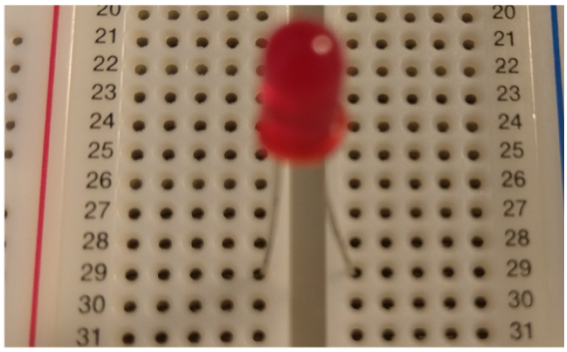


Breadboard B

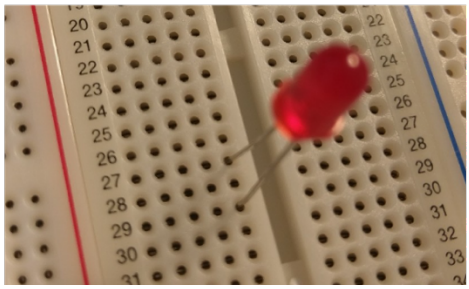
How to connect components on your Breadboard



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

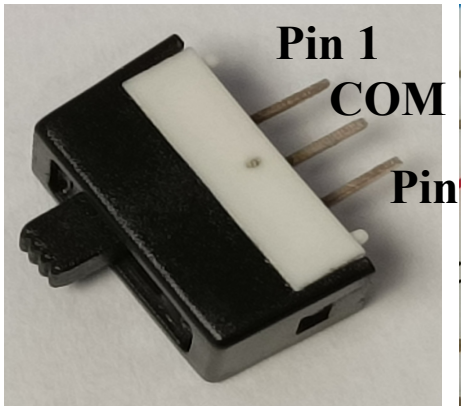


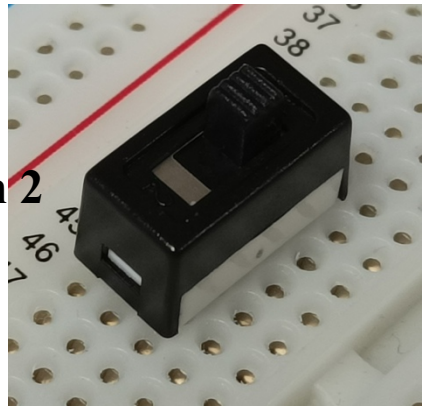
Correct! Both LED legs are connected on the same row, but on columns E and F. Therefore not connected together.

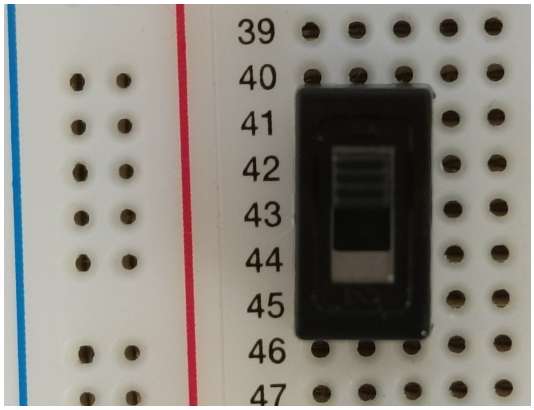


Correct! Both LED legs are connected on different rows. Therefore not connected together.

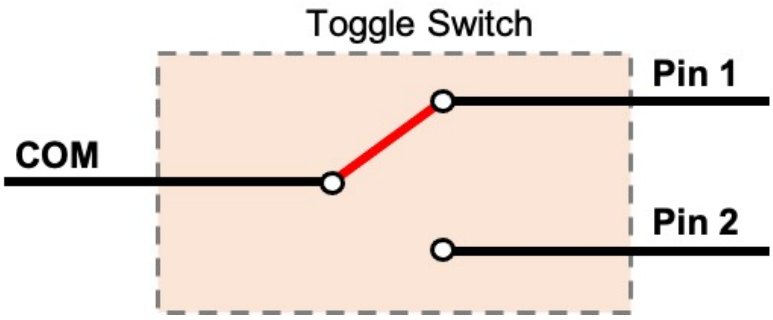
Toggle Switch (with 3 pins)







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,


Toggle Switch
Pin 1
COM
Pin 2

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.

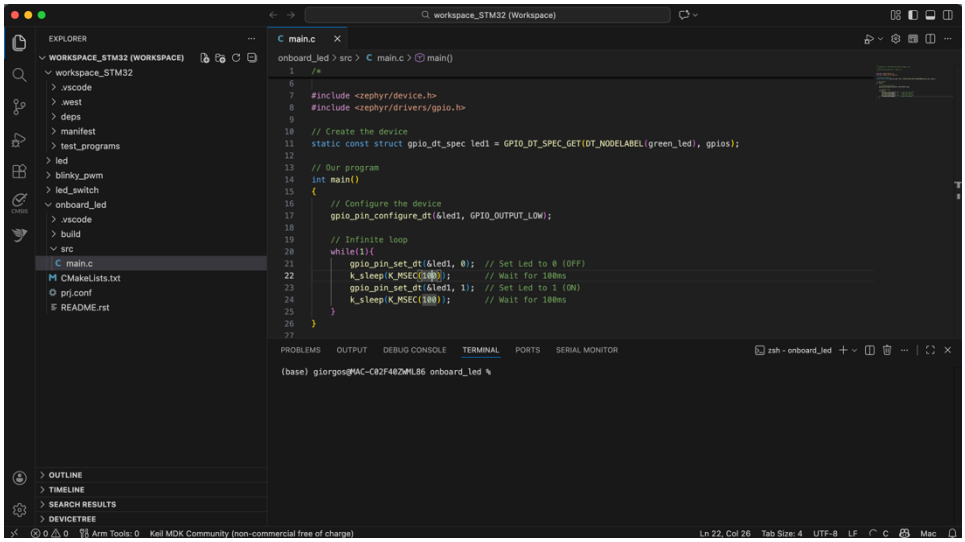


Figure 3 - Screenshot of VS Code compiler environment with Zephyr Workbench

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
    }
}
```

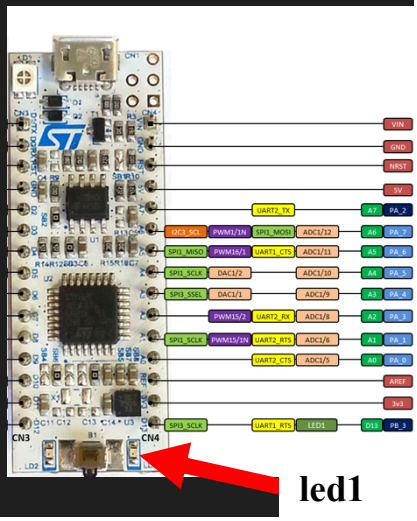
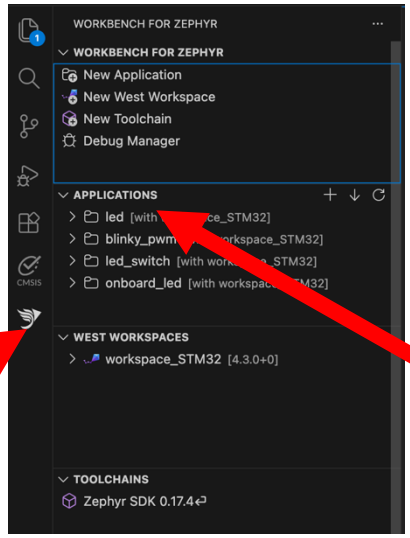


Figure 4 - The code to switch ON and OFF the on-board green led

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.

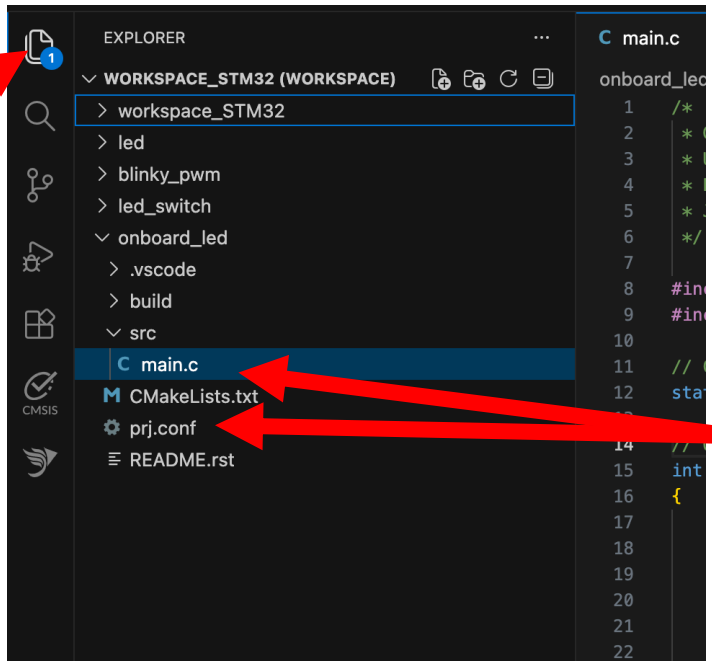


Zephyr Workbench extension

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.

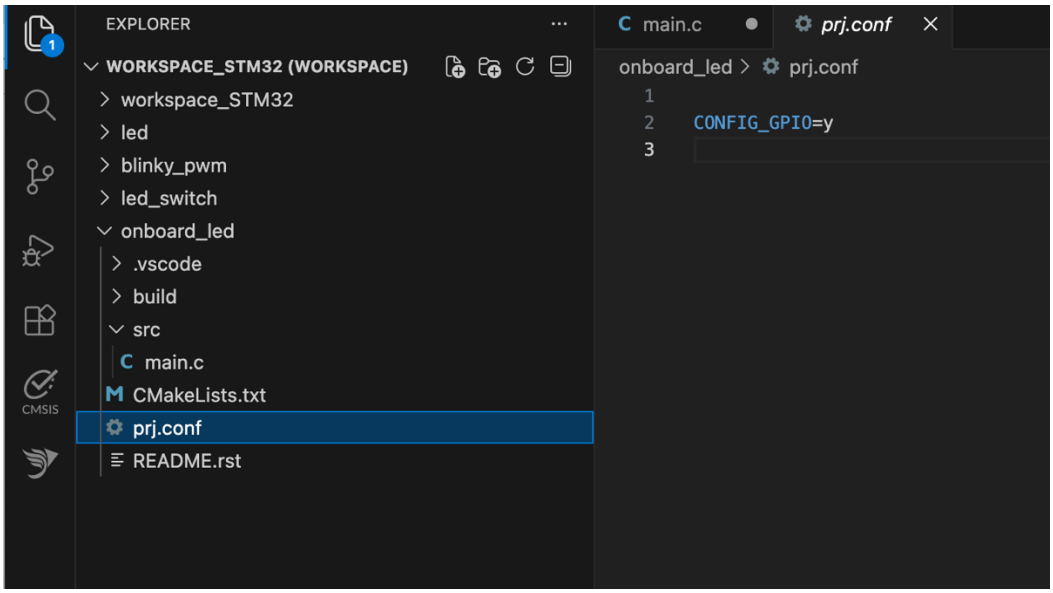
- K_NSEC( ); // delay in nanoseconds
- K_USEC( ); // delay in microseconds
- K_MSEC( ); // delay in milliseconds
- K_SECONS( ); // delay in seconds **
- K_MINUTES( ); // delay in minutes **
- K_HOURS( ); // delay in hours **
- K_FOREVER; // infinite delay

** 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
```

Acquiring the state of the button (btn) is done by the command:
`button_state = gpio_pin_get_dt(&btn);`

Below is some code that flashes one of two LEDs, depending on the state of a switch - it also gives us 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
        }
    }

}
```

Overlay

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

        led1_red: led1_red{
            gpios = <&gpiob 4 GPIO_ACTIVE_LOW>; // PB4 -> D12
        };
        led2_red: led2_red{
            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 PB4 (D12) as led1_red, which is a GPIO Output
2. Pin PB5 (D11) as led2_red, which is a GPIO Output
3. Pin PA8 (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

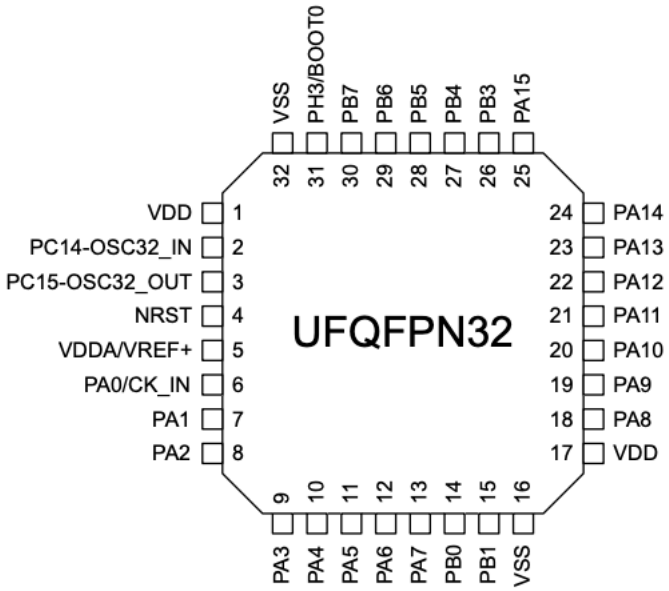
The **GPIOx_ODR** stands for **Output Data Register**, which is essentially the output state of the pin. As you can see in this register, there are in total 32 bits, however, only 16 can modify a pin, from 0-16. The offset, is an offset from the base address of the GPIO we want to access.
For example,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

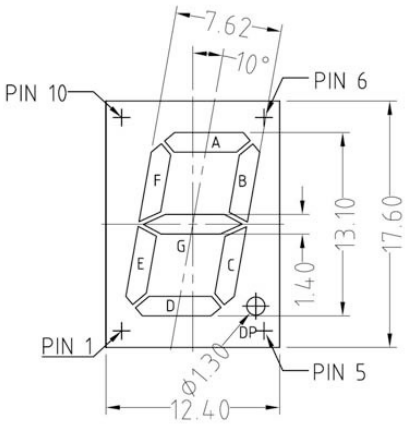


Figure 5a - Labelling of the 7 segments in a seven segment display

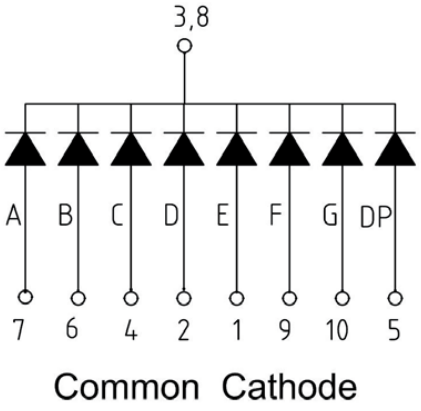


Figure 5b - the pin allocation of the Avago HDSP-C5L3 7 segment display

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).

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** technique. As we saw above, there are 16 bits in total available for the GPIOA. However, **we only want to change the first 8**, because we connect our 7-segment LED on PA0-PA7. Therefore, we use 0xFF which is equivalent to the binary number **0000 0000 1111 1111 = 0x00FF = 0xFF**

   For example, if we want to connect our 7-segment LED on pins PA0-PA5 and PA10, PA13 we should use as a mask: **0010 0100 0011 1111 = 0x243F**
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 PA4 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 by default to `status = "disabled";`, as its main functionality of the specific pins is set to GPIO. Note, 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 is as what you would expect, i.e. 2.5V. 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.

Analogue Input

The real world is analogue, and many sensors have analogue outputs, yet it is essential that a microcontroller have these signals available in a digital form. The ability to convert from analogue inputs to digital signals is so important that the Nucleo L432KC board can handle up to 10 analogue inputs at any given time. The Application Programming Interface for handling analogue input signals is similar in format to many of those we have looked at already.
As can be seen from the manual (https://www.st.com/resource/en/datasheet/stm32l432kc.pdf) the following pins can have Additional Function (AF) that can be ADC:

- **PA0-7** linked to ADC1 Channels 5-12
- **PB0-1** linked to ADC1 Channels 15-16

A simple way to observe the operation of the analogue input is to use a potentiometer (variable resistor) to vary the drive voltage being applied to an LED. Figure 6 (left) shows a 10 kΩ potentiometer connected to an analogue input of the MCU. The potentiometer is just acting as a voltage divider – the resulting voltage will change the brightness of the LED. The analogue input voltage is converted to a digital value, and then back to an analogue voltage to drive the LED. In the real world, we would do some computation in the digital domain, but this is simple example

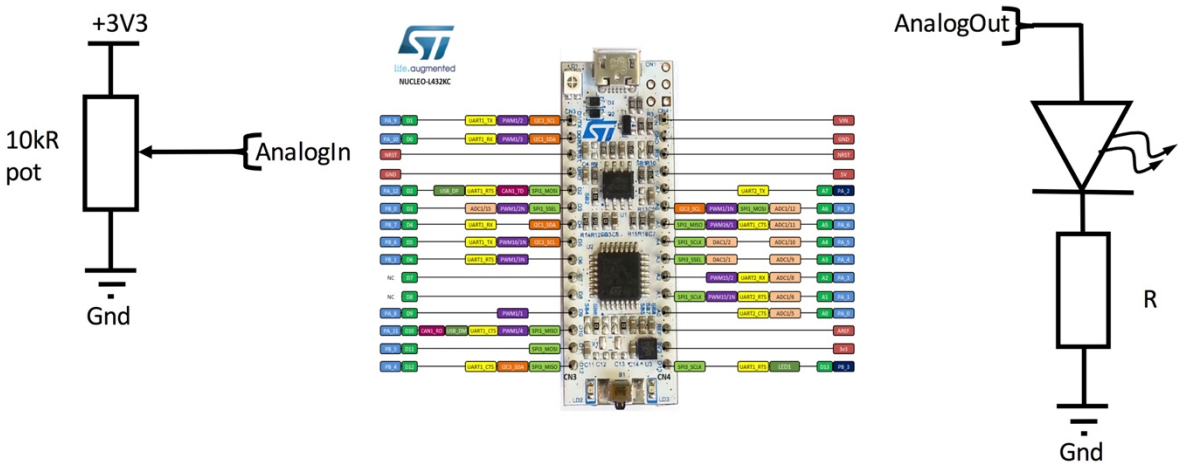


Figure 6 – Connection of 10 kΩ potentiometer and LED to demonstrate operation of analogue in and analogue out functions

Connect the circuit of Figure 6, then implement the code below to vary the brightness of the LED. You must first replace the ? with appropriate pin names of your choice.

Overlay file in: boards/nucleo_l432kc.overlay

```
/{
    /* This is the main Overlay Node. You can add any custom devices, like LEDs or
Buttons etc.. */

};

&dac1 {
    status = "okay";
};

/* Enabling and configuring the ADC Channel 5 */
```

```
&adc1 {
    status = "okay";
    st,adc-clock-source = "SYNC";
    st,adc-prescaler = <1>;
    #address-cells = <1>;
    #size-cells = <0>;

    channel_5: channel@5 {       // Channel 5 linked to pin PA0
        reg = <5>;
        zephyr,gain = "ADC_GAIN_1";
        zephyr,reference = "ADC_REF_INTERNAL";
        zephyr,acquisition-time = <ADC_ACQ_TIME_DEFAULT>;
        zephyr,resolution = <12>;
    };
    /* Repeat the above 'Channel' code block to use more channels. Only change reg
= <CHANNEL_NUMBER>; */

};
```

The proj.conf file

```
CONFIG_GPIO=y
CONFIG_ADC=y
CONFIG_DAC=y
```

The main.c program

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

// Create our ADC device
static const struct device *const adc_dev1 = DEVICE_DT_GET(DT_NODELABEL(adc1));

// Create variable adc_ch1_cfg with the configuration parameters from Overlay
static const struct adc_channel_cfg adc_ch1_cfg = ADC_CHANNEL_CFG_DT( DT_CHILD( DT_NODELABEL(adc1),
channel_5) );

// Create our DAC device
static const struct device *const dac_dev1 = DEVICE_DT_GET( DT_NODELABEL(dac1));
static const struct dac_channel_cfg dac_ch1_cfg = {
        .channel_id  = 1,
        .resolution  = 12,
        .buffered = 1
};

int main() {

    // Initialise the ADC and DAC
    adc_channel_setup(adc_dev1, &adc_ch1_cfg);
    dac_channel_setup(dac_dev1, &dac_ch1_cfg);
```

```
    // Create a measurement sequence. Can measure from many channels in one go
    uint16_t buf;
    struct adc_sequence sequence = {
        .channels = BIT(5), //Can be BIT(5) | BIT(6) to read from Channel 5 and 6
        .buffer = &buf,
        .buffer_size = sizeof(buf),
        .resolution  = 12
    };

    while(1){
        adc_read(adc_dev1, &sequence);          // Read from ADC
        dac_write_value(dac_dev1, 1, buf);      // Then Write to DAC

        int val_mv = buf * 3300/4095;           // Convert to mV by multiplying with Vref/(2^N-1)
        printk("Voltage = %d mV \n", val_mv);   // Display the current Voltage for us
        k_sleep(K_MSEC(100));
    }
}
```

Confirm that you can vary the intensity of the LED using the potentiometer.

Program 7: Illuminating LEDs depending on input voltage

Use your knowledge from previous milestones on creating custom LED GPIO devices, and write a program that will illuminate 4 LEDs depending on the value of input voltage that is produced using a potentiometer as was done in figure 6 and code above. The LED illumination in response to the input voltage should meet the requirements in the Table below. Use **If** and **elseif** and **else** statements.

Voltage	LED1	LED2	LED3	LED4
0 V< Vin <0.6 V	0	0	0	0
0.6 V< Vin <1.2 V	1	0	0	0
1.2 V< Vin <1.8 V	1	1	0	0
1.8 V< Vin <2.4 V	1	1	1	0
Vin >2.4V	1	1	1	1

Using a multimeter on the analogue input, confirm that the appropriate LEDs illuminate for the given input voltages

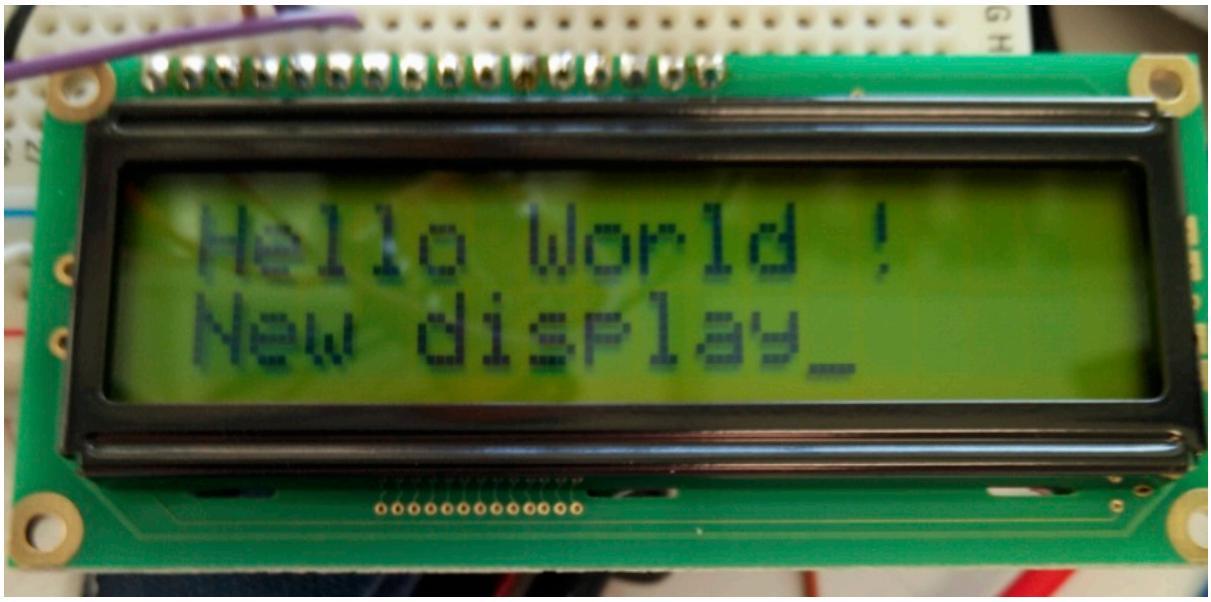
Learning Outcomes:

- Learn how to properly connect a variable resistor (potentiometer) on their MCU.
- Be able to explain how a variable resistor works.
- Learn to use the Analog Input functionality of MCU with Zephyr.
- Use the Analog Input to enable several GPIO Outputs.

Milestone 7: Show your code to a demonstrator and confirm that the LED pattern is correct for the input voltages.

Write characters to an LCD display

In order to have a meaningful relationship with your MCU, it needs to be able to speak to you. One such method is an alphanumeric LCD display. As shown below.



These are low cost, small and use little power. They are therefore ideal for outputting information to a user, such as yourself.

In order to use such a device, we need to carry out several steps, otherwise we would get bogged down in instruction manuals and coding.

1) Wire the LCD to your Nucleo board
2) Make a new program to write something to the display
3) Import the code library for this display from the mbed website, and add it to you program

- LCD Pin 1 is the ground – connect this to your ground.
- LCD Pin 2 is the +ve supply, connect this to mbed pin 5V (+5V USB supply output).
- LCD pin 4 is register select, connect to MCU pin PA6 (=A5)
- LCD pin 5 is read/write, connect to GND since we will only be writing.
- LCD pin 6 is enable, connect to MCU pin PA7 (=A6)
- LCD pins 11, 12, 13, 14 are the MSB end of the data but, connect these to the MCU pins PA8, PA11, PB5 and PB4 (D9, D10, D11, D12) respectively.
- LCD pin 3 is the contrast control, and requires a voltage slightly above ground. You can construct a voltage divider.
  Wire a potentiometer (variable resistor) to the +V and 0V supplies, and to LCD input Pin 3. With the potentiometer, you will essentially create a variable voltage divider and you can control continuously the LCD contrast level.

Overlay file in: boards/nucleo_l432kc.overlay

```
/ {
    lcd_screen: hd44780 {
        compatible = "hit,hd44780";

        mode = <4>;                                             // Operating in 4-bit mode.
        register-select-gpios = <&gpioa 6 GPIO_ACTIVE_HIGH>;     // LCD Register Select (RS)
GPIO PA6 to pin4
        enable-gpios = <&gpioa 7 GPIO_ACTIVE_HIGH>;  // Enable to send commands GPIO PA7 to pin6
        data-bus-gpios =    <0>,                     / An array of 8 GPIO pins. If this
                            <0>,                    // operates on 4-bit mode we only
                            <0>,                   // need to link the last 4 to GPIOs pins
                            <0>,
                            <&gpioa 8 GPIO_ACTIVE_HIGH>,        // GPIO PA8 to LCD pin11
                            <&gpioa 11 GPIO_ACTIVE_HIGH>,        // GPIO PA11 to LCD pin12
                            <&gpiob 5 GPIO_ACTIVE_HIGH>,        // GPIO PB5 to LCD pin13
                            <&gpiob 4 GPIO_ACTIVE_HIGH>;        // GPIO PB4 to LCD pin14
        columns = <16>;     // LCD number of columns
        rows = <2>;         // LCD number of rows
    };
};
```

The proj.conf file we add

```
CONFIG_AUXDISPLAY=y
```

The main.c file

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


// Create our LCD device
static const struct device *const lcd_display = DEVICE_DT_GET( DT_NODELABEL(lcd_screen) );

int main() {


    /* Clear display */
    auxdisplay_clear(lcd_display);

    const char *msg1 = "Some text here: Line 1";
    const char *msg2 = "Some text here: Line 2";
    while(1){
        /* Clear the LCD display */
        auxdisplay_clear(lcd_display);

        /* First line */
        auxdisplay_cursor_position_set(lcd_display, AUXDISPLAY_POSITION_ABSOLUTE, 0,0);
        auxdisplay_write(lcd_display, msg1, strlen(msg1));
```

```
        /* Second line */
        auxdisplay_cursor_position_set(lcd_display, AUXDISPLAY_POSITION_ABSOLUTE, 0,1);
        auxdisplay_write(lcd_display, msg2, strlen(msg2));

        k_sleep(K_SECONDS(1));          // Sleep for 1 second
    }
}
```

There are 3 commands that we use to control the display:

1. `auxdisplay_clear(lcd_display);` is to clear the display
2. `auxdisplay_cursor_position_set(lcd_display, AUXDISPLAY_POSITION_ABSOLUTE, 0,1);` is to position the writing cursor to the location we want. The last 2 numbers in this function indicate the x-location and y-location respectively.
3. `auxdisplay_write(lcd_display, msg2, strlen(msg2));` is to write the message on the LCD. The function takes as inputs the device variable (lcd_display), the message we want to write (msg2) and its length (strlen(msg2))

Program 8: Display a voltage to an LCD display

We will use the potentiometer shown in Fig. 6 (left) for this program. By combining what we learned in the previous Program 7, we would like to display on an LCD the Analogue Input voltage, defined by the potentiometer (variable resistor).

Note that the `auxdisplay_write()` function, takes an input that is a character, such as the `const char *msg1 = "Text";` from the program in the previous page. **We need to convert the voltage given in mV into char**. To do this we will use the internal C++ function: `snprintf()`. See below a snapshot of the code you can use:

```
char msg1[64];  // Create a char variable to store the information
while(1){
    /* Clear the LCD display */
    auxdisplay_clear(lcd_display);

    adc_read(adc_dev1, &sequence);  // Read from ADC
    int voltage = buf * 3300/4095;  // Convert Voltage to mV

    snprintf(msg1, sizeof(msg1), "Voltage is: %d mV", voltage);
    auxdisplay_write(lcd_display, msg1, strlen(msg1) ); // Write to LCD
    k_sleep(K_MSEC(200));           // Sleep for 200ms
}
```

Do NOT forget to create the ADC device and add the ADC in the overlay file. In addition DO NOT forget to add the ADC in the prj.conf file.

Learning Outcomes:

- Learn how to properly connect an LCD display on their MCU.
- Use the appropriate Zephyr Drivers `compatible = "hit,hd44780"` to define an LCD device and link that to MCU pins.
- Connect correctly the potentiometer and use the ADC
- Display messages on the LCD display and display the ADC voltage 0 – 3.3V.

Milestone 8: Show to you demonstrator that the voltage is correctly displayed on your LCD.

Pulse Width Modulation (PWM)

Producing analogue output signals is an important capability of a microcontroller, but sometimes it is useful to stay in the digital domain when outputting control signals. Pulse width modulation (PWM) offers this capability and its importance is demonstrated by the fact that the Nucleo board has several PWM output channels, in stark contrast to only two analogue outputs. The PWM outputs are usually directly linked to an MCU Timer. In Nucleo there are 2 such timers we can use for PWMs, TIM1 and TIM2 and each timer is multiplexed to MCU pins according to Table 15 of the specification sheet:
https://www.st.com/resource/en/datasheet/stm32l432kc.pdf

For our examples, we can use TIM2 linked to MCU pins as follows:

- TIM2_CH1 to pin PA0
- TIM2_CH2 to pin PA1
- TIM2_CH3 to pin PA2
- TIM2_CH4 to pin PA3
- TIM2_CH1 to pin PA5
- TIM2_CH1 to pin PA15

Pulse Width Modulation is a clever way to get a rectangular waveform to control an analogue variable. In PWM, the frequency of the signal is usually kept constant, but the pulse width, or "ON" time is varied – hence the name. A PWM signal is shown in Figure 7 below.

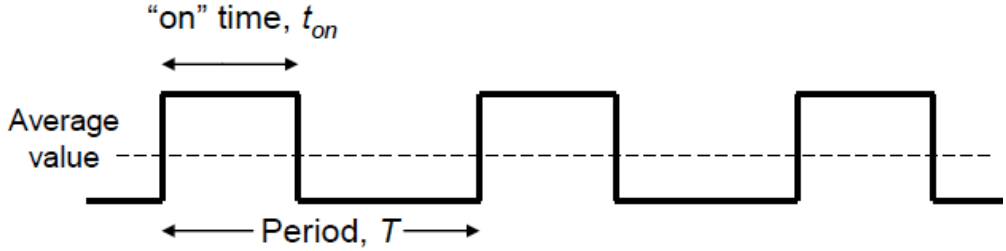


Figure 7 – A PWM waveform

The duty cycle is the proportion of time that the pulse is "on" and is expressed as a percentage,

$$\text{duty cycle} = \frac{\text{pulse on time}}{\text{pulse period}} x100\%$$

A 100% duty cycle means the signal is always ON. 0% duty cycle means the signal is always OFF. 50% duty cycle means that in a given period, the signal is ON for half the time and OFF for half the time – i.e. a square wave.

A PWM signal has an "average" value as shown in Figure 7, depending on the duty cycle. By controlling the duty cycle, the average value can be controlled. Key PWM parameters are the **pulse period** and **pulse width**.

The MCU pins are pre-configured to operate as GPIOs. If we want use an MCU pin as a PWM (i.e. a timer) we need to somehow disconnect the GPIO functionality and connect a PWM

functionality. One of the greatest advantages of modern MCUs is that pins can be re-configured to perform different functions. This can be easily done in Zephyr through the **Overlay** file.

Overlay file

```
/{

    pwmleds: pwmleds {
        compatible = "pwm-leds";
        status = "okay";

        green_pwm_led: green_pwm_led {
            pwms = <&pwm2 4 PWM_MSEC(20) PWM_POLARITY_NORMAL>; // &pwm2 links to
TIM1_CH4 and PA3
        };
    };

};

&pwm2 {
    status = "okay";
    pinctrl-0 = <&pwm2_custom>;
};


&pinctrl {
    pwm2_custom: pwm2_custom {
        pinmux = <STM32_PINMUX('A', 3, AF1)>; // Alternative function AF1 for this pin
    };
};
```

There are a few things we do in the Overlay file. First you may notice there is the "standard" **main node /{ … };** at the beginning and some "system" nodes `&pwm2 { … };` and `&pinctrl { … };`. The system nodes are pre-defined by STM however we can add some modifications. The user's Overlay file will always have priority.

A. The `&pwm2 { … };` node:
In this node we Enable the PWM2 peripheral using `status = "okay";` and at the same time we create a custom pin control variable `pinctrl-0 = <&pwm2_custom>;`, which will help us change the functionality of a pin.

B. The `&pinctrl { … };` node:
This is a system node where we can customise the functionalities of MCU pins. The variable pwm2_custom is linked here to pin PA3 using the command `pinmux = <STM32_PINMUX('A', 3, AF1)>;` This is a command specific to STM32 microcontrollers and it will be different for other manufacturers, however the idea is the same. What we essentially say to the MCU is that the pin 3 belonging to bus 'A', i.e. PA3, will have the functionality 'AF1', which according to Table 15 of the datasheet is a TIM2_CH4.

C. The `pwm_leds: pwm_leds { … };` node in the main Overlay node:

Here we will bundle our custom PWM devices that use the `compatible = "pwm-leds";` Zephyr drivers. There are many other PWM drivers you can use, i.e. for stepper motors, but the *pwm-leds* is the simplest one.

In the same way we defined LEDs at the beginning, we can define a custom device linked to a PWM pin and thus a TIM channel. Here we created a device named: `green_pwm_led` that has properties `pwms`. The property connects the `&pwm2` peripheral to channel `4` and initialises a PWM period of 20ms `PWM_MSEC(20)` and normal polarity, `PWM_POLARITY_NORMAL`.

Note, the period can be changed in the **main.c** file.

Prj.conf file

```
CONFIG_PWM=y
```

Main.c file

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


// Create our PWM device
static const struct pwm_dt_spec pwm_led0 = PWM_DT_SPEC_GET(DT_NODELABEL(green_pwm_led));


int main(void)
{
    // Initialise the PWM - Inputs are: device name, period, duty cycle
    int period = 1000000;     // Period in nanoseconds
    int pulse_width = period/2; // Pulse width in nanoseconds

    pwm_set_dt(&pwm_led0, period, pulse_width); // Set PWM and forget

    while (1) {
        // Do other things
    }
}
```

As you can see from the above, we can set the period and pulse width of a PWM during the program runtime. In this example, we set the PWM at the beginning and we forget about it. The While loop will do other things.

If you run the program above and look at the output form pin PA3 on an oscilloscope, you should see a square wave. What is the frequency?

Program 9: Create a PWM signal with period and duty cycle controlled by switches

Based on the above, and previous programs, use 2 switches to control both the period and duty cycle of a PWM signal.
One switch should control the Period between 1KHz and 1MHz and the other switch should control the duty cycle with values of 20% and 80 %. Use appropriate pull-down or pull-up resistors with your switches.

The PWM output should be observed on an oscilloscope.

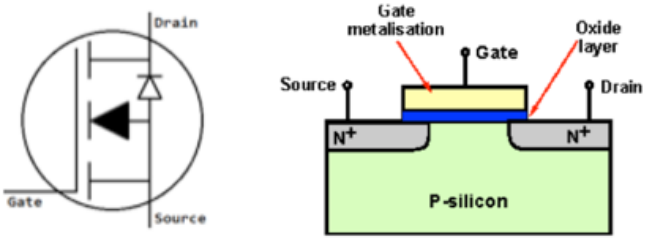
Learning Outcomes:

- Learn how to use the PWM functionality of the MCU.
- Learn how to modify MCU pin functionalities through Zephyr, using the MCU datasheet.
- Combine several MCU functionalities at the same time.
- Observe a signal on an oscilloscope.

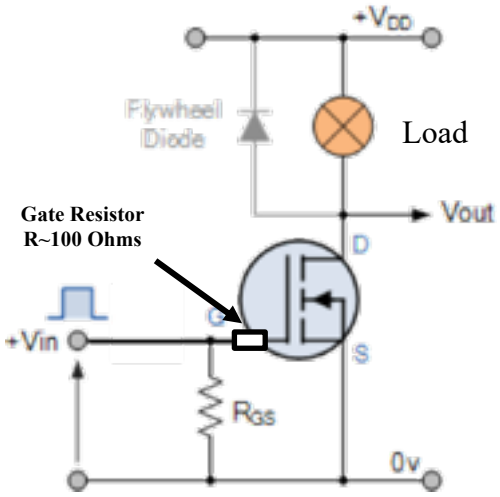
Milestone 9: Show your code and the resulting waveforms on an oscilloscope to a demonstrator.

Program 10: Control a motor or light bulb using the PWM output

In this exercise, the power consumption of the application exceeds that of the microcontroller output. In order to drive greater loads or higher voltages, a field effect transistor (FET) can be used. So long as we use a high enough voltage on the transistor's "gate" pin, it can be considered simply as a digital switch. The PWM signal to the gate pin of the FET turns it on and off rapidly. Due to the design of the FET, it is the field from the gate that causes conduction in the drain-source channel. A field is created simply from a voltage (like in a capacitor), and so long as the capacitance is small, very little current is required to turn on the FET.



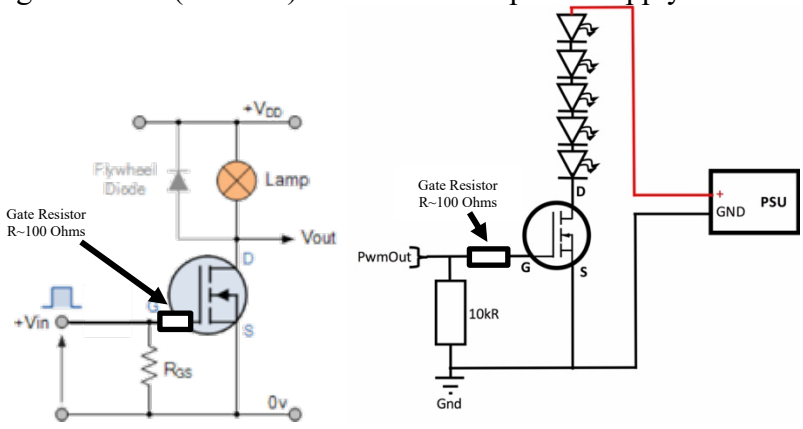
A schematic of a n-channel FET with a physical cut through showing the p channel and the n source and drain contacts. The gate contact is effectively insulated from the rest of the semiconductor so does not require much current to function.



The FET gate should be connected to the PWM output. A **pull down** resistor (10k-100k Ohm or so) can be added as good practice so the load isn't driven when the MCU is off, as well as a **Gate Resistor** (R~100-220 Ohms) to limit the current your transistor can draw from the MCU. For an inductive load (such as a motor or transformer), a **flywheel diode** must be added to protect the FET from back-emf.
The positive supply for the load can now be considerably more than 3.3V, and the load can be greater than the ~50mW that the MCU can drive directly.

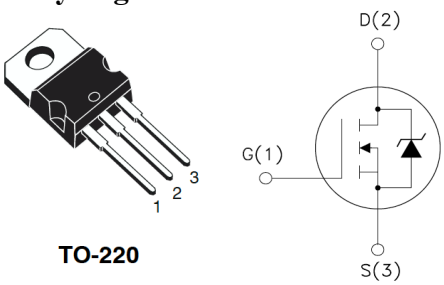
The LED strips or a Motor are good examples of typical loads you might want to drive.

LED strips run at 12V and 400mA so you need the external power supply to drive them.
Any more than about 20mA and the MCU cannot cope.
So, we need a digital switch (the FET) and an external power supply…



Use a flyback diode if the load is inductive, i.e. it has a coil in it (motor, speaker etc.)
LEDs are not inductive, so we do not need the diode protection here.

It does matter which way round you get the transistor !!!!!



The P16NF06L FET. N-CHANNEL 60V, 0.07 Ω, 16A

So we have,

- Gate (G): this is the part that goes to your delicate digital logic (PWM).
- Source (S): goes to your ground.
- Drain (D): goes to one side of your load.

Your power supply +ve goes ONLY to the other side of your load.
Ground is common to everything PSU, digital MCU and Source.

<u>YOU MUST KEEP THE +ve 12V away from the delicate digital logic. It will fry your MCU.</u>

Get a demonstrator to check your circuit before turning on the PSU.

Create **a new** program to change the PWM duty cycle to dim the lightbulb, or vary the motor's speed. You should choose an appropriate period to be set once in your program and now have the duty varying from 0 to 100%, using a variable resistor and an ADC.

Note the input from an ADC is always a value from 0 to $(2^N - 1)$. To achieve pulse width with duty cycles from 0 to 100% :

```
int pulse_width = period * buf/4095;
```

where the ADC value 'buf' is normalised to $(2^N - 1)$ and then multiplied by the period.

Learning Outcomes:

- Learn when we use transistors in conjunction with an MCU output.
- Learn what passive components should **always** be connected together with a transistor.
- Learn to use an external voltage supply source.
- Drive and control an external load.

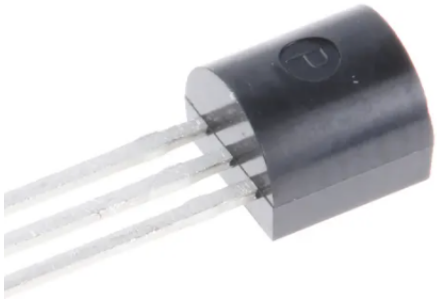
Milestone 10: Show your circuit and bulb/motor control to a demonstrator.

Program 11: Create a cooling fan

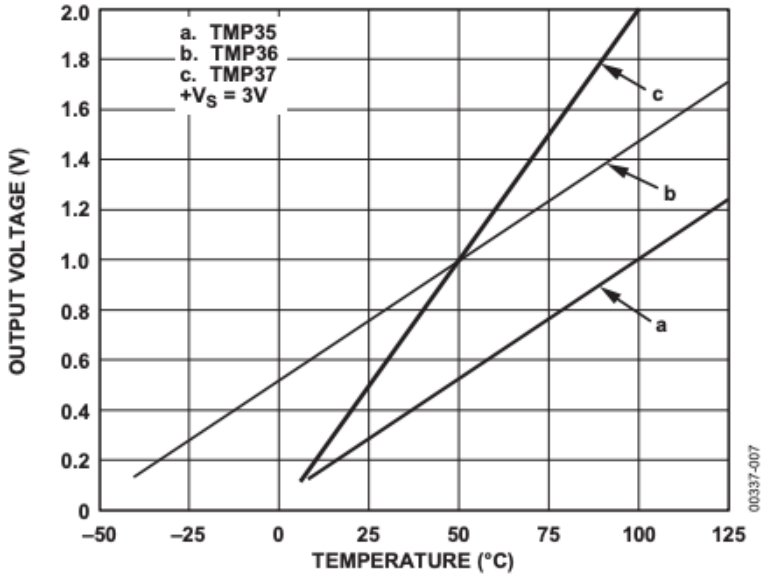
Every element that creates a lot of heat is equipped with a temperature sensor, such as a car engine or a computer CPU. Overheating these elements can have catastrophic consequences for your car or computer. Therefore, we would like to create a cooling fan that turns on when the temperature sensor received a temperature that is too high.

In this task you will use your LCD display to output the temperature to the user, a motor that will be your fan, a switch that will be used by the user to override the system and a temperature sensor (TMP37FT9Z).

Temperature Sensor:



PIN 1, $+V_S$; PIN 2, V_{OUT}; PIN 3, GND



The TMP37FT9Z is intended for applications over the range of 5°C to 100°C and provides an output scale factor of 20 mV/°C. As you can see from the manufacturer's plot above, the TMP37FT9Z provides a 500 mV output at 25°C.

Implementation:

1. Connect Pin-1 to VDD (3.3V) and Pin-3 to GND. Pin-2 should be connected to one of your MCU's Analogue-to-Digital Converters (ADC), i.e. PA0.
2. Create an appropriate algorithm to convert the ADC input to temperature, using the figure above.
3. Wire appropriately your LCD display and output the measured temperature on the screen.
4. Use a PWM output and control the duty-cycle based on the temperature. We would like for any temperatures below 35°C that the duty-cycle is 0%, i.e. the pulse width is 0. For temperatures >35°C and up to 100°C the duty cycle should change linearly from 0% to 100%. Use appropriate frequency/period for the PWM (see your lecture notes).
5. Implement a motor (see Milestone 10). You will need a power supply, a transistor, a diode, a gate resistor and pull-down resistor. Do not turn on the power supply unless your circuit is checked by your tutor.
6. Use the soldering iron to induce heat on your temperature sensor. Adjust the soldering iron temperature to 90°C.

GENTLY TOUTCH THE SOLDERING IRON ON THE SENSOR

6. Check on the LCD that the temperature is actually increasing and that once we exceed 35°C your motor starts to work.
7. **(Advanced task)** Now implement a pull-down switch together with an **interrupt**, so that the user can override the system. When the MCU receives logic 1 from the pull-down switch, the motor should start working on a 50% duty cycle and for 5 seconds, no matter what the temperature is. After the 5 seconds elapse, the fan should switch off and control of the fan should return to the temperature sensor.

Learning Outcomes:

- Learn to use a specification sheet of an analogue sensor
- Combine together elements from previous milestones, such as LCD, motor, transistor, ADC.
- Drive and control an external load (motor) using conditional statements.
- Use the Interrupt functionality of your MCU through Zephyr.

Milestone 11: Show your circuit to a demonstrator and demonstrate that the motor turns ON when the temperature is >35 degrees. Display the Interrupt functionality.

Note: Do not dismantle your program. You will need the same components for the next exercise.

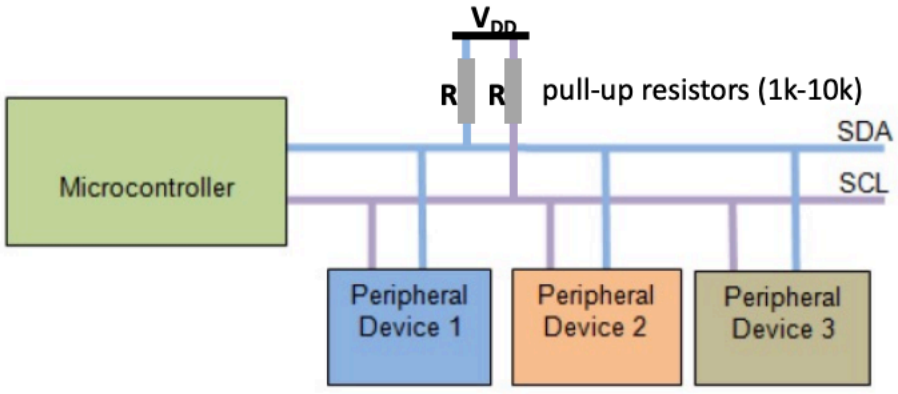
Serial Communications: The I2C

An efficient way of communicating with many devices, is through serial communications. Your MCU has multiple options of serial communications, such as the SPI, I2C, USART etc. In the following milestone we will explore a temperature sensor that is based on the **Inter-Integrated Circuit Bus (I2C)**.

The I2C bus is a synchronous communication protocol that uses only 2 wires for the transmission and reception of data. The two wires are named:

- SDA – serial data
- SCL – serial clock

In this type of communication protocol the SDA and SCL wires **have always a high-state (logic 1)** and it is up to the MCU or the external device to pull the voltage to a low-state (logic 0). To achieve a permanent high state on these wires, we connect them to the V_{DD} supply through a high value **pull-up resistor**, as shown below,



On your Nucleo MCU there are two available I2C ports you can use, **I2C1** and **I2C3**. For example, you can look for the pins that are marked as I2Cx_SCL and I2Cx_SDA, where x is the number of the port. The "_SCL" and "_SDA" denote the serial clock and serial data connections.

1. Pins PA9 & PA10 (D0,D1) (port 1)
2. Pins PB6 & PB7 (D4,D5) (copy of port 1)
3. Pins PB4 PA7 (D12, A6) (port 3)

How do we communicate with an I2C device?

1. The first thing we have to do is to enable our desired I2C bus in the Zephyr Overlay file. In this example, we will use the I2C1 bus, thus we will enable the I2C1 node.
2. Then inside the I2C1 node, we will add a device and define its unique address.
3. Lastly we will link that device to the appropriate MCU pins using the pinctrl{ }; node, similar to what we did for the PWMs.

Overlay

```
/ {
    // This is the main Overlay node. I am introducing you to a powerful node, where you can
create convenient names for devices, also known as aliases.
    aliases {
        mytemp = &myTempSensor;
    };
};

&i2c1 {
    status = "okay";
    clock-frequency = <I2C_BITRATE_STANDARD>;
    pinctrl-0 = <&i2c1_scl_pa9 &i2c1_sda_pa10>;

    myTempSensor: myTempSensor@4A {     // TC74-A2
        compatible = "i2c-device";
        reg = < 0x4A >;     //  For TC74A0 the internal address is 1001010 = 0x4A
    };
};

/* Define the alternate pins for I2C1 */
&pinctrl {
    i2c1_scl_pa9: i2c1_scl_pa9 {
        pinmux = <STM32_PINMUX('A', 9, AF4)>;  /* PA9 AF4 = I2C1_SCL */
        drive-open-drain;
        bias-pull-up;
    };

    i2c1_sda_pa10: i2c1_sda_pa10 {
        pinmux = <STM32_PINMUX('A', 10, AF4)>; /* PA10 AF4 = I2C1_SDA */
        drive-open-drain;
        bias-pull-up;
    };
};
```

Let's explain step-by-step what we do in the Overlay file.

1. The `&i2c1 { };` node: This is a system node that we would modify.
    a. `status = "okay";` This enables the I2C1 bus
    b. `pinctrl-0 = <&i2c1_scl_pa9 &i2c1_sda_pa10>;` We use this line to help us define variables for the MCU pins that will be used for Clock (SCL) and Data (SDA) signals. These variables are &i2c1_scl_pa9 and &i2c1_scl_pa10. Is always useful to come up with variable names that are self-explanatory.
    c. `myTempSensor: myTempSensor@4A { };` Inside the I2C1 bus node we create our Temperature sensor device. It uses a Zephyr generic "**i2c-device**" drivers and it has a **reg = <0x4A>** property. The reg property is the unique address of the device. For the T74A2 sensor the address is 0x4A.

2. The `&pinctrl { };` node: This is the Overlay node where we assign pins to various functions. This is similar to the PWM program we saw earlier. Looking at Table 15 from the datasheet https://www.st.com/resource/en/datasheet/stm32l432kc.pdf we see that I2C1 is linked to pins PA9 for SCL and PA10 for SDA. The **pinmux** property uses AF4 as per datasheet's Table 15.
    a. `drive-open-drain;` This property forces the pin to be floating
    b. `bias-pull-up;` This property connects the pin to an internal pull-up resistor

3. The `aliases { };` node in the main Overlay node: This is a very powerful node where we can give aliases to created devices and then call them in the **main.c** using that alias name. Here `mytemp = &myTempSensor;` the device that has a variable &myTempSensor is given the alias mytemp. This will be used in the main.c

Prj.conf

```
CONFIG_I2C=y
```

main.c

```
#include <zephyr/drivers/i2c.h>

// Create the I2C sensor device
static const struct i2c_dt_spec temp_i2c = I2C_DT_SPEC_GET( DT_ALIAS(mytemp) );

int main() {

    uint8_t temp;           // Create a variable to read the temperature
    uint8_t reg = 0x00;     // This the memory location of the sensor where
temperature is kept

    while(1){

        // i2c_write_read_dt(&temp_i2c, &reg, 1, &temp, 1);
        i2c_write_dt(&temp_i2c, &reg, 1);
        i2c_read_dt(&temp_i2c, &temp, 1);

        k_sleep(K_MSEC(500));           // Sleep for 500ms
    }
}
```

Let's explain step-by-step what we do in the main.c file.

1. Creating a variable for the device: We use the command `I2C_DT_SPEC_GET` and notice we also used the **DT_ALIAS** to get the properties of the device from the Overlay file.
2. The `uint8_t temp`: This variable is created to store the temperature when we get it from the sensor.
3. The `uint8_t reg = 0x00`: This variable has a value of 0x00, which is linked to the memory/register location in our temperature sensor, where the temperature is kept by

the sensor. How do we know this? The sensor datasheet tells us exactly what each resister does and what it stores.

4. `i2c_write_dt(&temp_i2c, &reg, 1);` This sends a signal to the sensor with variable name **temp_i2c** asking it for the information stored in the **reg=0x00**
5. `i2c_read_dt(&temp_i2c, &temp, 1);` This reads the temperature send by the sensor and stores it in the **temp** variable.

**** Note: When you put together Program 12 below, remember to include the**

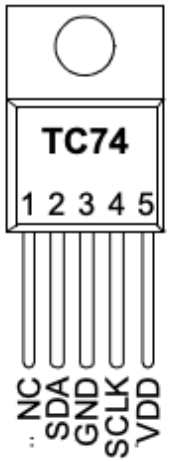
1. **LCD and PWM Overlay nodes,**
2. **LCD and PWM configuration in the Prj.conf and**
3. **LCD write commands and PWM in the main.c file (see LCD milestones) .**

Program 12: Create a cooling fan with a Serial temperature sensor

We would like to modify Program 11 and use a different temperature sensor, so please do not dismantle the LCD from the previous program. The sensor we will be using for this program is TC74. This is a **serial sensor** and communications between your MCU and the sensor have to be done through the **I2C serial port interface**. This means you will have to use 2 wires for communications, the **data wire** and the **clock wire**.

The wiring diagram of your sensor looks like the figure to the right. The pins are the following,

- NC – no connection,
- SDA – I2C serial data (bidirectional),
- GND – Ground,
- SCLK – I2C input serial clock,
- VDD – Power Supply 3.3V



The temperature resolution of this sensor is 1°C.

The TC74A2 is internally programmed to have a default I2C address value of **1001010 = 0x4A**, or if you have the TC74A0 the internal address is **1001000 = 0x48.** This value goes in the Overlay file in the **reg** property.

Serial Port Communication:
Use your knowledge from the I2C lecture and the previous lab pages, and create a serial communication. Remember in the I2C communication protocol the CLK (clock) and SDA (data) buses **must** be connected with a pull-up resistor to the V_{DD}.

The command we will 'write' (send) to the sensor to let it know that we would like to read its temperature is: **0x00** (see information in previous page).

For more information on these commands, I would advise you to have a quick look at the manufacturer's manual (link), Table 4-1.

To display the values of the sensor on your LCD you can use,

```
char msg1[64];
auxdisplay_clear(lcd_display);
snprintf(msg1, sizeof(msg1), "Temp is: %d oC", temp);
auxdisplay_write(lcd_display, msg1, strlen(msg1) );
```

This displays the temperature value on your LCD as a float number.

Learning Outcomes:

- Learn the Serial Communication Protocol of your microcontroller (I2C)
- Appropriately use the Clock and Data buses of your microcontroller, using the correct resistors.
- Link the MCU pins to the I2C bus through the Zephyr's Overlay file
- Communicate with the I2C sensor through read/write commands.
- Convert the information you receive from the sensor into meaningful information.

Milestone 12: Show your circuit and your program to a demonstrator and demonstrate that the motor turns ON when the temperature is >35 degrees.

Program 12: Timers - Parking sensor

Most modern cars are equipped with parking sensors, to aid the driver when parking the car. These sensors are easy to identify on a car, they are small disks positioned on the front and rear bumper of a car. These are in their majority 'ultrasonic' sensors, which they work in the following manner:

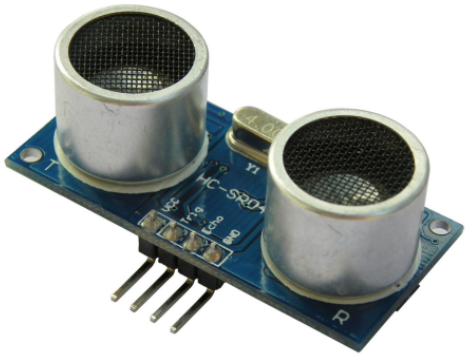
- The transmitter of the sensor sends an ultrasonic signal that travels towards an obstacle.
- The obstacle will reflect part of that ultrasonic signal
- Part of the reflected signal will arrive back at the receiver of the sensor

Since we know the travel speed of the ultrasonic signal (speed of sound 340m/s) and we can measure the time between sending and receiving the ultrasonic signal, then we can calculate the distance of our sensor from the obstacle. This is done by,

$$2x = u\,t$$

where x is the distance travelled by the signal, u is the speed of sound (340m/s) and t is the total time it took for the signal to arrive back at our sensor.

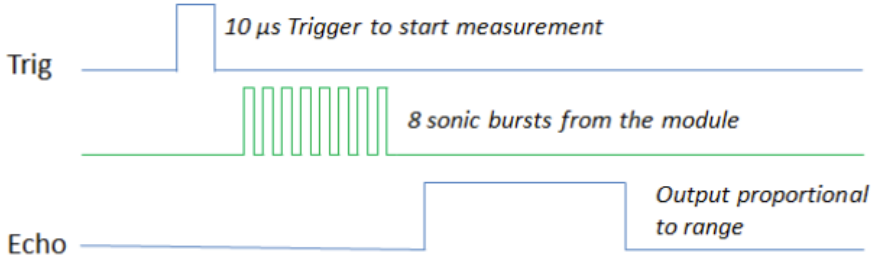
The HC-SR04 sensor



The sensor has 4 pins. The power supply, V_{DD} (5V), the ground pin (GND) and the Trigger and Echo pins.

To perform a measurement:

- We send a logic-1 trigger signal to the Trig pin for at least 10 μs.
- The sensor once it received this signal, will transmit a burst of 8 ultrasonic pulses and wait for any reflected signal.
- When the sensor receives the reflected signal it will create an echo signal on the Echo pin. The time duration of the echo signal is equal to the travel time of the ultrasonic pulse.
- The distance is calculated using the formula above.



Create a function that returns the distance

The algorithm below will help you measure the distance. Use your knowledge from the Timers lecture to create timers and time events. Use DigitalOut and DigitalIn signals for the Trigger and Echo pulses respectively.

```
float distance(){
        Create timer;
        Reset timer;
        Send a 10 μs trigger pulse to the sensor;
        Sample continuously the Echo pin until it goes high (logic-1);
        Start a timer;
        Sample continuously the Echo pin until it goes low (logic-0);
        Stop timer;
        Read the value in microseconds;
        If the timer value exceeds the maximum range delay (this is equivalent to 4m)
                Return -1.0;
        else
                Return the distance in centimetres;
}
```

Create a Parking Sensor

Use the distance you will calculate from the function above to warn the driver when is parking the car.

Use two methods for warning the driver,

1. Display the distance on your LCD screen in cm.
2. Use a buzzer (speaker) together with a PWM output to warn the driver through sound. The buzzer must start at a distance of 50cm and as the distance becomes smaller the buzzer frequency gets higher. At 5cm we should hear a continuous tone. Remember with the buzzer we need to use a transistor, a diode and a pulldown resistor and a gate resistor. You don't need an external power supply for the buzzer. Use the 3.3V.

Learning Outcomes:

- Understand how to use Timers and Events from your microcontroller.
- Create functions within your program.
- Convert time to distance.
- Use a buzzer and a PWM to control the sound tones.

Milestone 13: Show your circuit and your program to a demonstrator and demonstrate that you can measure distance and inform the driver via sound.