Embedded Programming with mbed-os

A Practical Course for Electronic Engineers

Dr. Nicholas Outram, University of Plymouth

1st Edition

Analogue I/O

Interfacing to the analogue world with mbed-os

In this section we look at interfacing with the analogue world using mbed. You will use a simple **potentiometer** to generate analogue voltages, and measure that voltage using an Analogue to Digital Converter (ADC) to perform **sampling**. Analogue signals suffer problems with noise and uncertainty. Analogue filters are used to prevent a phenomena known as aliasing when sampling a signal. For simple threshold detection, **hysteresis** is introduced to manage signal noise and avoid output jitter.

Photo of the Potentiometer used in this course

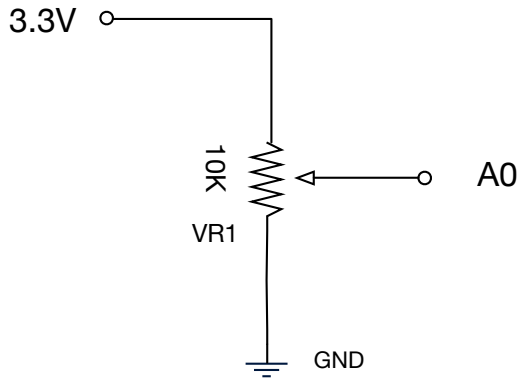
We then look at digital to analogue conversion, using both pulse-width modulation (crude, but useful for motor speed control) and a Digital-to-Analogue Converter (DAC) for more precise applications.

The mechanisms for conversion in mbed are very simple. **What is more important are the techniques for determining an accurate sampling rate and reducing aliasing noise**.For accurate sampling, we will use the `Ticker` component to generate interrupts driven by a hardware timer.

TASK 4.1.1

Wire the circuit shown. Instead of connecting the wire to A0, connect it to a DVM

By rotating the potentiometer, confirm the range of voltages to be between 0.0V and 3.3V. This is important to prevent damage later



Potentiometer Circuit providing a variable voltage input to the analogue input A0

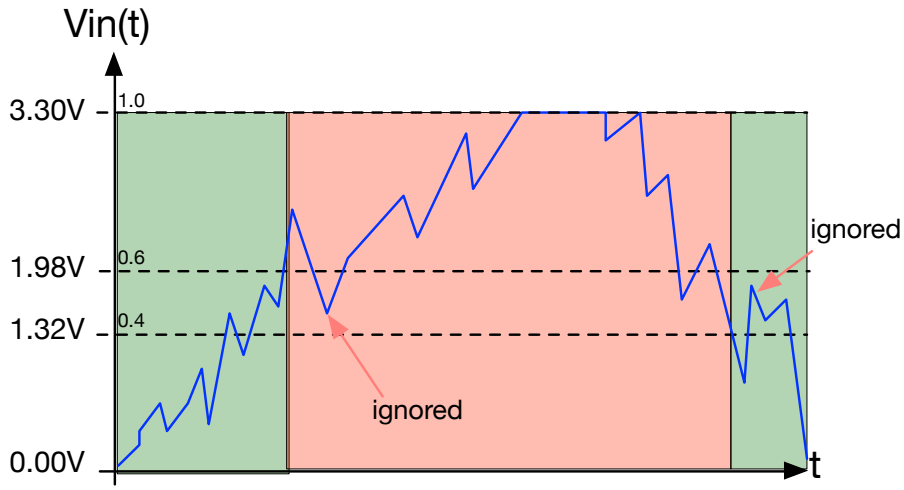
TASK 4.1.1

Connecting the centre terminal wire to A0
Import Task 411 into your mbed compiler
Run the terminal to see the output. If nothing appear, you may need to change the COM port setting (use Windows Device Manager to check)
Set the POT to the two extremes and observe the values
Set the POT such that you get close to 0.5.
Are you able to get a consistent and constant value of 0.5, and if not, **why**?
Using an if-else statement, write some additional code to switch the Green LED ON when the analogue input is between 0.0 and 0.5, the Red LED ON when greater than 0.5 and 1.0
Set the POT to halfway, such that the LED's switch on and off erratically. It can be tricky, but it's possible. **Explain the role of noise in observation.**

For a simple threshold detection task such as this, we might can use "hysteresis" to address the problem of noise.

Hysteresis

This is a technique for reducing (and sometimes effectively eliminating) the effects of noise when detecting if a signal crosses a threshold. Consider the figure below. This shows an analogue signal $V_{in}(t)$ varying with respect to time t.



Consider the example of the blue waveform in the figure. The signal is below the lower threshold at $t = 0$, so the output is GREEN. **Hysteresis** works as follows:

- In the **GREEN** (lower) state, the system will only switch to the RED state when the signal crosses above the **upper** threshold of 1.98V.

- Once in the **RED** (upper) state, the system will now only switch to the GREEN state when the signal crosses below the **lower** threshold of 1.32V.

As the signal rises above the upper threshold of 1.98V, the output state changes state to RED. Note that signal noise causes the signal to dip down below 1.98V shortly after, but that is ignored as it is still above the lower threshold.

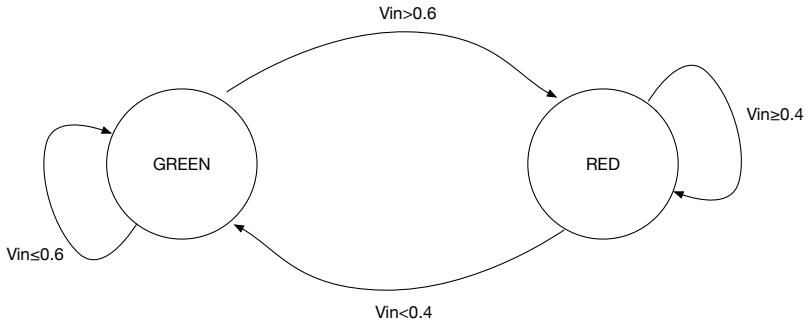
Later, the signal drops below the lower threshold causing a state change. Again due to noise, the signal rises above the lower threshold but is once again ignored as it does not exceed the upper threshold.

For this example, the noise margin is $(1.98V - 1.32V) = 0.66V$

A device that employs this technique in this way is known as a Schmitt Trigger https://en.wikipedia.org/wiki/Schmitt_trigger

TASK 4.1.3

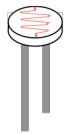
Import and study the code for Task413 uses hysteresis to avoid the problem of noise for a single threshold.
Question: what is the noise margin?
What type of state machine is this and **why**?
Draw the equivalent ASM chart in your logbook



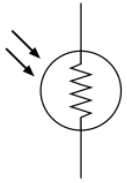
Finite state digram to model the hysteresis technique for reducing the effects of noise

Light Dependant Resistor

In this task we are going to use a new passive component, the **light dependent resistor (LDR**).

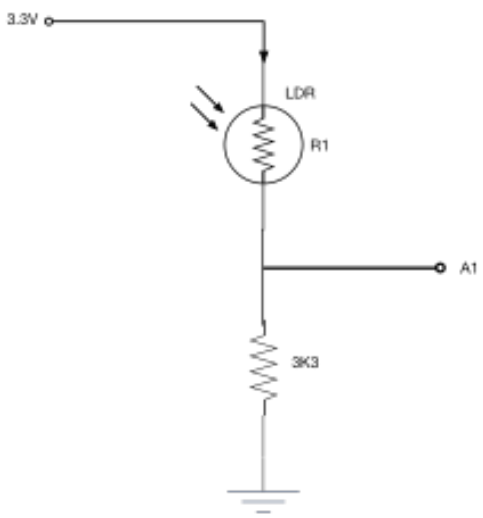


The light dependent resistor (LDR), or photo-resistor, is a variable resistor that depends on incident light (as opposed to the position of a shaft in a potentiometer)[6].



The resistance decreases as light is increased. In the dark, s resistance in the order of $M\Omega$ can be achieved. In very bright light, a resistance in the order of 100s of Ω are typical.

Used in a potential divider circuit, a simple light-to-voltage can be created, as shown:



Light dependent resistor input.
Connected to the analogue input A1

6 The resistance is also dependent on temperature, so these devices are not ideally suited to accurate measurement or instrumentation.

https://en.wikipedia.org/wiki/Photoresistor

TASK 4.2.1

Add the additional circuit to your existing design

Open the project Task421, build and run the sample code (also shown opposite)

Run PuTTY, and observe the outputs of this code in the terminal screen. Try rotating the potentiometer and blocking light from the LDR.

Set the potentiometer such that when you move your hand over the LDR, the LED's switch from OFF to ON

Question: is it possible to deduce the precise sampling rate in this code, and is this rate constant?

Now compare your answer to the comments on the next page. Did you spot all these points? Are lectures sufficient to capture all these details?

In practise, does it matter what the sampling rate for this application? **Justify your answer**.

In effect, you are using the POT to set the threshold for the light sensor.

```
#include "mbed.h"

#define kRED    (1 << 2)    //4
#define kYELLOW (1 << 1)    //2
#define kGREEN  (1 << 0)    //1
#define kALL (kRED | kYELLOW | kGREEN)

//Global objects
BusOut binaryOutput(D5, D6, D7);
DigitalIn SW1(D3);
DigitalIn SW2(D4);

AnalogIn POT_ADC_In(A0);
AnalogIn LDD_ADC_In(A1);
float fPOT, fLDR = 0.0;

//Main function
int main() {

   while(1) {

      //Read ADC
      fPOT = POT_ADC_In;
      fLDR = LDD_ADC_In;

      //Write to terminal
      printf("POT = %6.4f\tLDR = %6.4f\n", fPOT, fLDR);

      if (fLDR > fPOT) {
         binaryOutput = 0;      //Binary 000
      } else {
         binaryOutput = kALL;   //Binary 111
      }

      //Wait
      wait(0.1);

   } //end while(1)
} //end main
```

The time around the loop, and hence the sampling interval is dependant on the wait statement + time to execute the loop body.

The sampling rate is therefore not constant. The time between each sample read depends on the execution time of the code each time around the loop. This varies due:

- The conditional statements (if).
- As the code is maintained,
- With “optimisation settings” or even a change in “**compiler** version” will impact on code execution time.

To address this, we can use a timer **interrupt.**

Interrupts

A technique used by nearly every microprocessor and micro-controller to cause a branch in execution upon the occurrence of a hardware or software event.

There are two broad categories:

- **Hardware interrupt** - whereby the code is forced to branch when a hardware device (e.g. serial interface, GPIO input, timer, ADC etc..) flags that an event has occurred.
- **Software interrupt** - whereby the code is forced to branch when a software event occurs, such as a divide by zero or a trap instruction. It is still driven by hardware, but the code must be running for this to occur (and not in sleep mode).

When an interrupt occurs, a few things typically occur:

- **context save** - whereby CPU registers are preserved, typically on the current function stack
- return address is preserved
- the execution **automatically** branches to a specified function, known as an “**Interrupt Service Routine**” (ISR).
- upon completion, the return address is reinstated and the registers are restored, (as if nothing had occurred).

The ISR may well have modified some memory, or interacted with the state of the hardware. Such effects are sometimes known as “**side effects**”. The programmer has to be very careful not to cause data corruption as a consequence.

As you may already know, interrupts are commonly used to achieve a timely response to external events. For example:

- when a timer reaches its target value, a branch is forced so that a function executes at precise intervals. This is sometimes used in sampling.

- when a serial interface receives some data, an interrupt is forces the system to branch and read the data (and prevent data loss).
- if power begins to fail, an on-chip “brown-out” detection can generate an interrupt to put the system in a safe state before shutting down (assuming there is enough time to react of course).

Nested Interrupts

Some devices allow interrupts to interrupt (aka preempt) each other. They typically use a numerical priority scheme to determine which interrupts take precedence. *It can be quite challenging to design real-time systems using such as scheme.*

Power Saving

The CMOS logic used to build modern micros only consumes power when it is switching state. When a CPU has nothing to do, simply cycling inside a loop is therefore wasteful of power, as is powering unused peripherals.

Most micros can switch of peripherals that are not required (in the case of the ARM series of devices, the clocks are gated off to prevent them switching). Furthermore, most support idle modes, whereby the code execution logic is suspended in a low power state. It is only interrupts that can wake the CPU from the low power state to service a request. When all requests have been serviced, the CPU can return to an idle mode.

It is probably worth pointing out that without interrupts, the CPU cannot escape idle mode.

Interrupt Safety

Many functions, including `printf` and `scanf`, are not safe to use within interrupts. Furthermore, many of the mbed objects are not interrupt safe either. Always check the documentation before using a function or class in an interrupt.

mbed Ticker

In mbed, there is a component called a “Ticker”. This object maintains a hardware timer. When that timer reaches a certain value, an interrupt occurs, and branches to the function you specify. This function should take no arguments and return no data. For example:

```
void doISR() {
   state ^= 1;
}
```

It is probably best to give an example to explain this (opposite).

First we create an instance of the Ticker object

```
Ticker t;
```

Now we specify what function it shall call, and how often.

```
t.attach(doISR, 2);
```

The first parameter is the name of the function. In fact, this is actually the *address* of the function in program memory, but that is a detail right now. Sometimes you see it written like this:

```
t.attach(&doISR, 2);
```

where the prefix & means “address of”.

```
#include "mbed.h"
DigitalOut myled(LED1);
Ticker t;
static int state = 0;
void doISR();

int main() {
   t.attach(doISR, 2);
   myled = 0;

   while(1) {

      sleep();    //At is says

      //At this point, the ISR has run
      myled = state;

      if (state == 0) {
         printf("LED OFF\n");
      } else {
         printf("LED ON\n");
      }
   }
}

void doISR() {
   state ^= 1;  //TOGGLE
}
```

The `Ticker` does some “C++ magic” and is said to “encapsulate” a **hardware timer**, which will call the function `doISR` every 2 seconds. If you don’t understand this, don’t worry.

The interval between interrupts is specified by the second parameter (in seconds). What might surprise you is the first statement in the while loop, which is:

```
sleep();
```

This does what it says: puts the CPU into a sleeping state, sometimes known as **idle mode**. However, **the hardware timer is still running.** This separate piece of electronics inside the micro-controller genuinely runs in parallel with the CPU. *This is the most genuine form of multi-tasking.*

When 2 seconds have elapsed, the CPU is woken up and the `doISR` function is called for you (by the NVIC hardware).

All this function does is toggle a variable between 0 and 1 as follows:

```
void doISR() {
    //Toggle 0 to 1, or 1 to 0
    state ^= 1;
}
```

The code in the while loop can then resume beyond the `sleep()` function, which is to update the LED output and write to the terminal. When everything is done, the code loops and the CPU returns to the sleep mode, until the ticker wakes it again.

Question: Note that `printf` is in main, and not in the ISR. Why not just put all the code in the ISR?

Key Observation

The interrupt service routine runs once every 2 seconds. The frequency is fixed, and is independent of the code in the main function, compiler settings etc..

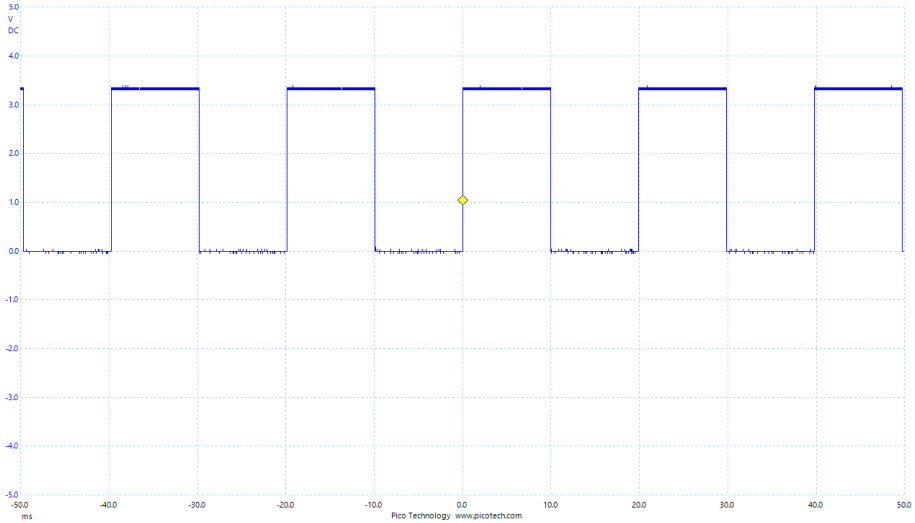
We can say that the ISR is “deterministic” (fully predictable in time).

This example is a rather special case. We have designed this system such that the CPU is always in an idle mode when the interrupt occurs. Therefore, we know where the program has got to when the interrupt occurs (sleep statement). **Please note** that this is not generally the case with interrupts, so do not extend this pattern into other applications.

Question: Why would this software pattern not be appropriate for a system with a single interrupt driven by a push switch?

TASK 4.2.1 B

Write some code to use a Ticker to toggle the pin D8
The Ticker rate should be precisely 100Hz (0.01s)
Keep the ISR as short as possible.
In main, you should simply sleep() inside a while loop
Use an Oscilloscope to verify the switching rate by monitoring pin D8



You should see a very regular and stable waveform as shown opposite.

Accurate Sampling with a Timer Interrupt

We are using the mbed framework (set of C++ objects) to perform rapid development of embedded software. We will now sample an analogue input using an on-chip **Analogue to Digital Converter** (ADC) and the `AnalogIn` component. Note the the default behaviour is to return a value a fractional value of type float, scaled between 0.0 and 1.0. In this task, we read the raw integer value.

TASK 4.4.2
Build and run the code in Task 442 (shown opposite)
Monitor the output with PuTTY (Set the rate to 115200)
Turn the POT to discover the minimum and maximum values
Convert the HEX values to decimal
How many bits resolution do you think the ADC has? (8,10, 12t or 16 bit?)
What is the smallest input voltage required to get a value of 1?

```
#include "mbed.h"

void doSample1Hz(); //Function prototype

//Global objects
AnalogIn POT_ADC_In(A0);
DigitalOut led(LED1);

//Shared variable
volatile static unsigned short sample16 = 0;

Ticker t; //The ticker, used to sample data at a fixed rate

int main()
{

   //Set up the ticker - 100Hz
   t.attach(doSample1Hz, 1.0);

   while(1) {

      //Sleep
      sleep();

      //READ ADC
      sample16 = POT_ADC_In.read_u16();

      //Displauy the sample in HEX
      printf("ADC Value: %X\n", sample16);

   } //end while(1)
} //end main

//ISR for the ticker - simply there to perform sampling
void doSample1Hz()
{
   //Toggle on board led
   led = !led;

}
```

Important Notes:

This example illustrates an important design pattern which works (only) for very simple single-interrupt problems.

- The main function is allowed to sleep.
- When the timer hits its target value, the CPU is woken and execution branches to the ISR
  - The ISR will then run the time critical code to completion - this is typically kept as short as possible.
- When the ISR exits, the main function is then allowed to resume, until it loops and sleeps again.

Note how in this specific (and simple) example, main and the ISR are time **synchronised**. This is because of sleep() and the fact that we only have one interrupt.

> We avoid race conditions because the CPU is always sleeping when the interrupt occurs
>
> Therefore, there is no danger of preempting the reading of `sample16` in main

In general, interrupt driven code can become much more complicated so we will need further strategies to avoid race conditions if our task is to scale.

Known Issues (mbed-os)

Note also this is using mbed classic (v2), and NOT the newer mbed-os v5.x. There seem to be issues with `sleep()` on the newer versions of mbed-os for this platform. This is being investigated.

Analogue Output

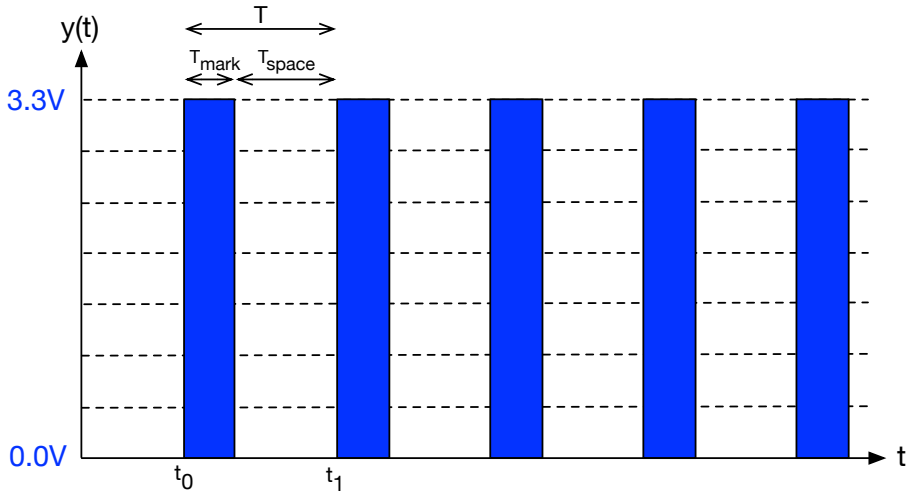
This is section, we look at ways to output analogue signals. For this we use two techniques: **P**ulse **W**idth **M**odulation (PWM) and a Digital to Analogue Converter (DAC). The FZ429ZI Microcontroller also has an on-chip DAC.

Pulse Width Modulation (PWM)

Pulse Width Modulation or PWM is a simple technique used to use an alternating voltage generate a signal with an “average DC voltage”. The technique is commonly used in control systems, including the control of DC motors.

PWM is the technique of generating a digital pulse that delivers a controllable amount of power. For example, if we wish to control a DC motor, then the amount of power delivered to the motor will dictate the motor speed. Equally, if controlling an LED, the power delivered will determine the brightness.

Consider the digital waveform below:



The digital pulse of ‘1’ is represented by the blue regions. The output is a ON for T_{mark} seconds and off for T_{space} seconds and that $T = T_{mark} + T_{space}$. If this signal is driving a device such as an LED or a DC motor, then clearly T needs to be kept small to avoid noticeable pulsing effects (although for motors, T must not be too small otherwise power is lost due to reactive properties of the motor).

The ratio of time spend ON, that is $\frac{T_{mark}}{T}$ is known as the **duty cycle**. Often this is expressed as a percentage, so

$$duty = \frac{T_{mark}}{T} \cdot 100\,\%$$

Intuition would suggest the average output voltage is somehow related to the proportion of time that the output is ON. Let's look at this more formally.

In general, for a voltage signal $v(t)$ that varies with time t, we can say that the mean voltage $\overline{V(t)}$ is derived as

$$\overline{V(t)} = \frac{1}{T}\int_{t_0}^{t_1} v(t)\ dt$$

However, from the graph, $v(t)$ is a constant over fixed durations, so we can split this up and exploit this. We say that

$$\overline{V(t)} = \frac{1}{T}\int_{t_0}^{t_0+T_{mark}} V\ dt + \int_{t_0+T_{mark}}^{t_1} 0\ dt, \text{ where } V(t) = 3.3V$$

The integral of 0 is always 0, so we can write

$$\overline{V(t)} = \frac{V}{T}\int_{t_0}^{t_0+T_{mark}} 1\ dt$$

Integrating, we get

$$\overline{V(t)} = \frac{V}{T}\left[1\right]_{t_0}^{t_0+T_{mark}}$$

$$\overline{V(t)} = \frac{V}{T}\cdot\left[t_0 + T_{mark} - t_0\right]$$

$$\overline{V(t)} = \frac{T_{mark}}{T}\cdot V$$

Intuition would also suggest the output power is somehow related to the proportion of time that the output is ON. Let's look at this more formally.

Consider driving power into a load resistance of $R\Omega$, the instantaneous power is calculated as

$$P(t) = \frac{v(t)^2}{R}$$

What is more useful is the mean power over a period of time. It is therefore the mean of $v(t)^2$ that is of interest. As V and R are constants, by the same reasoning

$$\overline{P(t)} = \frac{T_{mark}}{T}\cdot\frac{V^2}{R}$$

In summary, we can rewrite both these equations as follows:

$$mean\ voltage = \frac{T_{mark}}{T_{mark}+T_{space}}\cdot V$$

$$mean\ power = \frac{T_{mark}}{T_{mark} + T_{space}} \cdot \frac{V^2}{R}$$

Practical Application

Consider a DC motor. Put simply, we are turning the motor on and off at a fixed frequency. If $T = 1s$, then you would see the motor starting and stopping, causing a juddering. The fundamental frequency of the vibration is $f_0 = 1/T\ Hz$.

If f_0 is in the audible spectrum ($< 20kHz$), it will make the motor noisier. However, if f_0 is too high, the "reactive effects" of the motor will significantly reduce the power delivered to the motor.

As is often the case in engineering, you have to balance trade-offs between different criteria. For now, we are going to do this 'empirically'.

Pulse Width Modulation (PWM) with Timers

In this task we will implement PWM using a a simple timer (mbed Ticker).

TASK 5.2.1
Build and run Task 521 (also shown below)
What are the ON and OFF times?
Try different values of R

Note that the `wait` function is **blocking**. It does not use the CPU idle mode and simply spins, waiting for a timer to reach a specified value.

This approach does not **scale**[7] well as will be revealed in the next task 5.2.2. **Note** - do not spend too much time on the next task as it's supposed to be "awkward". It is intended to make a point!

A solution is also available.

[7] The complexity of the code does not grow proportionately with the number of LEDs

```
#include "mbed.h"

//Time period
#define T 0.001

//Mark-Space Ratio
#define R 0.1

//Mark and Space times
#define Tmark (R*T)
#define Tspace ((1.0-R)*T)

DigitalOut onboardRedLed(LED3);
DigitalOut redLED(D7);

int main() {
   printf("\nWelcome to ELEC143\n");

   while (1) {
      redLED = 0; //Space
      onboardRedLed = 0;
      wait(Tspace);
      redLED = 1;     //Mark
      onboardRedLed = 1;
      wait(Tmark);
   }
}
```

TASK 5.2.2

Using the same methods, now try and modify task 5.2.1 to also **independently** set the brightness of a green LED. **If you spend more that 10 minutes on this task, maybe stop.** There are easier ways!

If you have succeeded, consider how you might now also control the brightness of the yellow LED?

Describe what problems you encounter

You may have noticed already that this is rather cumbersome.

There are many ways to try and solve this problem. I have provided one solution that uses timer interrupts (although even this is not perfect).

What you might find is that your solution does not “scale” beyond two LEDs. One of the fundamental problems here is that the wait statement is “blocking” (you cannot do anything else unless you use an interrupt).

There are a number of elegant solutions. Timers are one possibility (as I have used), but you might want to reserve those timers for other tasks. I’ve kept the interrupt routines as short as possible to

avoid latency issues (in the event two overlap in time such that one has to wait for the other to finish).

Luckily, PWM is a common requirement and this problem was recognised a long time ago, and this microcontroller comes with PWM controllers built in! These controllers have their own timers and logic to control the output, so all run independently in parallel.

Remember: a single core microcontroller can only really execute one instruction at a time. Any attempt at multitasking is always an illusion and often comes with compromises.

Look at the code below to see how simple it is to use a PWM in mbed on this device.

```
#include "mbed.h"
DigitalOut led1(LED1);
PwmOut pwm(D6);

int main() {
   pwm.period_us(10);
   pwm = 0.5;

   while(1) {
      wait(0.5);
     led1=!led1;
    }
}
```

Task: *Use an oscilloscope to observe the signal on D6.*

This is very simple. This is because we are using the `PwmOut` object to do all the timing in parallel[8] to the CPU.

TASK 5.2.3

Using a Ticker object, make the LED slowly brighten and dim, by constantly changing the pwm period in the main loop. Remember to protect critical sections!

(you need to smoothly ramp the value of Tmark up and down between 0 and T)

We are controlling the brightness because the PWM is switching too fast for our eyes to notice. It is not true Digital to Analogue Conversion however, but a simple way to deliver variable “average power” to a load.

If you don’t have a DAC, we can get closer to true conversion by filtering the output.

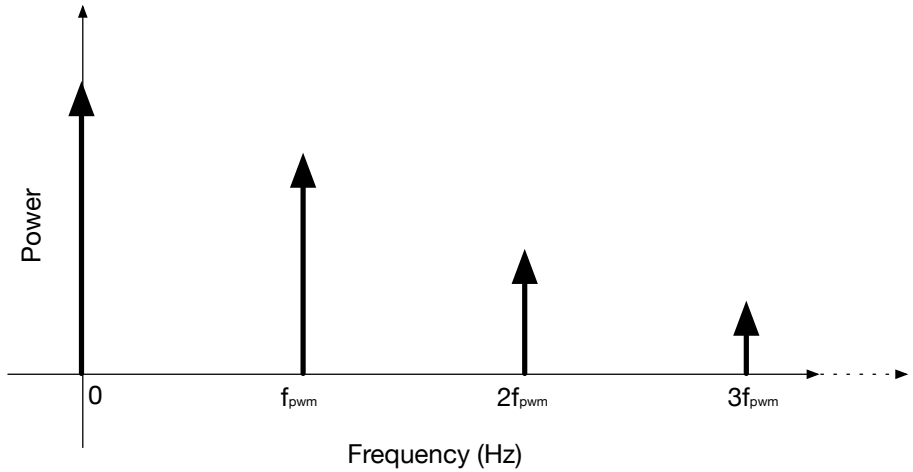
[8]The PwmOut object uses on chip PWM hardware. This run independently of the CPU. This is true concurrency.

Digital to Analogue Conversion with PWM

Consider the task of outputting a low frequency signal, for example:

- Constant voltage
- 1Hz sine wave
- Slow ramp waveform

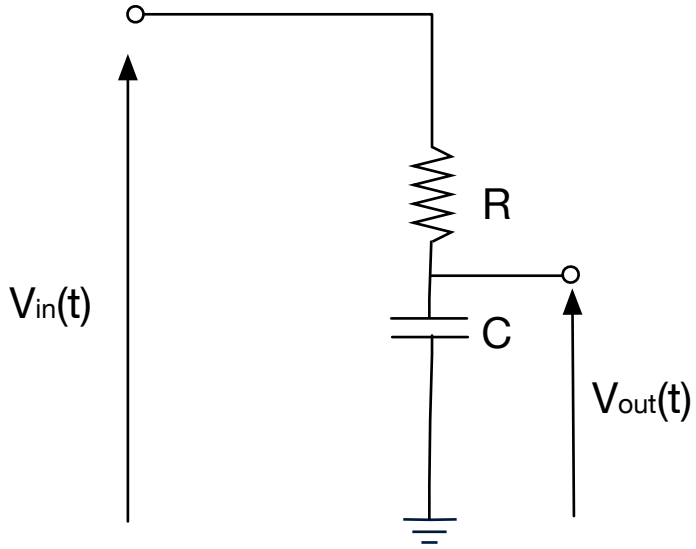
Now consider the spectrum of a stationary[9] PWM signal with period T. The repetition frequency $f_{pwm} = \frac{1}{T}$



We are only interested in the low frequency components $0 \leq f << f_{pwm}$

From the signal spectrum, as long as f_{pwm} is high enough, then a simple RC filter should be effective at removing the unwanted harmonics.

The schematic for a RC low-pass filter is given below.



You may recall the s-plane transfer function for this circuit:

$$V_{out}(s) = V_{in}(s) \cdot \frac{\frac{1}{sC}}{R + \frac{1}{sC}} = V_{in}(s) \cdot \frac{1}{sCR + 1}$$

Substitute $s = j\omega$ where $j = \sqrt{-1}$ and $\omega = 2\pi f$

[9] A stationary signal is one that has a fixed signal spectrum and statistics. In this context, over a short period of time, the PWM mark-space ratio is constant.

$$\frac{V_{out}(j\omega)}{V_{in}(j\omega)} = \frac{1}{\omega CRj + 1}$$

For the half-power point, $\left| \frac{V_{out}(j\omega)}{V_{in}(j\omega)} \right|^2 = \frac{1}{2}$

$$\left| \frac{1}{\omega CRj + 1} \right|^2 = \frac{1}{2}$$

Square-root both sides we get

$$\left| \frac{1}{\omega CRj + 1} \right| = \frac{1}{\sqrt{2}}$$

$$\sqrt{(\omega CR)^2 + 1^2} = \sqrt{2}$$

$$\therefore \omega = \frac{1}{CR}$$

Or, $f = \frac{1}{2\pi fCR}$Hz

This is known as the "-3dB point". The power (for a unit load) is defined as $10 \cdot log_{10}\left(\frac{V_{out}}{V_{in}}\right)^2$,

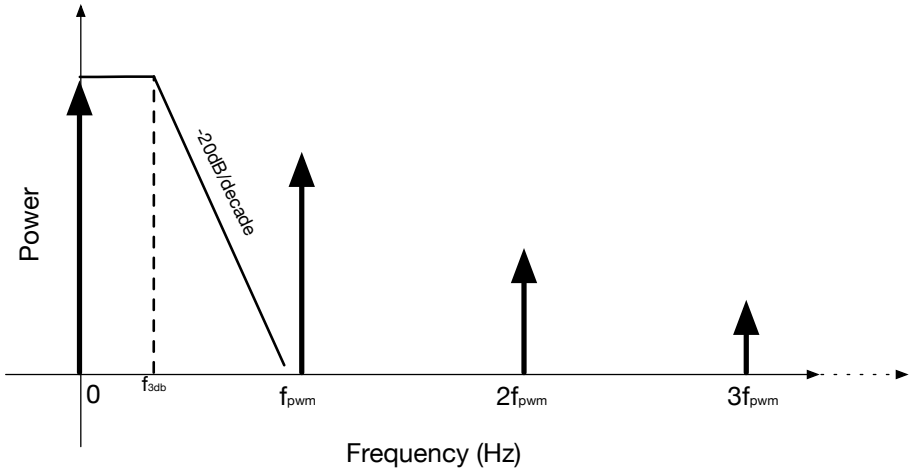
which can be written as

$$20 \cdot log_{10}\left(\frac{V_{out}}{V_{in}}\right)$$

For half RMS power, $20 \cdot log_{10}\left(\frac{1}{\sqrt{2}}\right) \approx -3.01dB$

Filter design

A single RC filter will roll off at 20dB each time the frequency increased by a factor of 10 (known as a decade).



Consider the figure above. If you choose the -3dB point to be 10Hz, then the gain will be (approximately) as follows:

Frequency (Hz)	Gain
10	-3
100	-23
1k	-43
10k	-63
100k	-83

TASK 5.2.3 - A

Using the code in Task 5.2.3, adapt the code produce a slow ramp (rise time 1s, fall time 1s)
Observe the waveform on an oscilloscope and you will observe the pulsed nature of the output
Now set the pulse repetition frequency to 100KHz
Add a simple RC filter to the output, with a -3dB point at 10Hz (at 100nF capacitor is a good choice).
Observe the output of the filter
The oscilloscope has a very high input impedance. Why is this important?

Alongside the resistor trays are 1nF, 10nF and 100nF capacitors.
You can also open the “PWM-as-DAC” project on os.mbed.org

Digital to Analogue Conversion (DAC)

You might have noticed that using a PWM is somewhat cumbersome. It’s advantage is simplicity, and particularly suited to tasks such as DC motor speed control (it is relatively simple to switch a power transistor on and off at high speed in order to drive a motor).

However, for outputting complex waveforms, such as audio or medical signals, you are advised to use a high precision and linear Digital to Analogue Converter (DAC). The micro controller you are using in these labs has an on-chip 12-bit DAC.

mbed provides an object AnalogOut which **encapsulates** all the necessary code to access the DAC.

TASK 5.2.3 - B

Modify the code opposite to generate a 5Hz sinusoidal tone (use the sin function).

Add the RC filter (from the previous task) to the output

Observe the output on the scope to check the frequency

Can you actually “see” the impact of the filter? If not, why not?

Now multiply the output signal by a factor of (1.0/1024), as if this was a “quiet section” in a music track. Increase the gain of the scope. Compare the signal before and after the filter. Comment on the result.

```
#include "mbed.h"
#include "math.h"

AnalogOut  aOut(PA_5);

int main(void) {
    while (true) {
        for (float i=0.0f; i<1.0f; i+=0.01f) {
            aOut = i;
            wait(0.001f);  //approx. 1kHz
        }
    }
}
```

TASK 5.2.3 - C

Modify the previous code to use a Ticker object to generate the output tone of 5Hz. Refer back to Task 4.4.2 for a generalised pattern.

Increase the frequency to 100Hz. What do you notice?

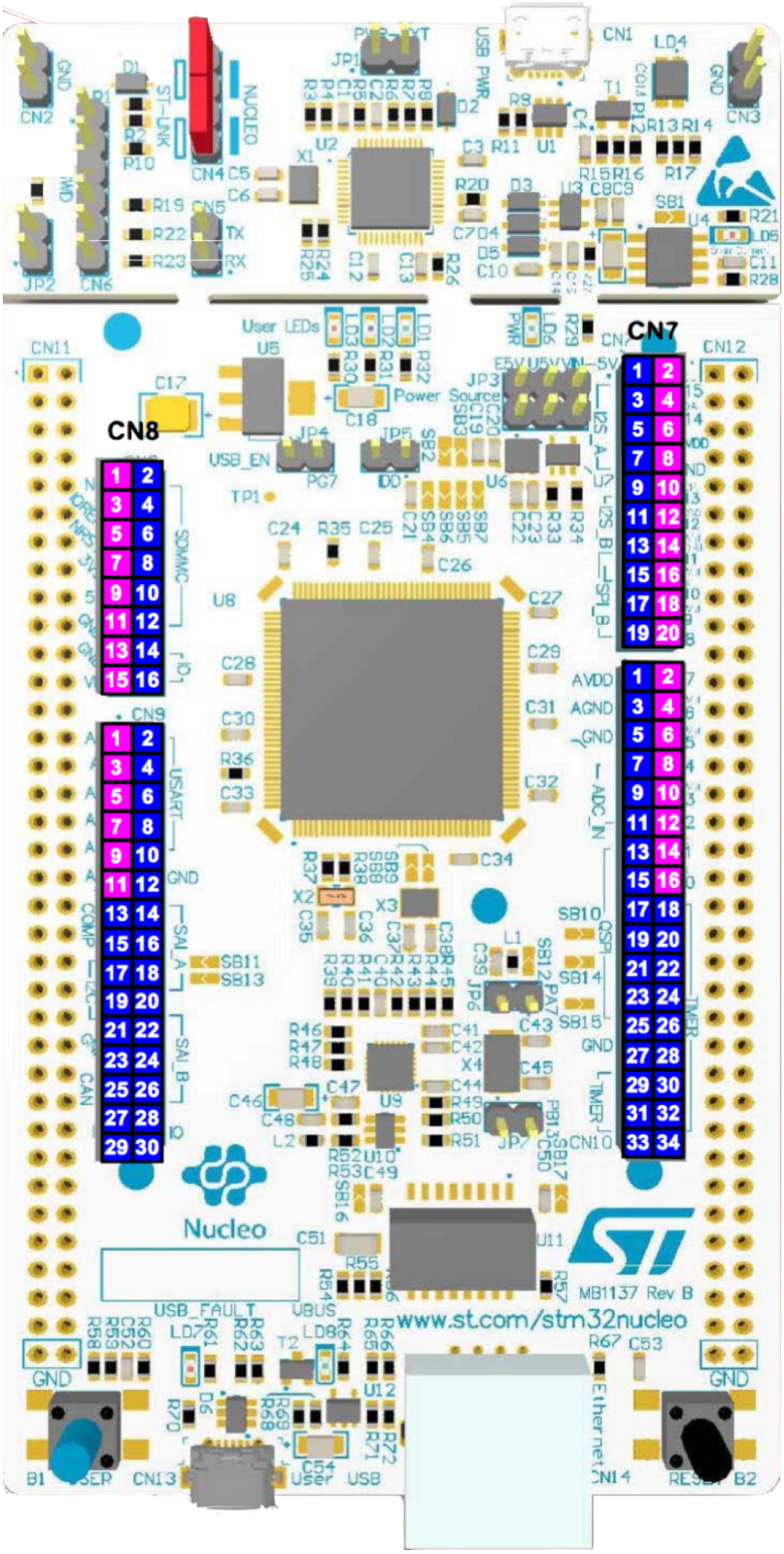
Appendix – NUCLEO F429ZI Connections

CN8

NC	NC	1	2	D43	PC8
IOREF	IOREF	3	4	D44	PC9
RESET	RESET	5	6	D45	PC10
+3V3	+3V3	7	8	D46	PC11
+5V	+5V	9	10	D47	PC12
GND	GND	11	12	D48	PD2
GND	GND	13	14	D49	PG2
VIN	VIN	15	16	D50	PG3
PA3	A0	1	2	D51	PD7
PC0	A1	3	4	D52	PD6
PC3	A2	5	6	D53	PD5
PF3	A3	7	8	D54	PD4
PF5	A4	9	10	D55	PD3
PF10	A5	11	12	GND	GND
NC	D72	13	14	D56	PE2
PA7	D71	15	16	D57	PE4
PF2	D70	17	18	D58	PE5
PF1	D69	19	20	D59	PE6
PF0	D68	21	22	D60	PE3
GND	GND	23	24	D61	PF8
PD0	D67	25	26	D62	PF7
PD1	D66	27	28	D63	PF9
PG0	D65	29	30	D64	PG1

CN7

PC6	D16	1	2	D15	PB8
PB15	D17	3	4	D14	PB9
PB13	D18	5	6	AVDD	AVDD
PB12	D19	7	8	GND	GND
PA15	D20	9	10	D13	PA5
PC7	D21	11	12	D12	PA6
PB5	D22	13	14	D11	PA7
PB3	D23	15	16	D10	PD14
PA4	D24	17	18	D9	PD15
PB4	D25	19	20	D8	PF12
AVDD	AVDD	1	2	D7	PF13
AGND	AGND	3	4	D6	PE9
GND	GND	5	6	D5	PE11
PB1	A6	7	8	D4	PF14
PC2	A7	9	10	D3	PE13
PF4	A8	11	12	D2	PF15
PB6	D26	13	14	D1	PG14
PB2	D27	15	16	D0	PG9
GND	GND	17	18	D42	PE8
PD13	D28	19	20	D41	PE7
PD12	D29	21	22	GND	GND
PD11	D30	23	24	D40	PE10
PE2	D31	25	26	D39	PE12
GND	GND	27	28	D38	PE14
PA0	D32	29	30	D37	PE15
PB0	D33	31	32	D36	PB10
PE0	D34	33	34	D35	PB11


CN7
CN8
Nucleo
MB1137 Rev B
www.st.com/stm32nucleo

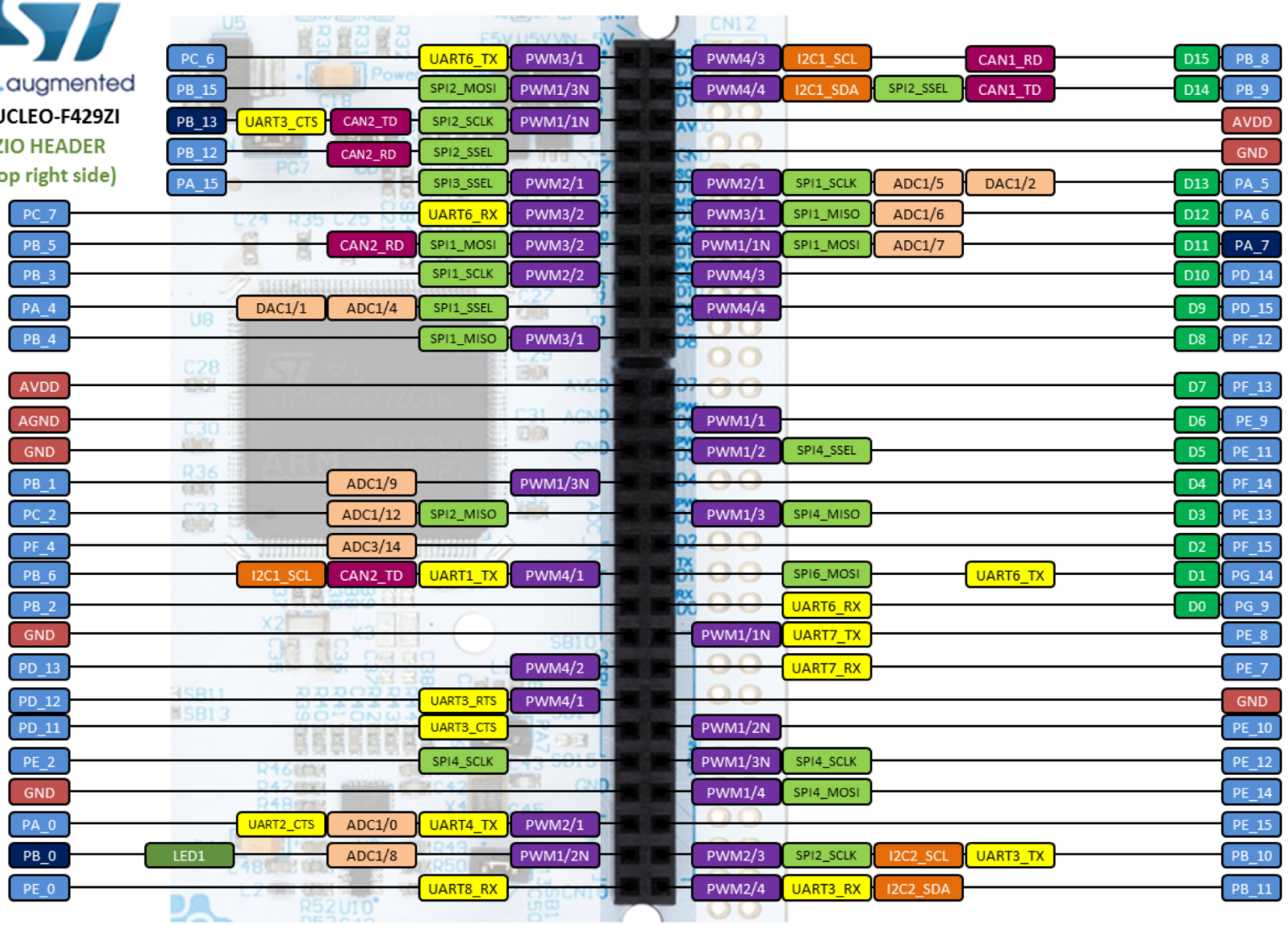
life.augmented

NUCLEO-F429ZI

ZIO HEADER

(top right side)

CN7



life.augmented

NUCLEO-F429ZI

ZIO HEADER

(top left side)

CN8