
Embedded Programming with mbed-os

A Practical Course for Electronic Engineers

Dr. Nicholas Outram, University of Plymouth

1st Edition

Forward	4
Digital Outputs	6
<i>An introduction to mbed-os.....</i>	<i>6</i>
Activity 1.1 - Simple LED Circuit	6
Activity 1.2 - Blinky.....	8
Activity 1.3 - Hello World.....	13
Traffic Lights	14
Serial Input.....	16
Using functions	17
Review	18
Digital Inputs and Multiple Outputs	19
<i>An introduction to mbed-os</i>	<i>19</i>
Non-Blocking Polling	26
Interrupts - a first look.....	30
Race Conditions and Corruption.....	33
Multiple Outputs.....	35
Important Terminology.....	36
Key Points on Interrupts	39
Analogue I/O.....	41
<i>Interfacing to the analogue world with mbed-os</i>	<i>41</i>

Interrupts.....	46
mbed Ticker.....	48
Accurate Sampling with a Timer Interrupt	51
Analogue Output.....	52

Concurrency and Synchronisation61

Using Interrupt and Thread Primitives.....61

Introduction to Multi-Threaded Programming.....	69
Multiple Threads and Synchronisation.....	73
Scheduling and Priority.....	90
Thread Abstractions	93
Network Programming.....	99
Reading and Writing the SD Card.....	100
Environmental Sensor	101

Appendix – NUCLEO F429ZI Connections....103



Forward

Computers are digital electronic devices that interface to the real world. This module aims to give you an introduction to how computers interface to the real word in real-time, including both digital and analogue signals.

This course is designed for stage-3 undergraduate students in Electronics and/or Robotics, who have prior experience in both electronics and C programming. Most electronic devices contain at least one microcontroller (MCU), but that microcontroller will typically be interfaced to multiple peripherals, such as sensors, motors and communication interfaces. Such devices produce or consume data in real time.

MCUs typically work with real-world signals, many of which are asynchronous in nature. To avoid missing an event or losing data, they are required to respond to inputs and generate outputs in a

timely and predictable manner. Often, response times are bounded, making it critical to guarantee responsiveness under all conditions.

The MCU is often required to service its inputs and outputs simultaneously (“concurrently”). Unlike an FPGA, a single core MCU can only perform one task at a time. In addition, most external input/output devices are much slower than the MCU, meaning the MCU often must wait for operations to complete, while still servicing other devices. This module is concerned with understanding the strategies, risks and limitations to overcome these problems in a safe and scalable way.

As the number of devices increases, so the complexity of your software grows. It is critical to write and structure software in a way that scales gracefully and that is testable. One popular approach is to divide your functionality into “software components”, or objects. This is known as Object Orientated Programming (OOP). You will learn the benefits and techniques to use software objects (mbed-os is an OOP framework) and the mechanism for creating and testing them. For this, you will learn the C++ language and the basics of OOP.

We will be using the mbed-os® framework from ARM, which can be thought of as a superset of the popular mbed classic. Both are object orientated frameworks. A key feature of mbed-os is the

ability to multi-task in a way that allows the engineer to safely manage real-time signals.

Intended Learning Outcomes

1. Critically discuss and apply reasoning to the operation of single-threaded and multi-threaded computer programs and manipulate data in advanced embedded platforms.
2. Develop structured algorithms that exploit concurrency and/or parallelism in modern microprocessors.
3. Use appropriate software tools to produce and debug programs to solve given problems and test solutions.

Setting the Right Pace

To complete all tasks will require self study and working in your own time. There may be additional tasks for those that finish early or want to do further self study.

You can take your kits home, so it is also possible to catch up on Wednesday afternoons, evenings and weekends.

Chapter 1

Digital Outputs

An introduction to mbed-os

Welcome to Advanced Embedded Programming.

This module is a first introduction to the subject of interfacing with the mbed-os® framework from ARM. The focus of this module is on software, written to work in modern electronic systems.

In this section you will learn to use / consume software components (objects) from the mbed frameworks, and later, you will learn how to create your own.

Activity 1.1 - Simple LED Circuit

In electronics, we mostly work with one (or both) of the following signals types :

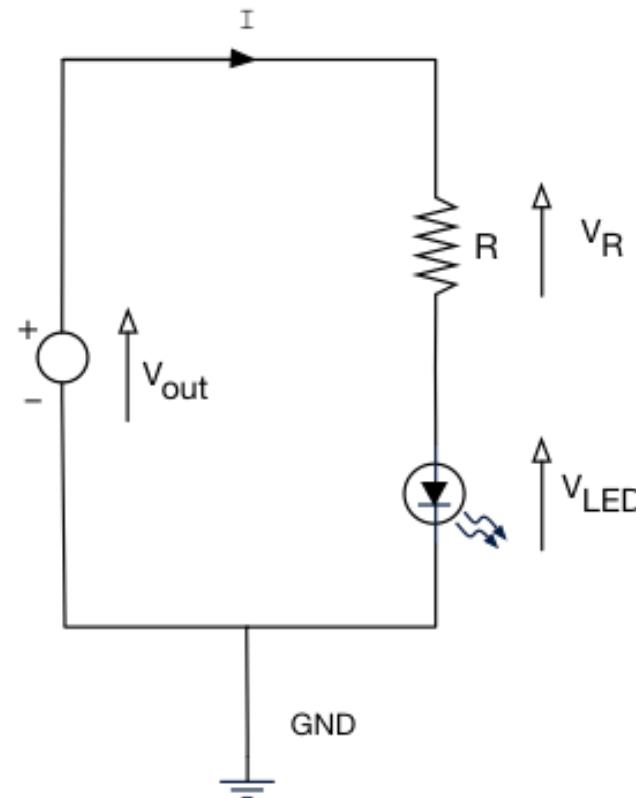
Analogue - where voltages and currents can take any value, typically between an upper and lower limit

Digital - where signal voltages can only be one of two possible values

For this section, the signals are all going to be digital. Let's start with a simple circuit schematic as shown.

All components should be mounted on the prototyping board provided, and nothing will work without a power supply!

Is obtained from the Nucleo Board.



LED Characteristics

In your kit, you are provided with three LEDs, red, green and amber. The data sheets for these devices are available. However, the most important characteristics for a clear visible brightness are as follows:

	current (approx.)	Voltage
RED	2mA	1.7V
AMBER	2mA	1.85V
GREEN	2mA	1.9V

LED Voltage at 2mA

You can drive much higher currents through these LED's but the additional perceived brightness is not significant. We want to keep current **LOW** for two reasons:

- The source of power for the LED will be our microcontroller. From the data sheet, the maximum output current is limited to 25mA for a single pin, up to a total maximum of 120mA for all pins.
- We want to (and should) try and save power

- There will be tolerances around these values, and they do not need to be precise.

Consider the green LED in the circuit above.

- The voltage source $V_{out}=3.3V$
- We want the current $I=2mA$
- For this current, from the LED data we expect $V_{LED}=1.9V$

TASK

Calculate R

As a simple test, wire up this circuit opposite on the prototyping board.

We are using the digital 3.3V and ground (GND) pins on the Nucleo board to power this circuit. See the glossary item Nucleo Board to help you find the 3.3V power pins (+3V3 and GND).

Confirm you found the correct pins on the header CN8.

Once confirmed, connect your F429ZI Nucleo board to the USB port of your PC, and the LED should light.

This confirms the power supply is working. Leave this connected as it will be used throughout this section.

Note that +3V3 and GND are the DIGITAL power sources, not to be confused with the analogue.

Activity 1.2 - Blinky

When learning to program a new language or framework, there are one or two traditional programs you almost always write - here is the most popular in the embedded software world, "blinky".

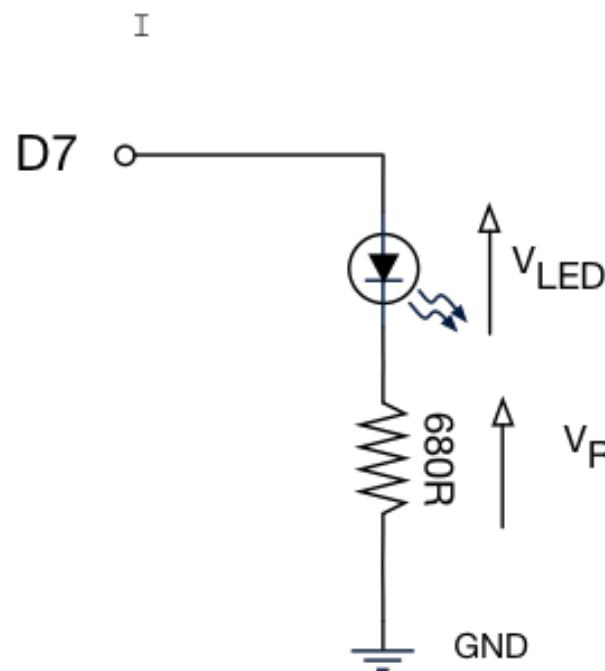
Before we write this software, first you modify the circuit.

- The figure opposite is the schematic for the blinky circuit.
- You will find the GPIO D7 labelled on the **Nucleo Board**.

If you were to write some code to flash the LED by referring to the microcontroller data sheet, there would be a number of things you would have to do, including:

- Set the mode of the GPIO pin (output, push/pull)
- Turn on the clock
- Set a bit somewhere in a register to set the output value
- Possibly write a for-loop to create a delay?

It's good to know how to do this, but it's not overly productive and will result in code written specifically for a small number of devices (or even just one device).



Using mbed Objects

Opposite is the source code for blinky written with mbed (task 1.2.1). Let's break this down line by line.

Firstly, consider a simple variable declaration in C, we might write something like this:

```
int x = 0;
```

The variable is called `x`, it's **data type** is `int` (integer) and it is initialised to the value 0.

Now let's look at one of the first lines in the Blinky code.

```
DigitalOut myled(D7);
```

This is the declaration and definition of the object variable `myled`, where the data type is `DigitalOut`. This is a custom data type that is included with mbed.

Note how a single parameter (the pin name) is passed. This is because when you initialise an **object**, you have the option to pass parameters that allow the object to initialise its internal state.

Note how simple the code is!

```
//This is known as a "header file"
//In short, this copies and pastes the text file
//mbed.h into this code
#include "mbed.h"

//Create a DigitalOut "object" called myled
//Pass constant D7 as a "parameter"
DigitalOut myled(D7);

//The main function - all executable C / C++
//applications have a main function. This is
//our entry point in the software
int main() {

    // ALL the code is contained in a
    // "while loop"
    while(1)
    {
        //The code between the { curly braces }
        //is the code that is repeated
        myled = 1; // External LED is ON
        wait(1.0); // 1 second
        myled = 0; // LED is OFF
        wait(1.0); // External 1 second
    }
}
```

The complex details are hidden away inside the software component `DigitalOut`. We are using a `DigitalOut` component to set a pin (D7) high or low just by equating it to 1 or 0.

All the initialisation code, including write to registers to turn on the clock, set GPIO mode etc. are hidden inside the component `DigitalOut`.

When we add an instance of this component to our application, note we are also telling it which pin it is assigned to by passing a parameter. You can create more than one instance, but each instance must use a unique pin.

In our code, we say the object `myled` is of an **instance** of type `DigitalOut`

One of the great things about components is that they hide the complex details inside, again very much like an electronic

¹ Consider the car as a useful analogy. You can learn to operate a car by using its interface (steering wheel, pedals and gear change stick). A car also has properties which can be monitored and sometimes changed, including:

- fuel level (read/modify)
- tyre pressure (read/modify)
- engine temperature (read only)

Details workings of a car are hidden from you. You do not need to take the engine apart to be able to drive a car. It's hard to build a car, but relatively easy to drive one. We call this *abstraction*. A software component (or object) is very similar in these respects. You don't need to see the inner workings of an object in order to use it. mbed includes a collection of very useful objects.

component. *We just have to work with its external interface*. This promotes reuse and shorter / easier to write code¹.

The mbed framework contains many such components, such that if you rebuild your code on a different board, it should just work (you might have to change the pin assignments). We say it is **portable**.

C++ also supports something known as operator overloading (changing the contextual meaning of +,-,= etc..). This is used extensively on MBED to make the resulting code read more intuitively.

Consider the following line.

```
mled = 1;
```

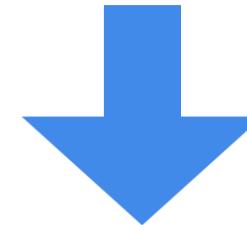
Here we simply assign an integer value to an instance of DigitalOut, and the output pin D7 changes.

Such clarity and simplicity is possible because the component understands the '=' operator to mean "set the pin to this value".

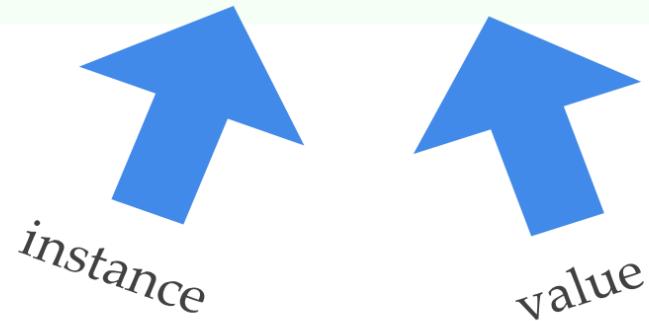
That is because for DigitalOut, **mbed redefines the meaning of the = operator.**

We will meet a topic called "operator overloading" shortly where you will learn to do this yourself. Done well, this results in easy to read and debug code. Done badly, and you can end up with ambiguous code!

assignment



```
myled = 1;
```



In C++ it is possible to redefine the function of the = assignment operator (and others) using a process known as "*operator overloading*". This is a short cut for writing myled.write(1);

Now consider the next line:

```
wait(1.0);
```

wait is one of the global utility functions in mbed².

In this line of code, we are invoking a function “wait”. This simply delays execution for a specified time. This function is written for us (it is part of the mbed library).

wait has a single parameter. Note this time it is a fractional number. Note that when you invoke `wait()` in this way, the CPU can do nothing else. We say it is a blocking wait.

call the wait
“function”



```
wait(1.0);
```



Pass a parameter
1.0

TASK 1.2.1

Modify the software to do the following

1. Change the ON time to 5s and the OFF time to 2.5s
2. Change the ON time to 1ms (0.001s) and the OFF time to 9ms
3. What do you observe about the LED in (2)?

What we've written is “blinky” - the classic starter application for embedded programming.

² See <https://docs.mbed.com/docs/mbed-os-api-reference/en/latest/APIs/tasks/wait/>

Activity 1.3 - Hello World

The other tradition is Hello World. This is normally the first program you write when learning to program on a desktop computer. However, we're a generous lot, and wanted you to enjoy both with mbed.

See the code opposite. Note the object `Serial`.

```
Serial pc(USBTX, USBRX);
```

This object has a “member function” (function that acts on the data inside the object) `baud` which sets the speed (bits/s).

```
pc.baud(9600);
```

Objects are sometimes said to have “methods” or “member functions” (fancy word for functions that belong to objects). These perform specific tasks that relate to the object (they will typically update its internal state). You could compare this to a pin on a chip that performs a specific task when you pull it high or low. On doing so, a sequence of internal electronic states are updated.

Another function that `Serial` provides is `printf`.

```
pc.printf("Hello World\n");
```

The output of `printf` is sent to the `Serial` device represented by `pc`.

```
#include "mbed.h"

//Create an instance of a Serial
//object called pc.
//Transmit and receive pins have pre-defined
//names USBTX and USBRX
Serial pc(USBTX, USBRX);

int main()
{
    //Set the baud rate property (bits per second)
    pc.baud(9600);

    //Call the printf method on pc
    pc.printf("Hello World\n");

    //Run in an infinite loop
    while(1) {
    }
}
```

Watch the following video

<https://plymouth.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=1f5a72c3-9458-4084-96b0-a5c9b973274f>

Debugging with printf?

It is common to use `printf` function to debug embedded systems, and although a debugger is preferable during the development phase. It can also be invaluable for diagnostics and logging. However, it is also quite a complex function and can consume a significant amount of CPU time and memory.

Have a go at the following tasks:

TASK 1.3.1

Modify the software to write Hello World to the serial interface every 1s. Don't forget the new line character `\n`.

Hint - write your code inside the while loop.

TASK 1.3.2

Add two additional LED's to the breadboard. Don't forget to include the current limiting resistors.

Drive the three LEDs separately using D5, D6 and D7

Write some code to generate a traffic light sequence that repeats forever.

RED, RED-AMBER, GREEN, FLASHING-AMBER

then repeat (use a while loop)

Add `printf` statements for each state, and write to the terminal (for debug purposes).

Traffic Lights

Let's now combine what we've learned. In this example we will use `printf` to debug and trace our code.

TASK 1.3.3

TASK REMOVED

TASK 1.3.4

Modify your software to use a `BusOut` object instead of a `DigitalOut`

See <https://developer.mbed.org/handbook/BusOut> for details

```
#include "mbed.h"

Serial pc(SERIAL_TX, SERIAL_RX);

int main()
{
    char nameStr[30]; // array of chars (string)
    int age;          // integer to hold your age

    pc.printf("%s %d", nameStr, &age);
    pc.printf("Hello %s \n\r", nameStr);
    pc.printf("You are %d \n", age);
}
```

TASK 1.3.5

Modify your software to use a `Ticker` to perform the flashing?

See <https://developer.mbed.org/handbook/Ticker> for details

This task requires is a little different as it uses an *interrupt* (a topic we will cover quite soon), and you need to read the documentation and look at the example code. You should get into the habit of reading the documentation on all the mbed Classes.

Serial Input

You can also type into the terminal and read the data from the Serial Interface.

Question: What statements are blocking in this code?

TASK 1.3.6

Build and test the code shown below. In a PuTTY terminal, enter a name followed by a space then a number and press enter.

Can you explain what is happening? If not ask! Change the code to ask for two numbers which you then multiply and send the answer to the terminal.

Using functions

You may recall C functions from previous years. Here is some revision:

ADDITIONAL TASK 1.3.7

The code opposite does the same as the previous example but the data input section has been moved to a separate **function**.

As a revision exercise, modify the code to move the output section to another function.

Note the variables nameStr and age are now outside of all the functions. They have “global scope” (can be reached from anywhere in your code).

Question: what is mean by global scope, and how does it differ from local scope?

```
#include "mbed.h"

Serial pc(SERIAL_TX, SERIAL_RX);

// array of chars (string)
char nameStr[30];
// integer to hold your age
int age;

void getData()
{
    pc.printf("%s %d", nameStr, &age);
}

int main()
{
    getData();

    pc.printf("Hello %s \n\r", nameStr);
    pc.printf("You are %d \n", age);
}
```

ADDITIONAL TASK 1.3.9 (Challenge)

Read the keyboard input and translate into morse code.

Echo the morse code using LEDs - one colour for letters and another colour for numbers.

Begin with just one letter or number, and then a whole word or long number.

Output summary of time taken by each word to the console for transmission billing purposes.

Review

In this section, we wrote some very concise code that has only one task. There was no need for concurrency and assuming the `wait()` function is power-efficient, then nothing we have done is particularly wasteful of power.

LEDs were controlled using General Purpose Input / Output (GPIO). So we have only considered a Digital Output.

We also used two more objects:

`BusOut` - for writing multiple GPIO pins

`Serial` - for communicating with a UART

In the next section, we add digital inputs, which brings new challenges if we still want to be power efficient and responsive.

Chapter 2

Digital Inputs and Multiple Outputs

An introduction to mbed-os

In this section, we will look at digital inputs, and how we might use them to control digital outputs.

We will be using a simple “**push-to-make switch**” to generate a single digital bit of data.



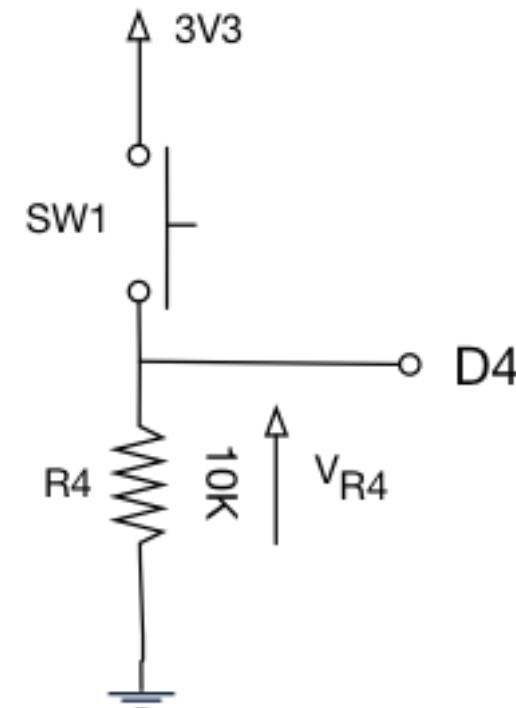
Activity 3.1 - Simple Switch Input

Remember that a single digital signal is either ON or OFF.

Numerically, we represent this as 1 or 0 (a single binary digit).

The simplest way to generate a single bit input signal is with a push switch (shown opposite). The schematic to generate a digital input is as follows.

Wire up this circuit on your prototype board.



[Using a SPST switch as a digital input](#)

Activity 3.2 - Reading Digital Inputs

Wire up 3 LED's as shown here. These will be used for outputs.

For this task, we are going to read the digital input using software.

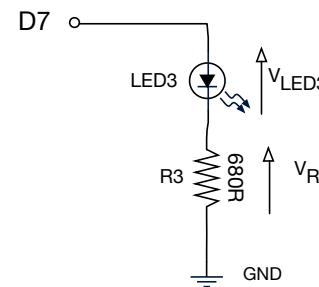
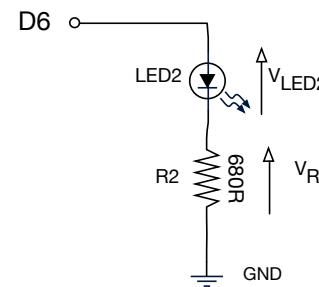
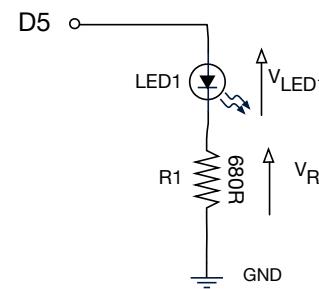
The objective is as follows:

- When the user depresses a button, the red LED switches ON and stays ON (until the system is reset).

Remember - the input is connected to D4.

Maybe unsurprisingly, we use a component called a `DigitalIn`.

TASK 3.2.1	
Create a new project and paste in the code (below). Build and run to test.	
When the yellow and green lights come on, press and release SW1.	
Press the black RESET button on the Nucleo board to run again.	
Question: which line(s) is considered to be blocking?	
Task: Sketch a flow chart for this code	



Under the hood, there is a significant amount of work going on to set up the hardware in the right mode. The specifics are also vendor / board dependent.

We benefit greatly from **hardware abstraction** provided by the mbed C++ classes DigitalIn and DigitalOut. mbed provide identical interfaces for all supported mbed compliant boards. This is an example of a **device driver**.

Once again, we use of the the component from mbed to make interfacing with hardware simple.

```
DigitalIn SW1(D4);
```

where SW1 is an instance of DigitalIn, which can read pin D4.

A critical line in this code is as follows:

```
while (SW1 == 0) { }
```

While the digital input SW1 is equal to zero, this code **blocks the CPU** in a tight **polling loop**. This is also known as **spinning**.

This is NOT power or CPU efficient

Once the switch is pressed, an attempt to read SW1 will return a 1 and so the CPU will unblock.

```
#include "mbed.h"

DigitalOut red_led(D7);
DigitalOut yellow_led(D6);
DigitalOut green_led(D5);
DigitalIn SW1(D4);

int main() {

    //Switch on Yellow and Green to indicate
    //that the code is running
    yellow_led = 1;
    green_led = 1;
    red_led = 0; //Set RED LED to OFF

    // Wait for SW1 to be pressed
    while (SW1 == 0) { }

    red_led = 1;      //Turn ON LED

    while (1) { }      //Repeat forever
}
```

TASK 3.2.2

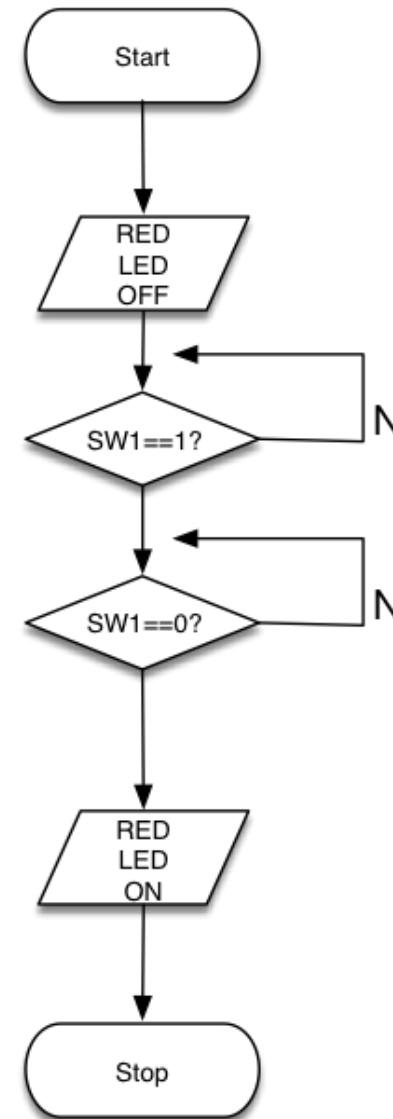
Modify the code so that the red LED only comes on after the SW is pressed **and** then released

Hint: use the flow chart opposite as a guide

Test your solution. Is it entirely reliable?

You might have found this to be an unreliable solution (depending on how clean the switch contacts are).

In the next section, you will learn about “switch bounce” and how to avoid it.



Real World Signals

When working with physical hardware, both hardware and software engineers can take steps to reduce the probability of an erroneous input.

In almost every case, errors are probabilistic. You can never engineer the probability to zero (ok, philosophical point maybe, but often it's a very real issue).

As engineers, we have to recognise and mitigate against such failures, and at least know the probability and impact of failure.

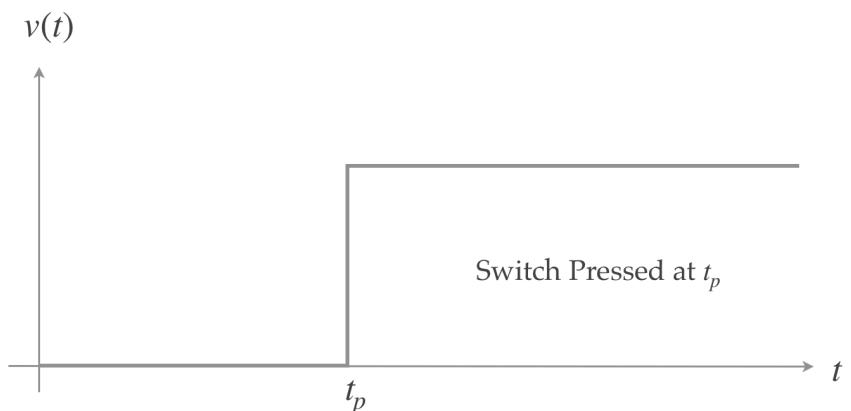
A consumer gadget can accept a higher chance of failure than an anti-lock braking system or fly-by-wire system in a commercial aircraft.

Some medical devices have to work with small and noisy signals from the human body, and have to use sophisticated mathematics and software to manage the error.

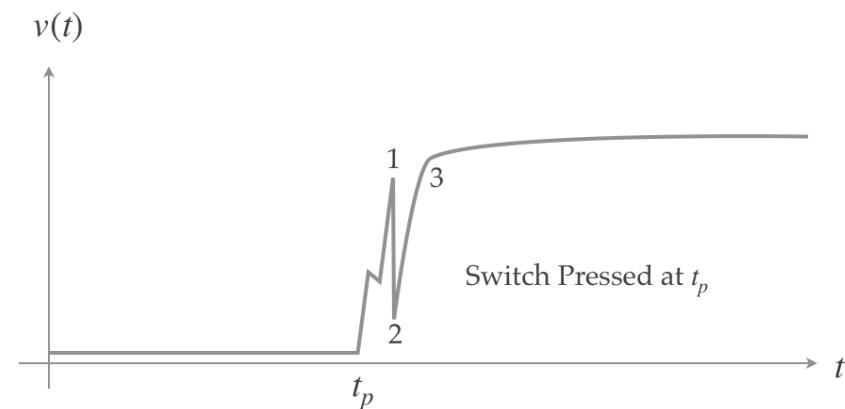
Switch Bounce

The reality is that mechanical switches are not perfect. Most switches suffer switch bounce to some degree.

As you can see in the second diagram opposite, the waveform is not a perfect step like the ideal.



Ideal model of a digital signal generated from a mechanical switch



Depicting switch bounce from a mechanical switch which could be registered as 3 events: (1)ON,(2)OFF,(3)ON.

There are two common phenomena we must consider:

Noise - random signals superimposed on top of the waveform

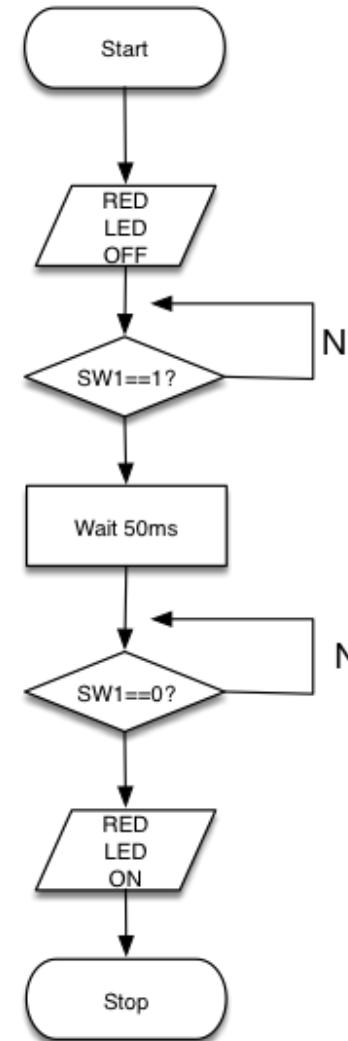
Reactive effects - if you've not covered reactance, it should suffice to say that perfect step waveforms are almost impossible - signals often rise as an exponential curve (although it might be very rapid)

This reinforces the points that electronics is ultimately analog and signals are rarely ideal.

A problem with such phenomena is that their occurrence is stochastic (random, probability based). An engineer must be mindful of such issues, and design systems to minimize the probability of such events occurring or causing malfunction, especially in safety critical systems.

In the case of switch bounce, the exact nature is highly unpredictable.

Consider the revised flow-chart opposite. This has added a delay after the first press is detected.



Adding a delay to address the problem of switch bounce

TASK 3.2.3

Modify the code to add this delay

Does it impact on the user experience? Is the delay long enough? Does it always work? Ask other students for their views. Why might different people give different accounts?

What would happen if you made the delay too long (e.g. 5s)?

hint: use the wait function that we used previously

Critically discuss (in your log book) how reliability is traded against latency. Is there a delay that will make the circuit 100% reliable?

Adding the delay makes the system less responsive. This is often referred to as **latency**. This task illustrates some key points:

- electronic signals are rarely “ideal”
- switch inputs suffer “switch bounce”
- we sometimes have to make trade-offs, such as latency for reliability.

Non-Blocking Polling

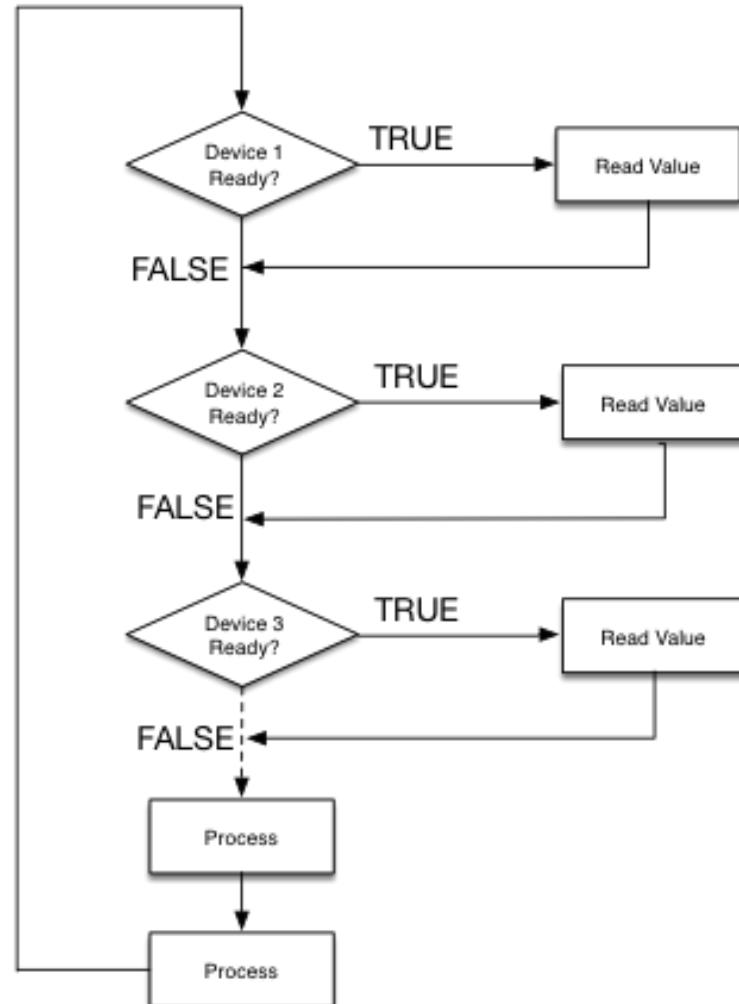
A technique to read data from multiple input sources without blocking a thread of execution.

In the figure opposite, each device is *polled* to see if it has any available data (or if there has been any change). If yes, then the value is retrieved and stored. The next device is then polled.

This method is an improvement over the busy-wait approach as it does not block the “thread” of execution, so all devices are queried at (typically) high rates.

This method does have some disadvantages however. The loop must repeat fast enough to read all devices before data is lost. This again means CPU cycles and power are being consumed even where there is no new input to process. Furthermore, the presence of conditional statements (if/while/switch) means the execution path through the code will vary, and thus the loop timing is likely to jitter. This can cause problems where data must be read at fixed intervals (as we will discover later).

In the next task we implement such a methodology where we poll two devices: a timer and a switch. You should **read** the mbed documentation for the [Timer](#) class before you attempt the next task.



TASK 3.2.4 Polling Method

A. First modify the previous task to be in accordance with the flow chart opposite. *This still uses the busy-wait approach.*

B. Add another switch and a (green) LED to your circuit.

Use D4 and D3 for the switches SW1 and SW2 (DigitalIn)

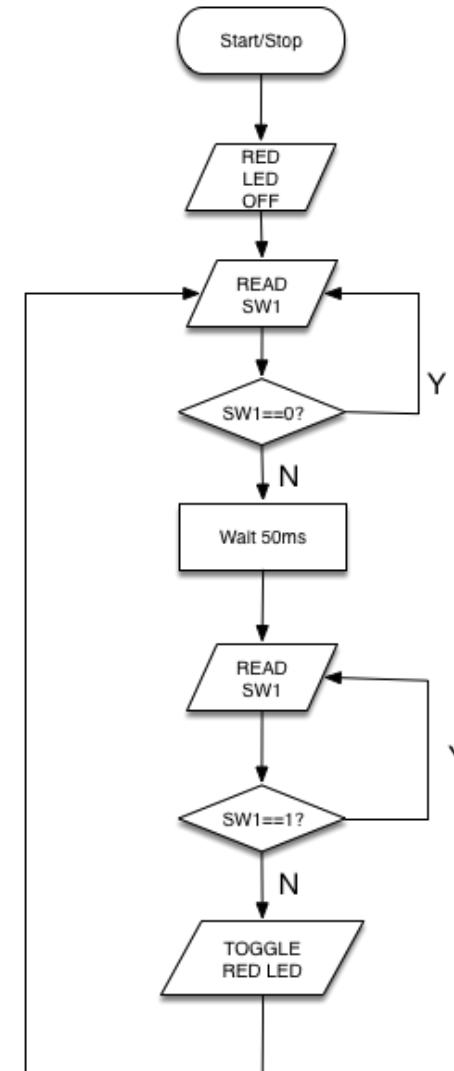
Use D7 and D5 for the RED and GREEN LEDs (DigitalOut)

Your task is to control **BOTH** the red LED with switch 1 **and** the green LED using switch 2 independently. Again, you need to press and release the switch to **toggle** the state of the respective LED, and avoid switch bounce. **Poll a Timer, don't use wait()**. Study and use [Task324](#) sample code as a starting point. Make sure you fully understand it. This uses the polling method and follows the state diagram shown on the following page. You need to scale this to also control the green LED using SW2, without blocking.

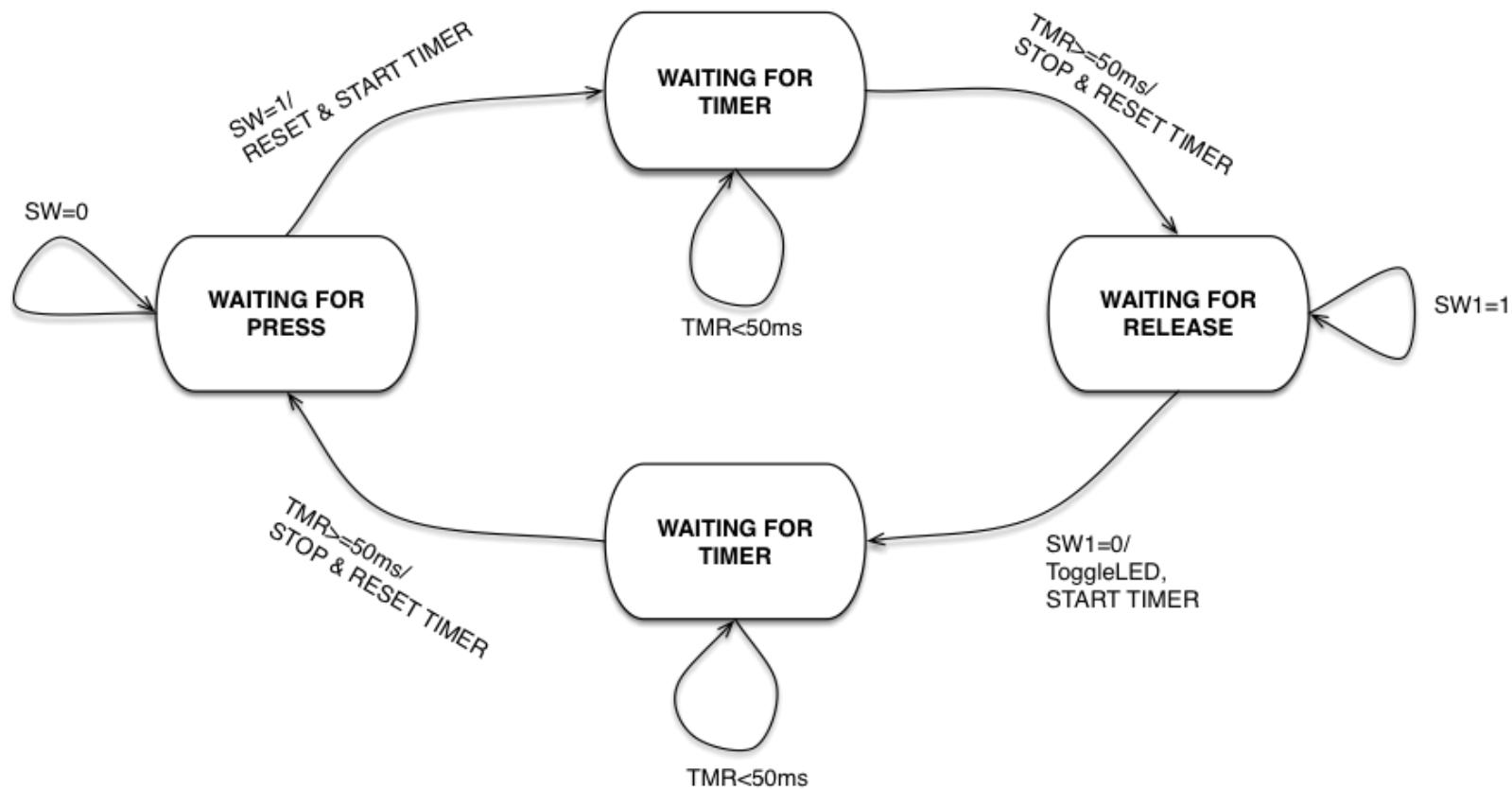
A big difference is that you now have to manage two input switches. **Question:** Why can't you use the busy-wait technique?

Question: Are the switches polled at a constant rate? Explain.

Question: Do we need to poll the switches so fast? (this MCU is driven by a 180Mhz Clock)



Modified flow chart that **toggles** a LED and repeats. This uses busy-wait so will only work for a single input switch



State Diagram for Task 3.2.4 - single switch SW and LED. For state transitions, the format is: {input condition / (mealy) Output}

The basic code for this can be found here <https://os.mbed.com/teams/University-of-Plymouth-Stage-2-and-3/code/Task324/>

The solution is here: <https://os.mbed.com/teams/University-of-Plymouth-Stage-2-and-3/code/Task324Solution/>. Only look at this if you are completely stuck.

Self-Study Challenge: Finite State Machines in C

Look at the code solution and observe how the switch-case is used to implement a state machine. Make notes in your log books on implementation of software state machines using a switch-case.

Self-Directed Task

Using the same hardware as above, write a new project to do the following:

- On power up, the Green LED should flash once a second (1Hz).
- Pressing SW1 should reduce the flashing rate (frequency)
- Pressing SW2 should increase the flashing rate
- Pressing SW1 and SW2 together should reset the frequency to 1Hz
- Take steps to avoid switch bounce. Poll a timer to check the time. Do not use wait()
- Use one state machine per switch / timer pair. Show the tutor your state-diagram before you write the code.

Document this in your log book. Include a state diagram.

Remember that your state diagram has two inputs, a switch and a timer, although you won't necessarily poll both in each state.

Tips:

State diagrams can be sketched on paper, photographed with a smartphone or tablet and inserted into your logbook. **I**

recommend using Microsoft Office Lens and OneDrive to make this much more seamless.

You are strongly advised to complete this task before the next lab session. Remember a taught module is 200Hrs. This should be approximately 16 hours / week split between VHDL and C.

4Hrs / week lectures

4Hrs / week practicals

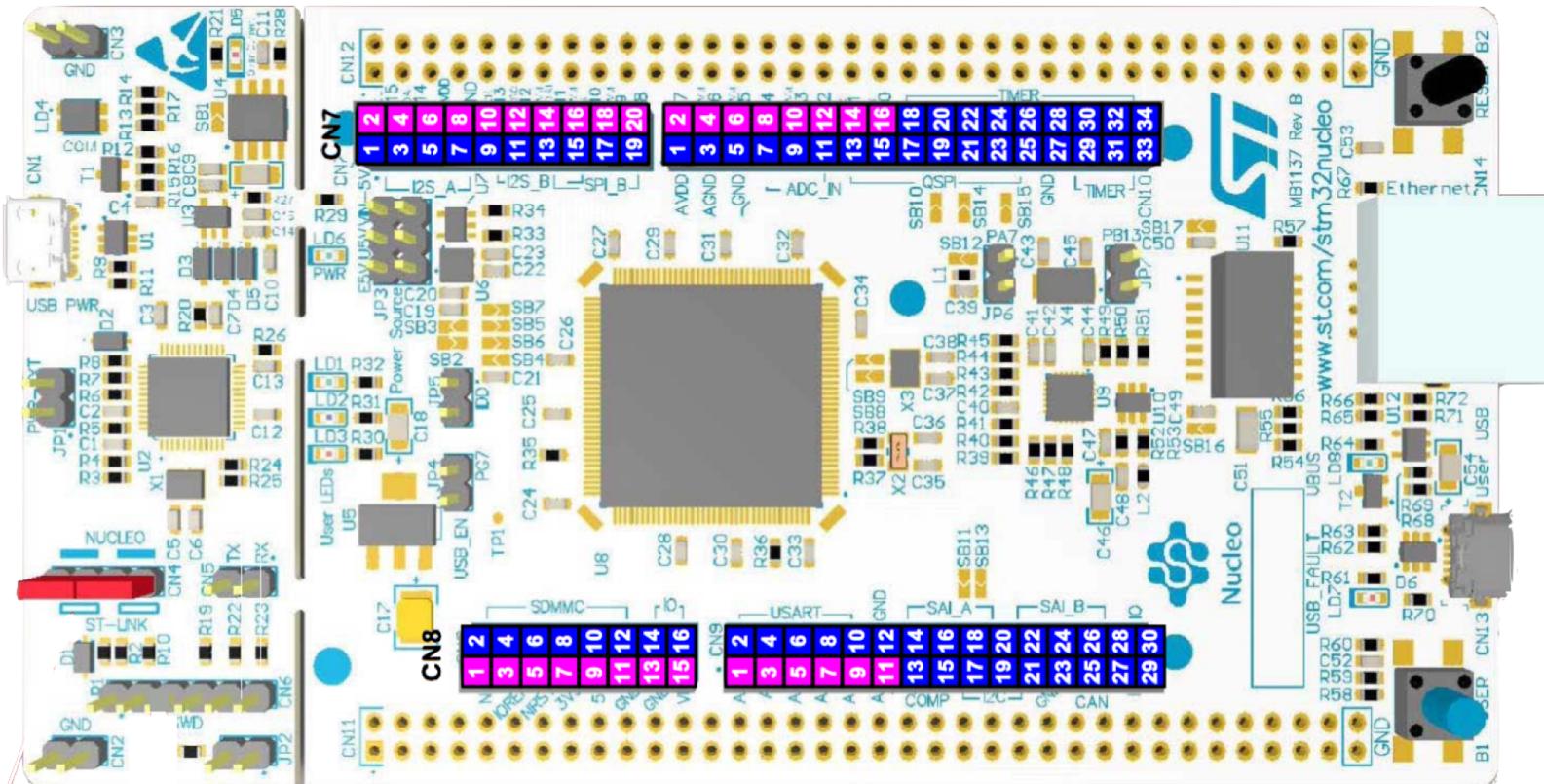
8+hrs/week self study

Advanced Task (last 10%)

The state machines above are Mealy Machines (outputs are changed on state-transitions). Can you re-write this task as a Moore Machine? *Hint: Split the state machine into two distinct switch-case blocks, one for the next state and the next for the output.*

Appendix – NUCLEO F429ZI Connections

		CN8				CN7	
NC	NC	1 2	D43	PC8	PC6	1 2	D15 PB8
IOREF	IOREF	3 4	D44	PC9	PB15	3 4	D14 PB9
RESET	RESET	5 6	D45	PC10	PB13	5 6	AVDD AVDD
+3V3	+3V3	7 8	D46	PC11	PB12	7 8	GND GND
+5V	+5V	9 10	D47	PC12	PA15	9 10	D13 PA5
GND	GND	11 12	D48	PD2	PC7	11 12	D12 PA6
GND	GND	13 14	D49	PG2	PB5	13 14	D11 PA7
VIN	VIN	15 16	D50	PG3	PA4	15 16	D10 PD14
PA3	A0	1 2	D51	PD7	PB4	17 18	D9 PD15
PC0	A1	3 4	D52	PD6	AVDD	19 20	D8 PF12
PC3	A2	5 6	D53	PD5	AGND	1 2	D7 PF13
PF3	A3	7 8	D54	PD4	GND	3 4	D6 PE9
PF5	A4	9 10	D55	PD3	PB1	5 6	D5 PE11
PF10	A5	11 12	GND	GND	A6	7 8	D4 PF14
NC	D72	13 14	D56	PE2	PC2	9 10	D3 PE13
PA7	D71	15 16	D57	PE4	PF4	11 12	D2 PF15
PF2	D70	17 18	D58	PE5	PB6	13 14	D1 PG14
PF1	D69	19 20	D59	PE6	PD6	15 16	D0 PG9
PF0	D68	21 22	D60	PE3	PB2	17 18	D42 PE8
GND	GND	23 24	D61	PF8	GND	19 20	D41 PE7
PD0	D67	25 26	D62	PF7	PD13	21 22	GND GND
PD1	D66	27 28	D63	PF9	PD12	23 24	D40 PE10
PG0	D65	29 30	D64	PG1	PD11	25 26	D39 PE12
					PE2	27 28	D38 PE14
					PA0	29 30	D37 PE15
					PB0	31 32	D36 PB10
					PE0	33 34	D35 PB11



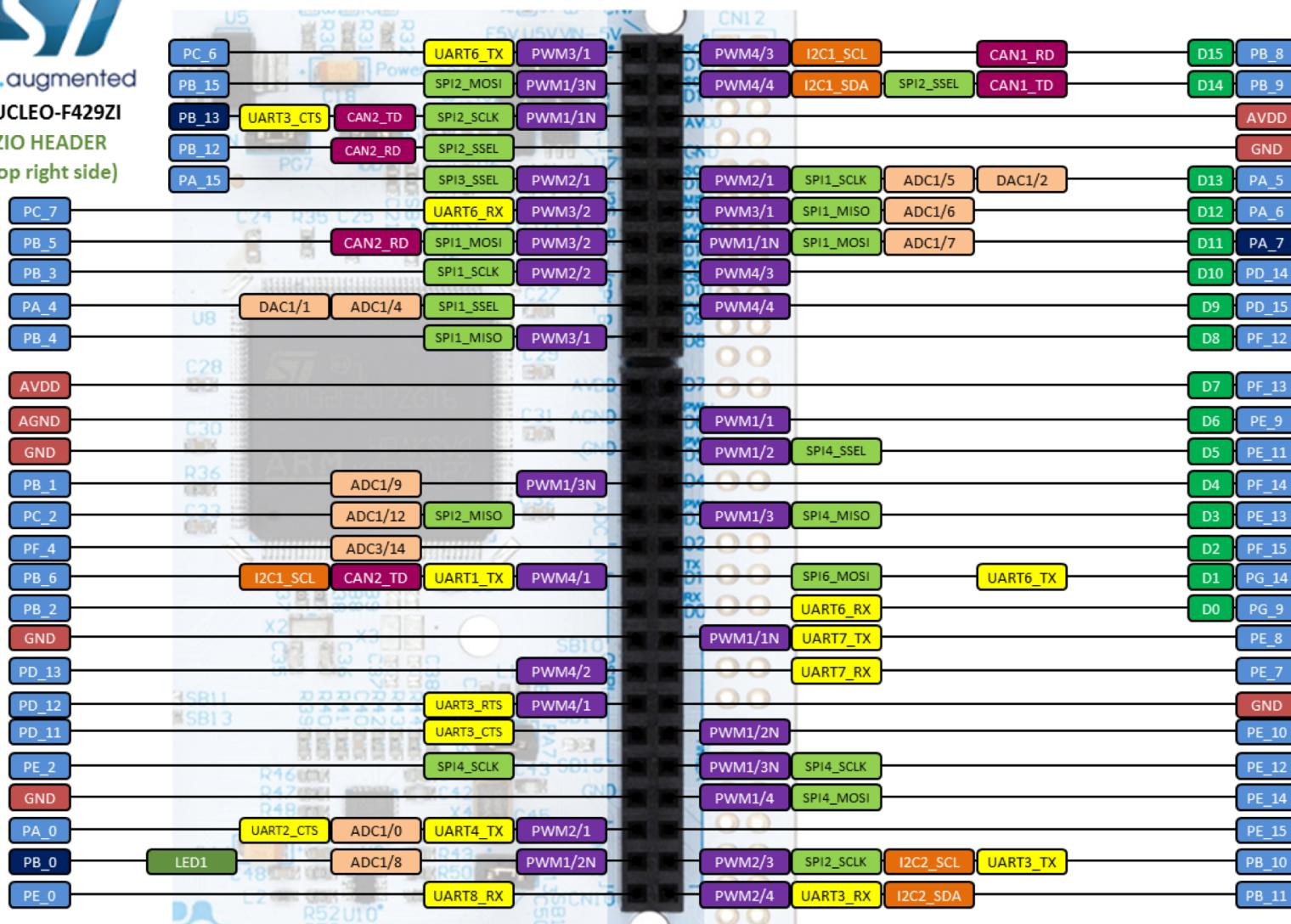


life.augmented

NUCLEO-F429ZI

ZIO HEADER
(top right side)

CN7





life.augmented
NUCLEO-F429ZI
ZIO HEADER
(top left side)

