# Applying the mbed microcontroller

Train The Trainer – IAC

# Digital input and output

**Train The Trainer – IAC**

# Agenda

- Introduction to digital terminology
- Digital outputs on the mbed
- Using LEDs on the mbed pins
- Connecting switches to the mbed
- Implementing a digital switch input on the mbed

# Introduction to digital terminology

- The mbed uses a power rail of 3.3 Volts - 0 Volts indicates ‘off’ and 3.3 Volts indicates ‘on’.
- A number of terms are used interchangeably to describe on and off in digital systems:

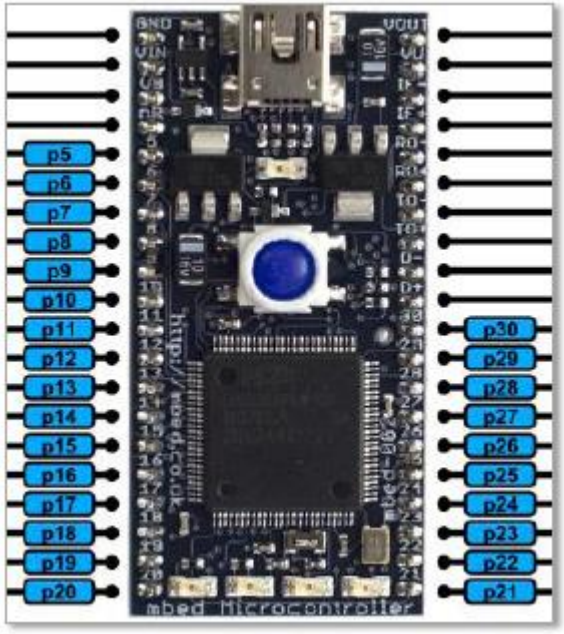
| 0V | 3.3V |
|---|---|
| Open | Closed |
| Off | On |
| Low | High |
| Clear | Set |
| logic 0 | logic 1 |
| False | True |

- Note: terms ‘logic 0’ and ‘logic 1’ may be simply referred to as ‘0’ and ‘1’

# Digital outputs on the mbed

- On the mbed, the four on-board LEDs are digital outputs which have been specially configured to operate with no extra wires or connections needed.
  - LED1 →P1.18, LED2 → P1.20, LED3 →P1.21, LED4 → P1.23

- The mbed also has 26 digital IO pins (pins 5-30) which can be configured as inputs or outputs.



# Digital outputs on the mbed

- The available library functions are shown in the table below.

| DigitalOut | A digital output, used for setting the state of a pin |
|---|---|
| **Functions** | **Usage** |
| DigitalOut | Create a DigitalOut connected to the specified pin |
| write | Set the output, specified as 0 or 1 (int) |
| read | Return the output setting, represented as 0 or 1 (int) |
| operator= | A shorthand for write |
| operator int() | A shorthand for read |

# Digital outputs on the mbed

- The digital IO pins are configured by defining them at the start of the program code.
- Each digital IO is given a name and associated pin, for example:

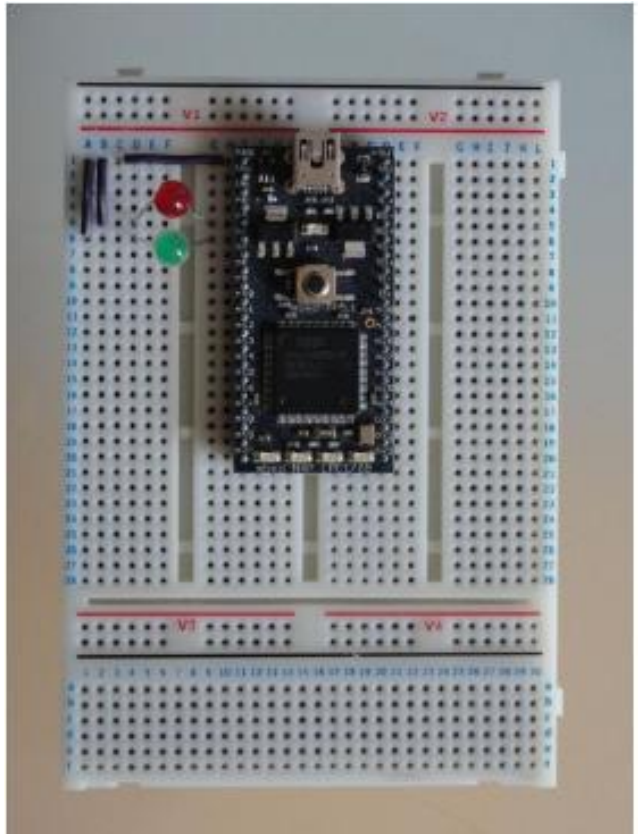
```
DigitalOut myname1 (p5);
DigitalOut myname2 (p6);
DigitalOut myname3 (p7);
```

- The DigitalOut interface can be used to set the state of the output pin, and also read back the current output state.
- Set the DigitalOut to 0 to turn it off, or 1 to turn it on.

# Using LEDs on the mbed pins

- Connect the mbed to a breadboard and attach a red LED to pin 5 and a green LED to pin 6.

- Remember to attach the positive side of the led (the side with the longer leg) to the mbed. The negative side should be connected to ground.
- The mbed has a common ground on pin 1.



# Using LEDs on the mbed pins

- Exercise 1: Create a new program for the external LED project.
- Modify the default main.cpp code to become the following:

```
#include "mbed.h"
DigitalOut redled(p5);
DigitalOut greenled(p6);
int main() {
    while(1) {
        redled = 1;
        greenled = 0;
        wait(0.2);
        redled = 0;
        greenled = 1;
        wait(0.2);
    }
}
```

- Compile, download and run the code on the mbed.

# Using LEDs on the mbed pins

- Look at the example program and identify the key C programming elements as follows:
  - The mbed.h library file is linked to by the `#include' statement.
  - DigitalOut objects are defined with a name and a chosen mbed pin.
  - The main program function exists inside `int main() { ... program... }'.
  - An infinite loop is implemented by the while(1) statement, so that the program continuously loops forever, allowing the led to flash continuously.
  - Digital outputs are controlled simply by setting the relevant objects equal to 0 or 1.
  - The mbed wait() function is used for timing control.

# Digital inputs on the mbed

- Digital inputs values can be read.
- As with digital outputs, the same 26 pins (pins 5-30) can be configured as digital inputs, as follows:

```
DigitalIn myname1 (p5);
DigitalIn myname2 (p6);
DigitalIn myname3 (p7);
```

- The DigitalIn Interface determines the current logical state of the chosen input pin, e.g. logic '0' or logic '1'.
- Zero volts on a digital input pin returns a logical 0, whereas 3.3 Volts returns a logic 1.

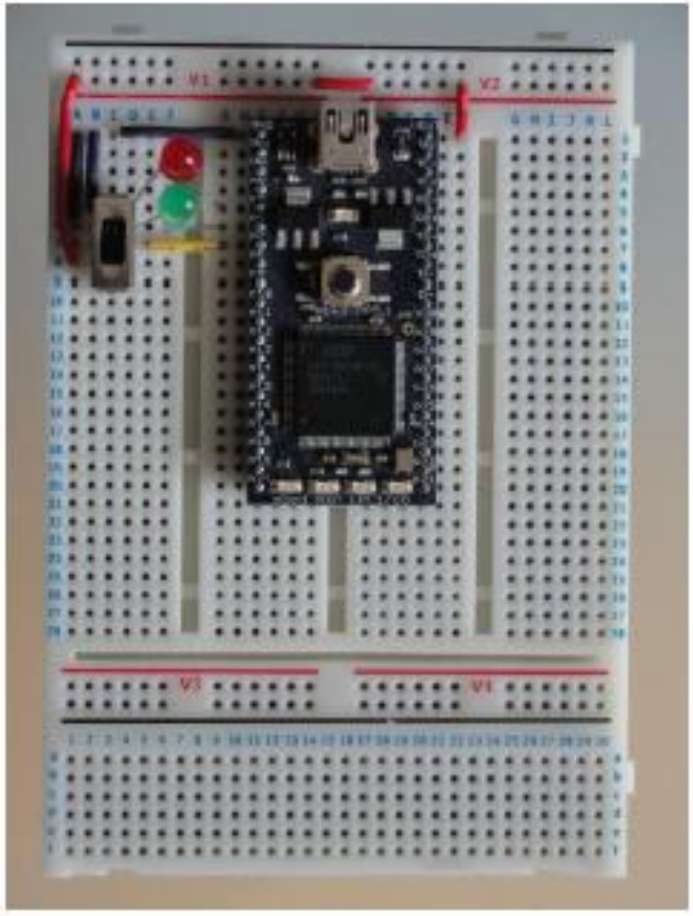
# Digital inputs on the mbed

The available library functions are shown in the Table below.

| DigitalIn | A digital input, used for reading the state of a pin |
|---|---|
| **Functions** | **Usage** |
| DigitalIn | Create a DigitalIn connected to the specified pin |
| read | Read the input, represented as 0 or 1 (int) |
| mode | Set the input pin mode |
| operator int() | A shorthand for read |

# Connecting switches to the mbed

- A simple mechanical switch to set a digital input pin either high (‘1’) or low (‘0’) by connecting it to switch between the 3.3V and GND rails.



- To the setup included in the previous example, add a mechanical switch output to pin 7.
- Connect the switch inputs to 0V (GND, pin1) and 3.3V (Vout, Pin 40).

# Implementing a digital switch input on the mbed

- Exercise 2: Create a new program for the LED switch project. Modify the default main.cpp code as shown.

- When two forward slash symbols (//) are used, the compiler ignores any proceeding text, so we can use this to write useful comments.

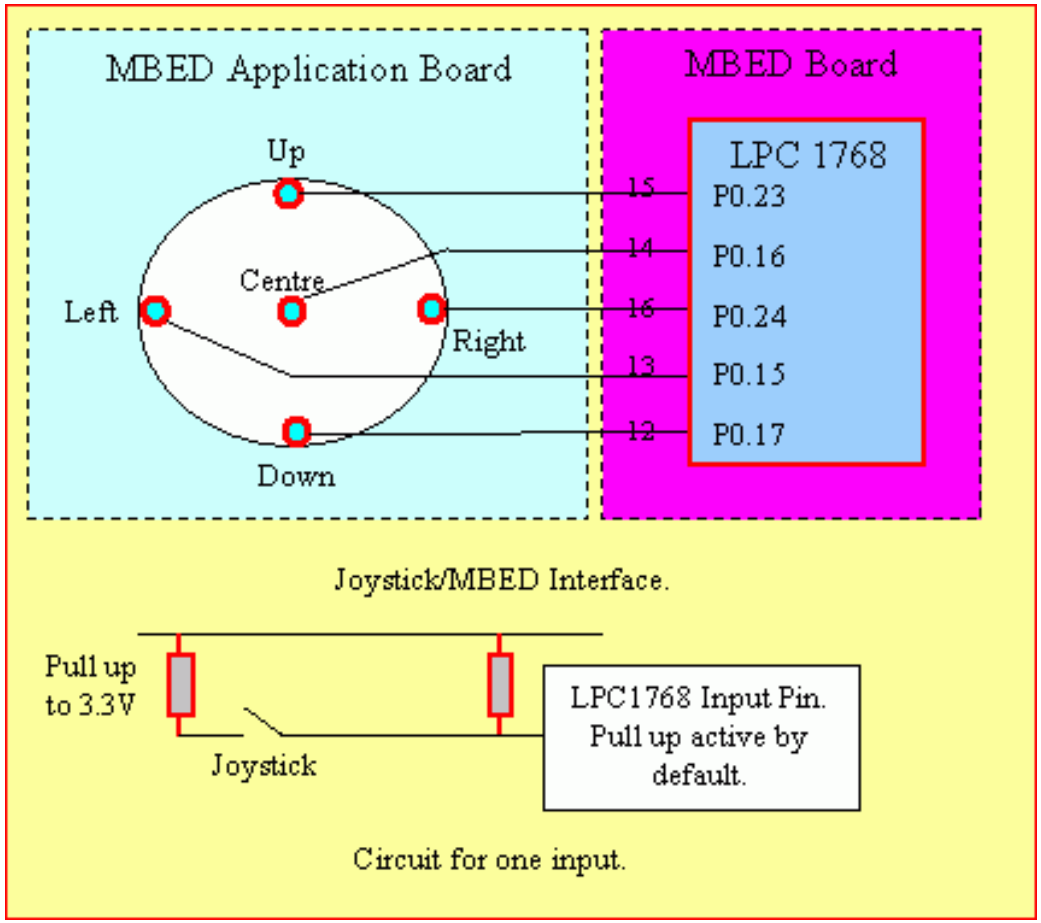
```
#include "mbed.h"
DigitalOut redled(p5);
DigitalOut greenled(p6);
DigitalIn  switchinput(p7);
int main() {
    while(1) {

      if (switchinput==1) {
        greenled = 0;    //green led is off
        redled = 1;      // flash red led
        wait(0.2);
        redled = 0;
        wait(0.2);
      }
      else if (switchinput==0) {
        redled = 0;      //red led is off
        greenled = 1;    // flash green led
        wait(0.2);
        greenled = 0;
        wait(0.2);
      }
    }
}
```

# Digital switch exercises

- Exercise 3: Create a 1000 Hz digital square wave output which can be analysed on an oscilloscope.
- Exercise 4: Using a digital input switch, create a square wave output that doubles in frequency when a digital input switch is switched on.
- Exercise 5: Create a system which counts the number of times a digital switch is pressed or changed, and lights an LED when 10 instances have been counted.

# Joystick interface to the MBED application board

- **Exercise 6:** Create a new program for the LEDs and joystick project, using the MBED application board. Modify the default main.cpp code as shown.



# Joystick interface to the MBED application board

```
#include "mbed.h"

DigitalOut myled1(LED1);
DigitalOut myled2(LED2);
DigitalOut myled3(LED3);
DigitalOut myled4(LED4);
DigitalIn  joyctr(p14);            // configure joystick centre as DigitalIn
DigitalIn  joyup(p15);             // configure joystick up as DigitalIn
DigitalIn  joydwn(p12);            // configure joystick down as
DigitalIn
DigitalIn  joylft(p13);            // configure joystick left as DigitalIn
DigitalIn  joyrgt(p16);            // configure joystick right as DigitalIn
BusIn joy(p15,p12,p13,p16);        // Combine bus in
BusOut leds(LED1,LED2,LED3,LED4);  // Combine bus out

int main() {
   while(1) {
      leds = 0;
      if (joyup) {
         myled1 = 1;
         myled2 = 0;
         myled3 = 0;
         myled4 = 0;
         //leds = joy       // use bus in and bus out
         }
```

continue →

```
 else if (joyrgt) {
         myled1 = 0;
         myled2 = 1;
         myled3 = 0;
         myled4 = 0;
         //leds = joy       // use bus in and bus out
         }
else if (joydwn) {
         myled1 = 0;
         myled2 = 0;
         myled3 = 1;
         myled4 = 0;
         //leds = joy       // use bus in and bus out
         }

 else if (joylft) {
         myled1 = 0;
         myled2 = 0;
         myled3 = 0;
         myled4 = 1;
         //leds = joy       // use bus in and bus out
         }
      else if (joyctr) {
         leds = 0xf;
         }
      else {
         leds = 0;
      }
   }
}
```

# Summary

- Introduction to digital terminology
- Digital outputs on the mbed
- Using LEDs on the mbed pins
- Connecting switches to the mbed
- Implementing a digital switch input on the mbed

# Digital input and output

Train The Trainer - IAC

# Analog input and output

**Train The Trainer – IAC**

# Agenda

- Introduction to analog signals and data
- Concepts of analog-to-digital conversion
- Analog inputs on the mbed
- Reading and logging data from analog inputs
- Concepts of digital-to-analog conversion
- Analog output with the mbed
- Generating output waveforms

# Introduction to analog data

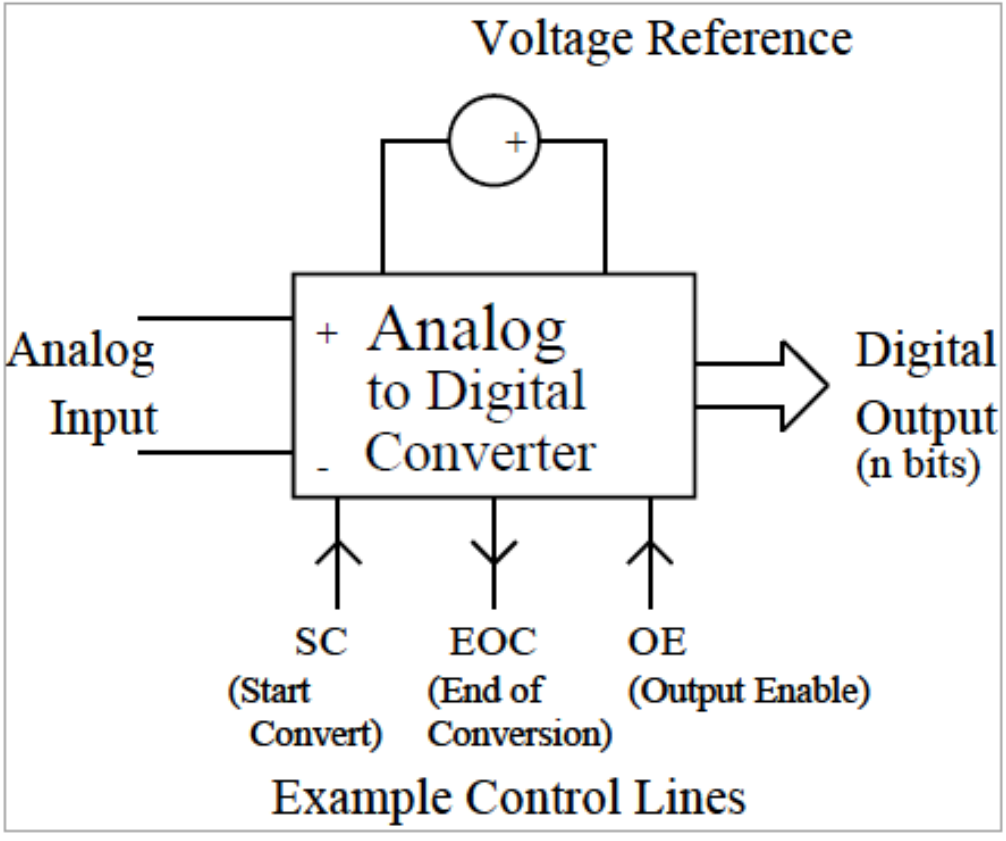
- Microcontrollers are often required to interface with analog signals
- They must be able to convert input analog signals, for example from microphone or temperature sensor, to digital data
- They must also be able to convert digital signals to analog form, for example if driving a loudspeaker or dc motor
- We will first consider conversion from analog-to-digital, before later looking at digital-to-analog conversion

# Concepts of analog-to-digital conversion

- An analog-to-digital convertor (ADC) is an electronic circuit whose digital output is proportional to its analog input
- Effectively it "measures" the input voltage, and gives a binary output number proportional to its size
- The input range of the ADC is usually determined by the value of a voltage reference

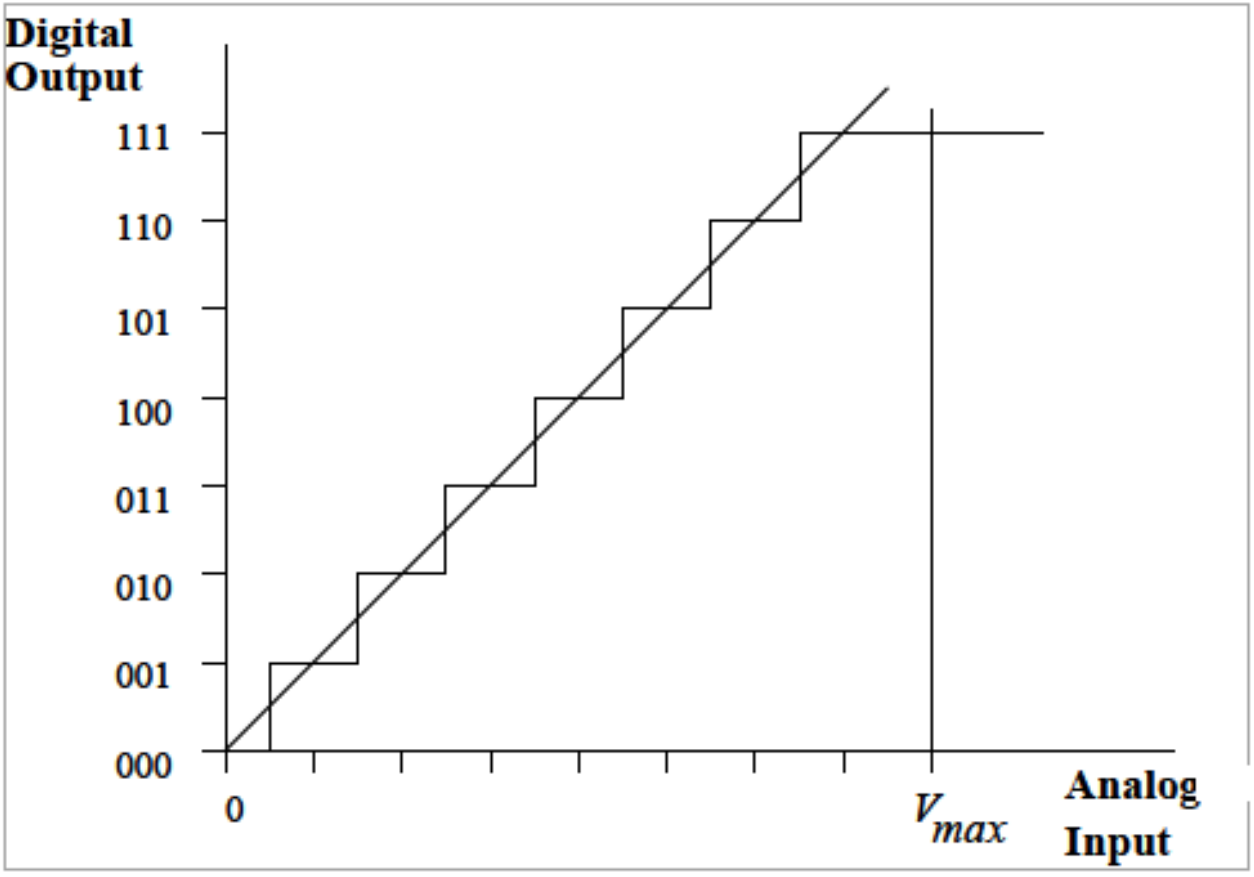
# Concepts of analog-to-digital conversion

- The conversion is started by a digital input, called here SC
- It takes finite time, and the ADC signals with the EOC line when the conversion is complete
- The resulting data can then be enabled onto a data bus using the OE line



# Concepts of analog-to-digital conversion

- The `staircase' visible in a 3-bit ADC



# Concepts of analog-to-digital conversion

Resolution and quantisation

- By converting an analog signal to digital, we are effectively approximating it, as any one digital output value has to represent a very small range of analog input voltages, i.e. the width of any of the steps on the “staircase” n.
- If we want to convert an analog signal that has a range 0-3.3 V to an 8-bit digital signal, then there are 256 (i.e. 28) distinct output values. Each step has a width of 3.3/256 = 12.89 mV, and the worst case quantisation error is 6.45mV.
- The mbed uses a 12-bit ADC. This leads to a step width of 3.3/212, or 0.8 mV; the worst case quantisation error is therefore 0.4 mV.

# Concepts of analog-to-digital conversion

Sampling frequency

- When converting an analog signal to digital, we repeatedly take a ‘sample’ and quantise this to the accuracy defined by the resolution of our ADC.

- The more samples taken, the more accurate the digital data will be. Samples are normally taken at fixed periods (i.e., every 0.2ms) and define the rate of sampling by the sampling frequency (the number of samples taken per second).

# Concepts of analog-to-digital conversion

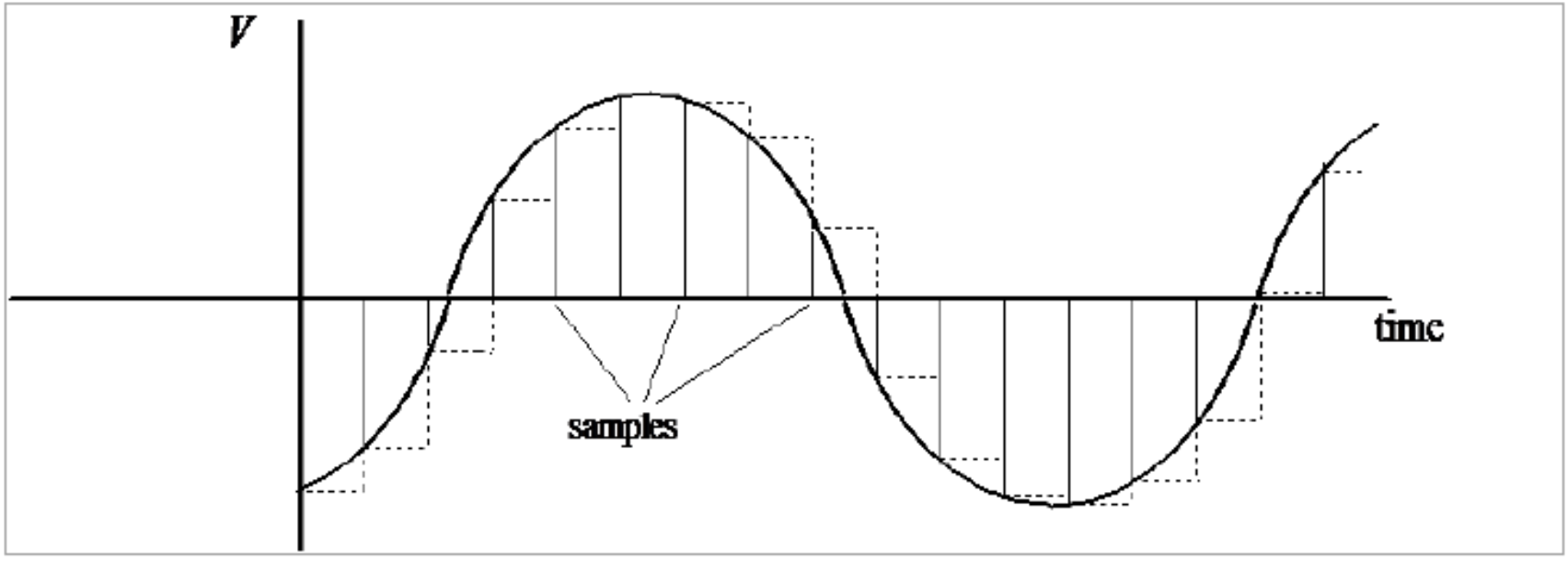
Sampling frequency

- The sample frequency needs to be chosen with respect to the rate of which the sampled data is changing. If the sample frequency is too low then rapid changes in the analog signal may not be obvious in the resulting digital data.

- For this reason the Nyquist sampling criterion states that the sampling frequency must be at least double that of the highest frequency of interest.

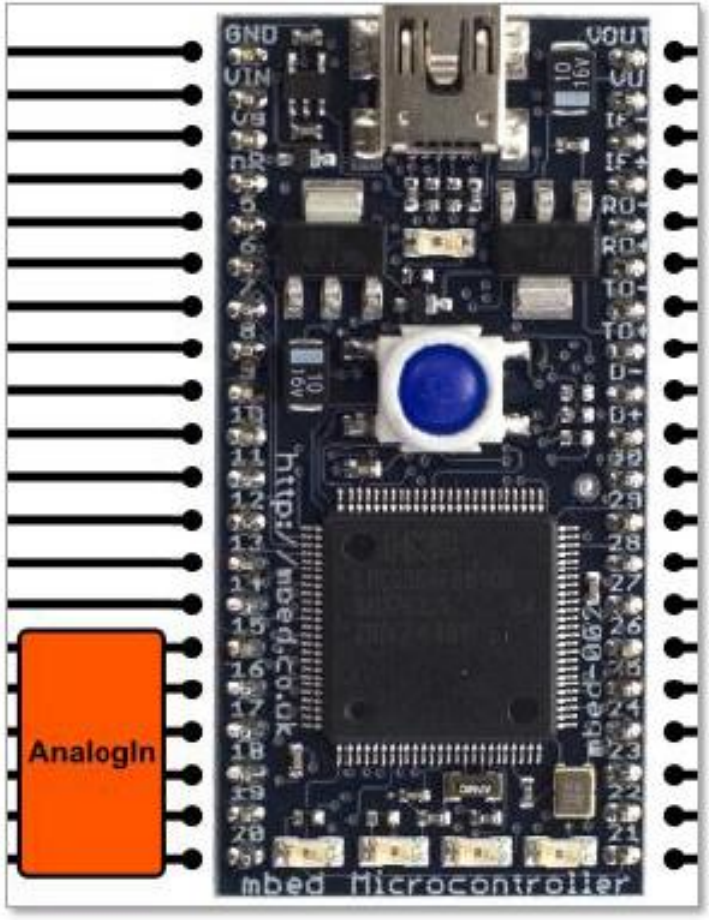
# Concepts of analog-to-digital conversion

- Digital sampling of an analog signal



# Analog inputs on the mbed

- The mbed has up to six analog inputs, on pins 15 to 20



# Analog-to-digital conversion on the mbed

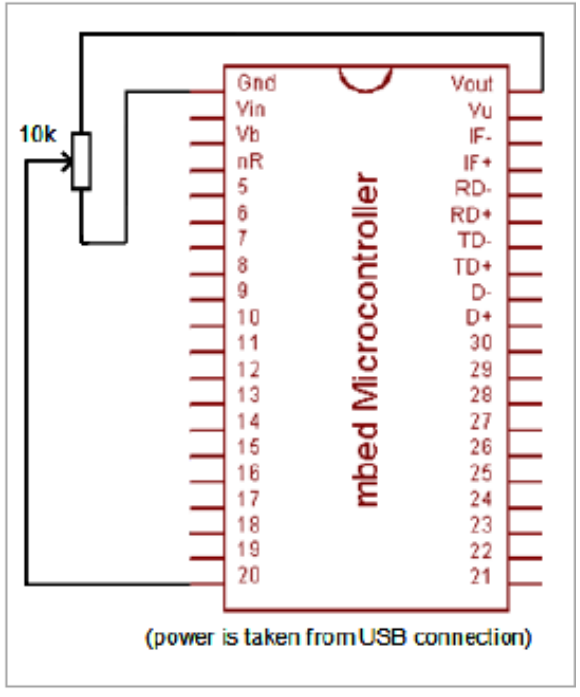
- The library functions are shown in the table below

| AnalogIn | An analog input, used for reading the voltage on a pin |
|---|---|
| **Functions** | **Usage** |
| AnalogIn | Create an AnalogIn, connected to the specified pin |
| read | Read the input voltage, represented as a float in the range [0.0, 1.0] |
| read_u16 | Read the input voltage, represented as an unsigned short in the range [0x0, 0xFFFF] |
| operator float | operator float An operator shorthand for read() |

# Reading and logging data from analog inputs

- Exercise 1: Attach a potentiometer output to mbed pin 20. (**Note:** pin 20 is connected to potentiometer Pot 2 of the **application board**)
  - Start a new mbed project and enter the code below.
  - This code will continuously display the analog input value when used with a host PC terminal application.

```
//Reads input through the ADC, and transfers to PC terminal
#include "mbed.h"

Serial pc(USBTX, USBRX);
AnalogIn Ain(p20);
float ADCdata;

int main() {
    pc.printf("ADC Data Values... \n\r");
    while (1) {
        ADCdata=Ain;
        pc.printf("%f \n\r",ADCdata);
        wait (0.5);
    }
}
```



# Reading data from analog inputs

- Exercise 2: Using the four onboard mbed LEDs, write a program that will use a potentiometer input on pin 20 to continuously control how many LEDs are on. Use the following chart to define the LED control:

| Analog input value x | LED1 | LED2 | LED3 | LED4 |
|---|---|---|---|---|
| x <= 0.2 | 0 | 0 | 0 | 0 |
| 0.2 < x <= 0.4 | 1 | 0 | 0 | 0 |
| 0.4 < x <= 0.6 | 1 | 1 | 0 | 0 |
| 0.6 < x <= 0.8 | 1 | 1 | 1 | 0 |
| 0.8 < x <= 1.0 | 1 | 1 | 1 | 1 |

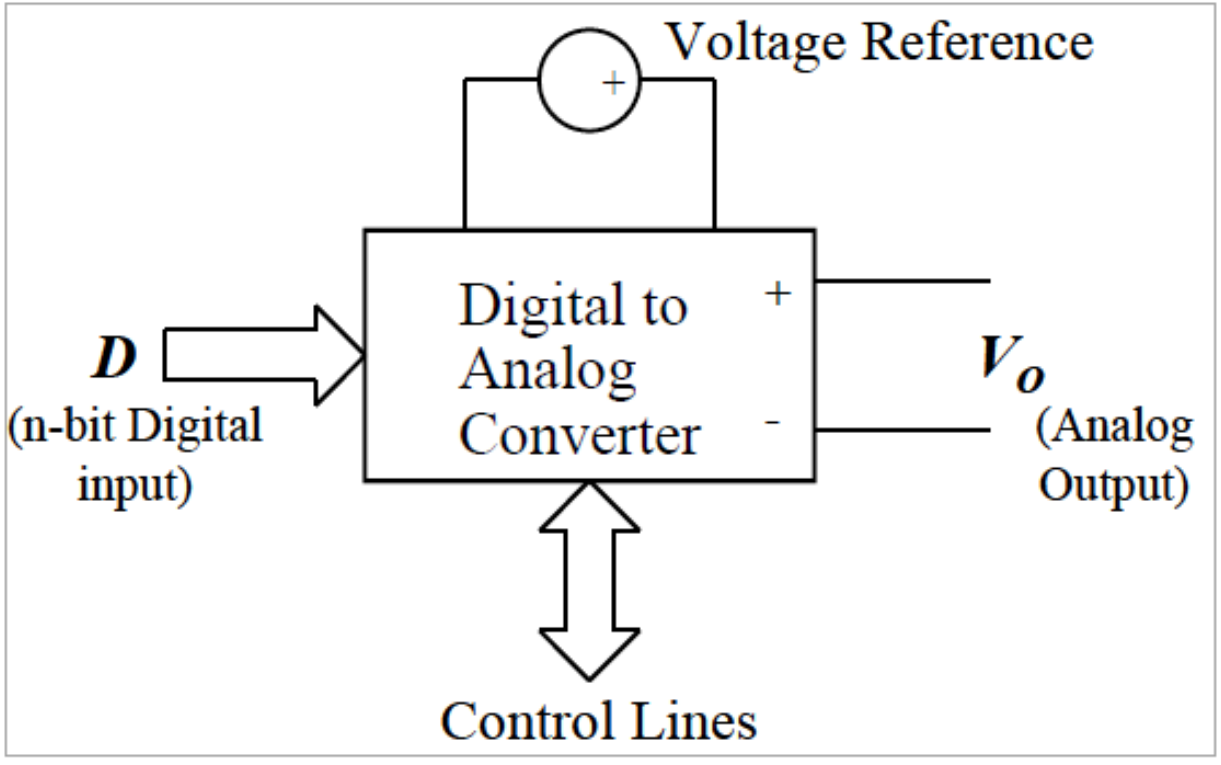
# Concepts of digital-to-analog conversion

- We can represent the digital-to-analog convertor (DAC) as a block diagram with a digital input, D, and an analog output, *vo*
- The output range of the DAC, vr , is the difference between the maximum and minimum output voltages, i.e.

  $$vr = vmax - vmin$$

- The particular output range is usually defined by a fixed voltage reference supplied to the DAC
- Digital control lines allow a microcontroller to setup and communicate with the DAC

# Concepts of digital-to-analog conversion




- For each digital value input to the DAC, there is a corresponding analog output value given by

$$v_0 = \frac{D}{2^n} v_r$$

# Concepts of digital-to-analog conversion

- The mbed's LPC1768 chip has a 10-bit DAC (i.e. n=10)
- The mbed uses its own 3.3 V power supply as voltage reference
- There will therefore be 2n steps in the mbed DAC output characteristic, i.e. 1024
- The step size, or resolution, is therefore be 3.3/1024, i.e. 3.2mV per bit

# Digital-to-analog conversion on the mbed

- The library functions are shown in the table below

| AnalogOut | An analog output, used for setting the voltage on a pin |
|---|---|
| **Functions** | **Usage** |
| AnalogOut | Create an AnalogOut connected to the specified pin |
| write | Set the output voltage, specified as a percentage (float) |
| write_u16 | Set the output voltage, represented as an unsigned short in the range [0x0, 0xFFFF] |
| read | Return the current output voltage setting, measured as a percentage (float) |
| operator= | An operator shorthand for write() |
| operator float() | An operator shorthand for read() |

# Analog output with the mbed

- The mbed analog output on pin 18 is configured by the following declaration:

```
AnalogOut Aout(p18);
```

- By default, the analog object takes a floating point number between 0.0 and 1.0 and outputs this to pin 18
- The actual output voltage on pin 18 is between 0V and 3.3V, so the floating point number that is output as a voltage is scaled by a factor of 3.3

# Analog output with the mbed

- Exercise 3: compile the program shown below and, using an oscilloscope, familiarise yourself with the analog output.

```
#include "mbed.h"
int main() {
AnalogOut Aout(p18);
    while(1) {
    Aout=0.25;                  // 0.25*3.3V = 0.825V
    wait(1);
    Aout=0.5;                   // 0.5*3.3V = 1.65V
    wait(1);
    Aout=0.75;                  // 0.75*3.3V = 2.475V
    wait(1);
    }
}
```

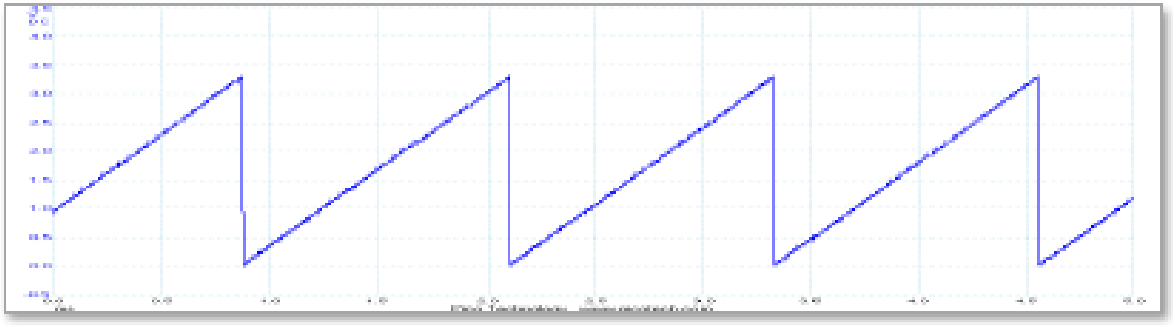
# Analog output with the mbed

- Exercise 4: Now make a sawtooth wave and view it on an oscilloscope.
- Create a new program and enter the following code

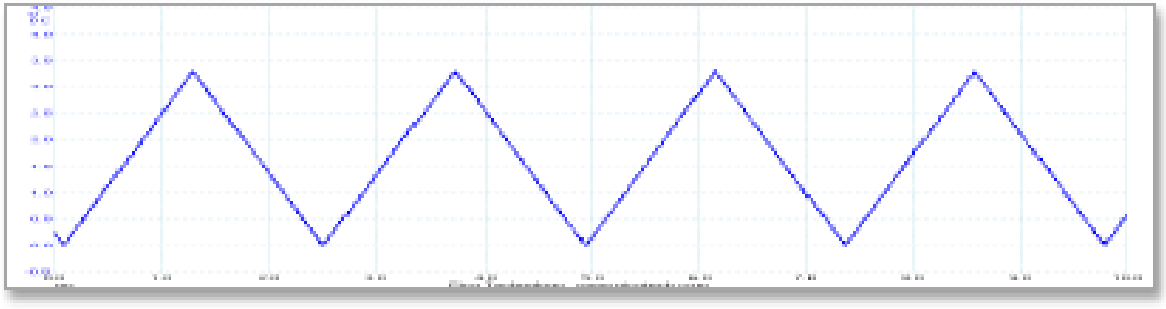
```
//Sawtooth waveform on DAC output to view on oscilloscope
#include "mbed.h"
AnalogOut Aout(p18);
float i;
int main() {
    while(1)
    {
    for (i=0;i<1;i=i+0.1)
      {
      Aout=i;
      wait(0.001);
      }
    }
}
```

# Analog output with the mbed

- Exercise 5: Modify your code to create a smoother sawtooth wave, by implementing finer steps in the for loop:

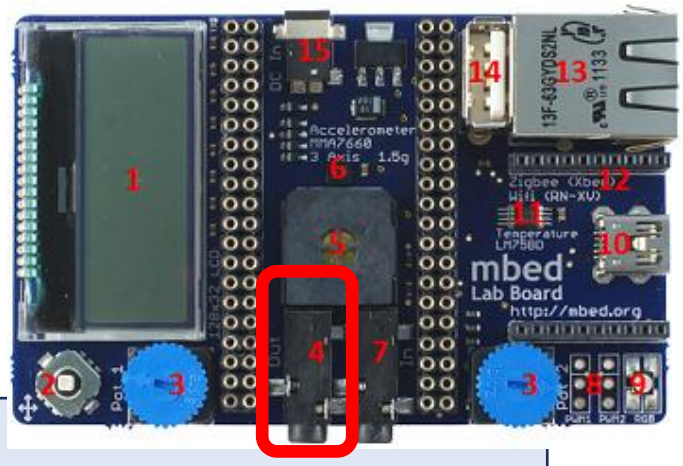


- Exercise 6: Create a smooth triangular wave by implementing a second loop to count down also:



# Exercise using MBED application board

- **Exercise 7 :** Create a new program for the AnalogIn and AnalogOut project, using the MBED application board. Modify the default main.cpp code as shown. Plug in a pair of earphones to 3.5mm Audio jack (Analog Out)




```
#include "mbed.h"

AnalogOut Aout(p18);  // Connected to Audio jack of Application Board (Analog Out)
AnalogIn pot1(p19);     // Connected to Pot 1 of Application Board (Analog in)

int main()
{
   while(1) {
      for(float i=0.0; i<1.0; i+=0.1) {
         Aout = i;
         wait(0.00001+(0.0001*pot1.read()));
      }
   }
}
```

```
#include "mbed.h"

AnalogOut Aout(p18);  // Connected to Audio jack of Application Board (Analog Out)
AnalogIn pot1(p19);     // Connected to Pot 1 of Application Board (Analog in)

int main()
{
   while(1) {
      for(float i=0.0; i<1.0; i+=0.1) {
         Aout = i;
         wait(0.00001+(0.0001*pot1.read()));
      }
   }
}
```

# Summary

- Introduction to analog signals and data
- Concepts of analog-to-digital conversion
- Analog inputs on the mbed
- Reading and logging data from analog inputs
- Concepts of digital-to-analog conversion
- Analog output with the mbed
- Generating output waveforms

# Analog input and output

Train The Trainer - IAC

# Pulse Width Modulation

**Train The Trainer – IAC**

# Agenda

- The concept of pulse width modulation (PWM)
- Applications using pulse width modulation
- Evaluating pulse width modulation on the mbed
- Controlling LED brightness with PWM
- Controlling servo position with PWM
- Outputting to a piezo buzzer

# Introducing pulse width modulation (PWM)

What is Pulse Width Modulation?

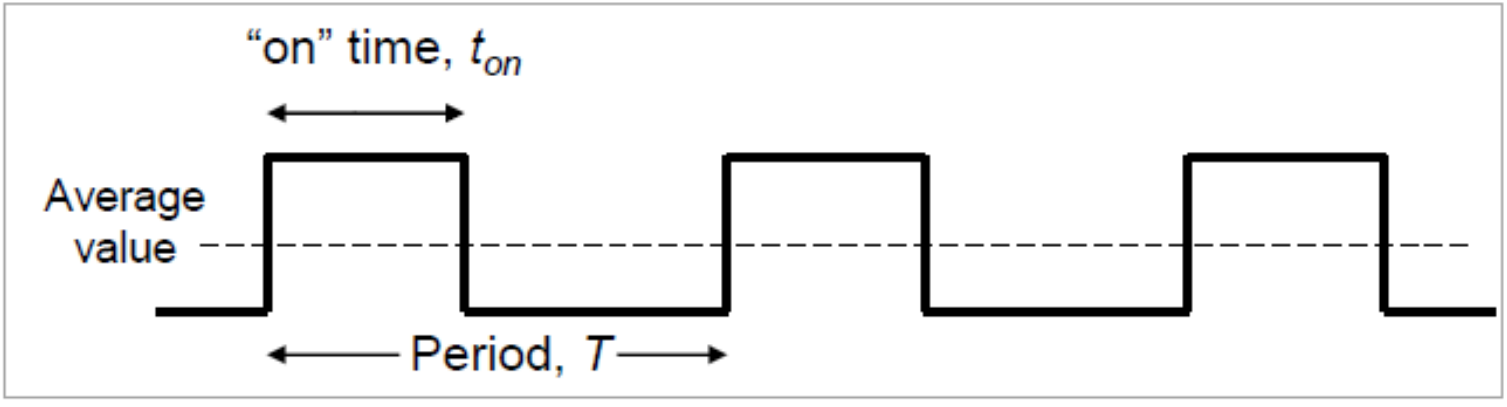
- Pulse width modulation (PWM) is a simple method of using a rectangular digital waveform to control an analog variable

- PWM control is used in a variety of applications, ranging from communications to automatic control

# Introducing pulse width modulation (PWM)

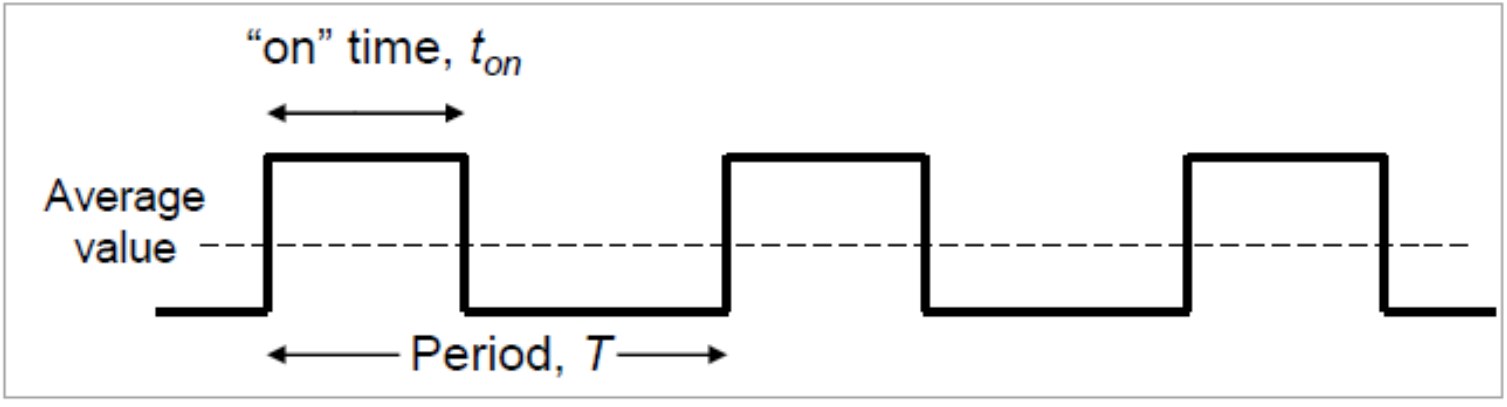
- The period is normally kept constant, and the pulse width, or “on” time is varied
- The duty cycle is the proportion of time that the pulse is ‘on’ or ‘high’, and is expressed as a percentage:

*duty cycle = 100% * (pulse on time) / (pulse period)*



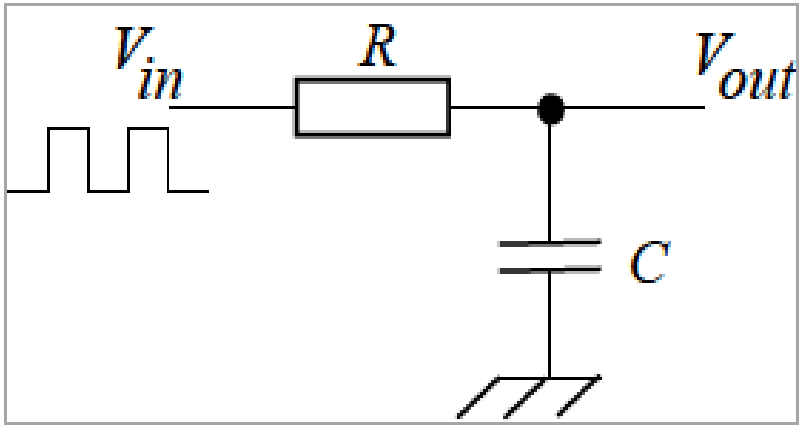
# Introducing pulse width modulation (PWM)

- Whatever duty cycle a PWM stream has, there is an average value, as indicated by the dotted line
- If the on time is small, the average value is low; if the on time is large, the average value is high
- By controlling the duty cycle, we control this average value



# Introducing pulse width modulation (PWM)

- The average value can be extracted from the PWM stream with a low-pass filter
- In this case, and as long as PWM frequency and values of R and C are appropriately chosen, Vout becomes an analog output
- In practice, this sort of filtering is not always required; many physical systems have response characteristics which, in reality, act like low pass filters



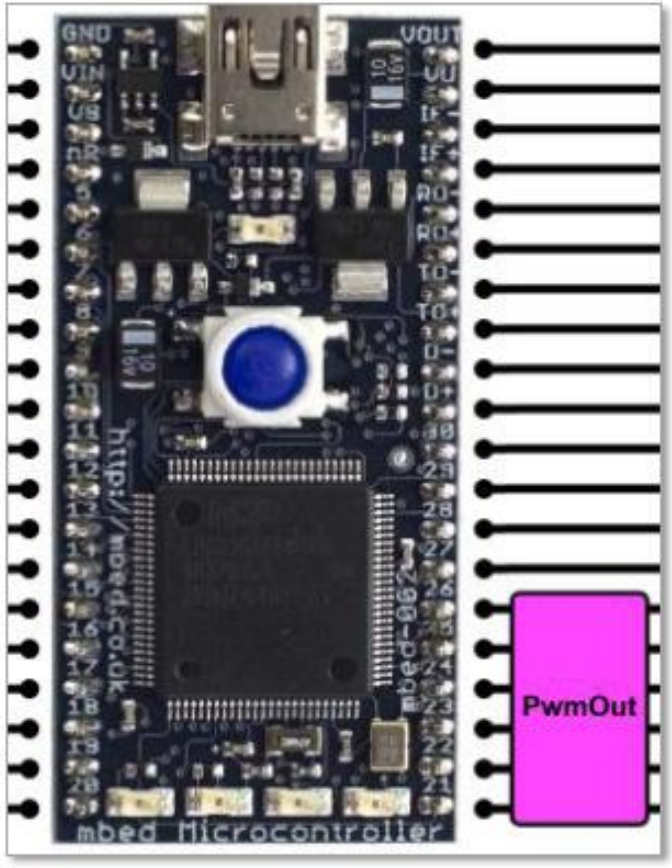
# Applications using pulse width modulation

- Devices used in robotics
  - DC motors
  - Servos
  - Solenoids
  - Closed loop control systems
  - Communications and pulse code modulation

- Benefits include
  - Microprocessor control
  - Efficient use of power
  - Tolerance to analog noise
  - Not susceptible to component drift

# Pulse width modulation on the mbed

- The PwmOut interface is used to control the frequency and mark-space ratio of a digital pulse train
- The mbed has up to six PWM outputs, on pins 21 to 26, although these PwmOuts all share the same period timer



# Pulse width modulation on the mbed

- The library functions are shown in the table below

| PwmOut | A pulse-width modulation digital output |
|---|---|
| **Functions** | **Usage** |
| PwmOut | Create a PwmOut connected to the specified pin |
| write | Set the output duty-cycle, specified as a normalised float (0.0 – 1.0) |
| read | Return the current output duty-cycle setting, measured as a normalised float (0.0 – 1.0) |
| period<br>period_ms<br>period_us | Set the PWM period, specified in seconds (float), milli-seconds (int) or micro-seconds (int), keeping the duty cycle the same. |
| pulsewidth<br>pulsewidth_ms<br>pulsewidth_us | Set the PWM pulsewidth, specified in seconds (float), milli-seconds (int) or micro-seconds (int), keeping the period the same. |
| operator= | A operator shorthand for write() |
| operator float() | An operator shorthand for read() |

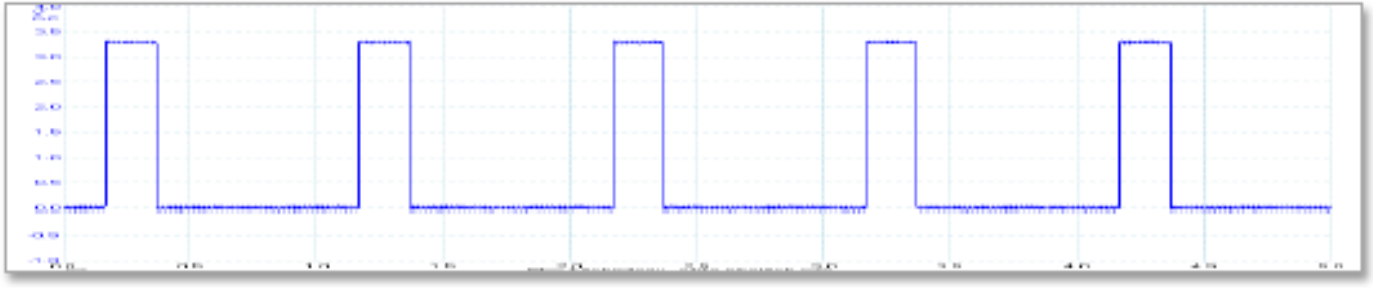
# Evaluating pulse width modulation on mbed

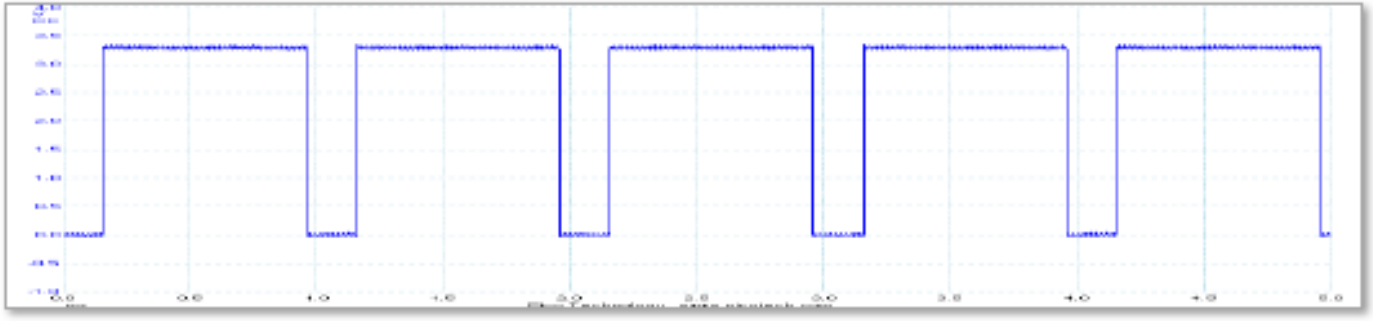
- Exercise 1: Create a PWM signal which we can see on an oscilloscope. The following code will generate a 100 Hz pulse with 50% duty cycle

```
#include "mbed.h"
PwmOut PWM1(p21);
int main() {
    PWM1.period(0.010);          // set PWM period to 10 ms
    PWM1=0.5;                    // set duty cycle to 50%
}
```

# Evaluating pulse width modulation on mbed

- Exercise 2: Change the duty cycle to some different values, say 0.2 (20%) and 0.8 (80%) and check the correct display is seen on the ‘scope, as shown below





# Controlling LED brightness with PWM

- Exercise 3: This example code uses a pulse width modulation signal to increase and decrease the brightness of the onboard LED
- The program requires the use of a host terminal application to communicate the brightness value to the mbed, in this example by using the ‘u’ and ‘d’ keys

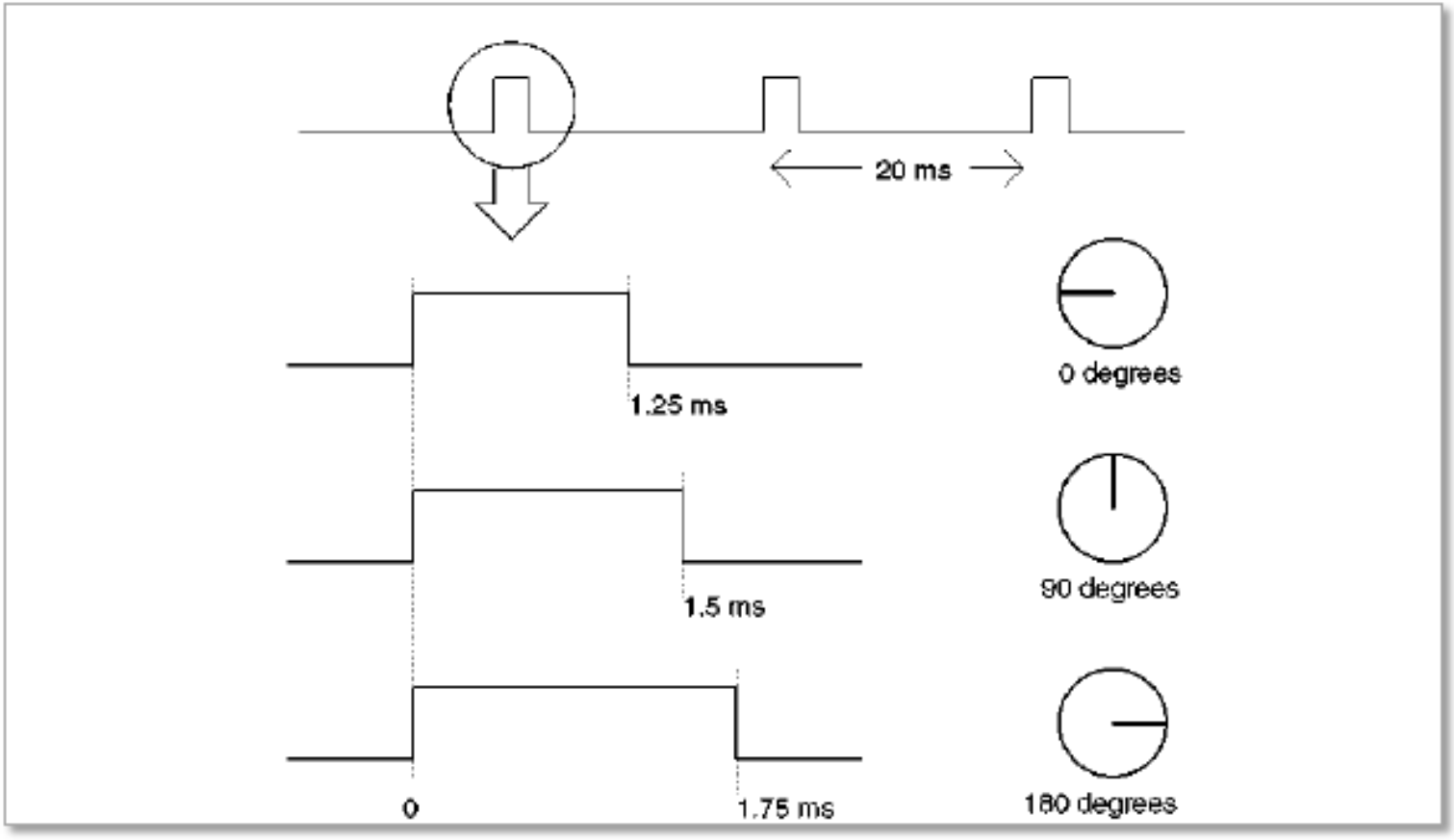
```
// host terminal LED dimmer control

#include "mbed.h"
Serial pc(USBTX, USBRX);            // tx, rx
PwmOut led(LED1);
float brightness=0.0;

int main() {
    pc.printf("Control of LED dimmer by host terminal\n\r");
    pc.printf("Press 'u' = brighter, 'd' = dimmer\n\r");
    while(1) {
        char c = pc.getc();
        wait(0.001);
        if((c == 'u') && (brightness < 0.1)) {
            brightness += 0.001;
            led = brightness;
        }
        if((c == 'd') && (brightness > 0.0)) {
            brightness -= 0.001;
            led = brightness;
        }
    pc.printf("%c %1.3f \n \r",c,brightness);
    }
}
```

# Controlling servo motor position with PWM

- A servo is a small rotary position control device, used for example in radio-controlled cars and aeroplanes to position controllers such as steering, elevators and rudders
- The servo shaft can be positioned to specific angular positions by sending the servo a PWM signal
- As long as the modulated signal exists on the input line, the servo will maintain the angular position of the shaft
- As the modulated signal changes, the angular position of the shaft changes

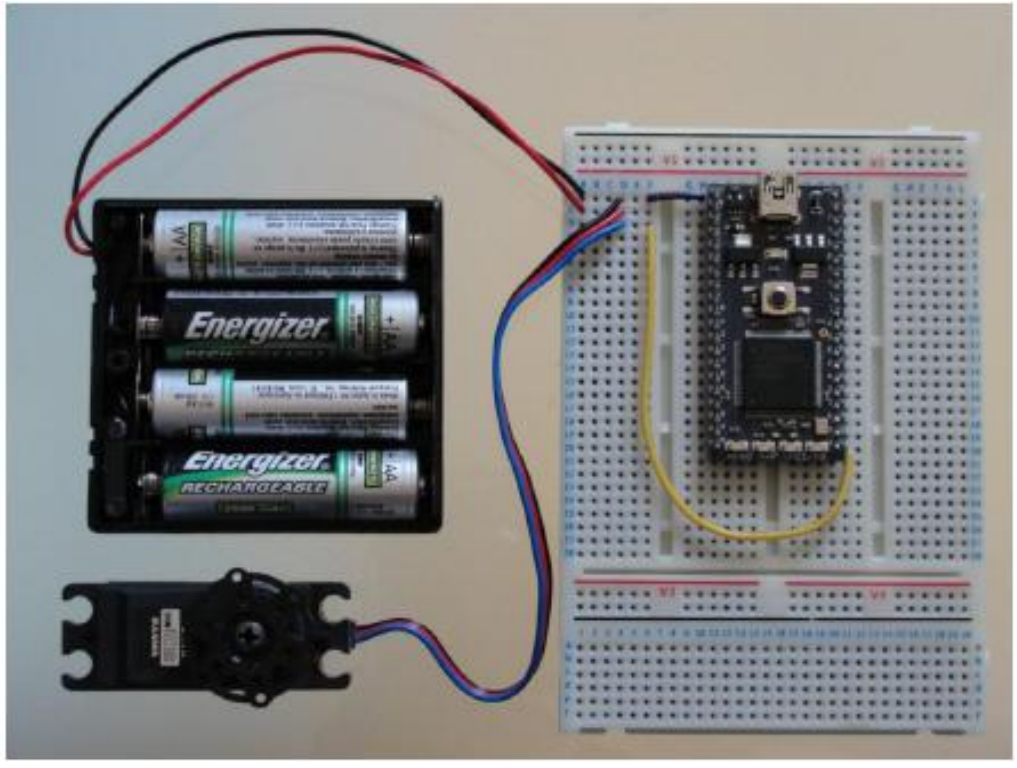
# Controlling servo motor position with PWM

- This diagram shows the PWM timing requirements to control the servo between 0 and 180 degrees



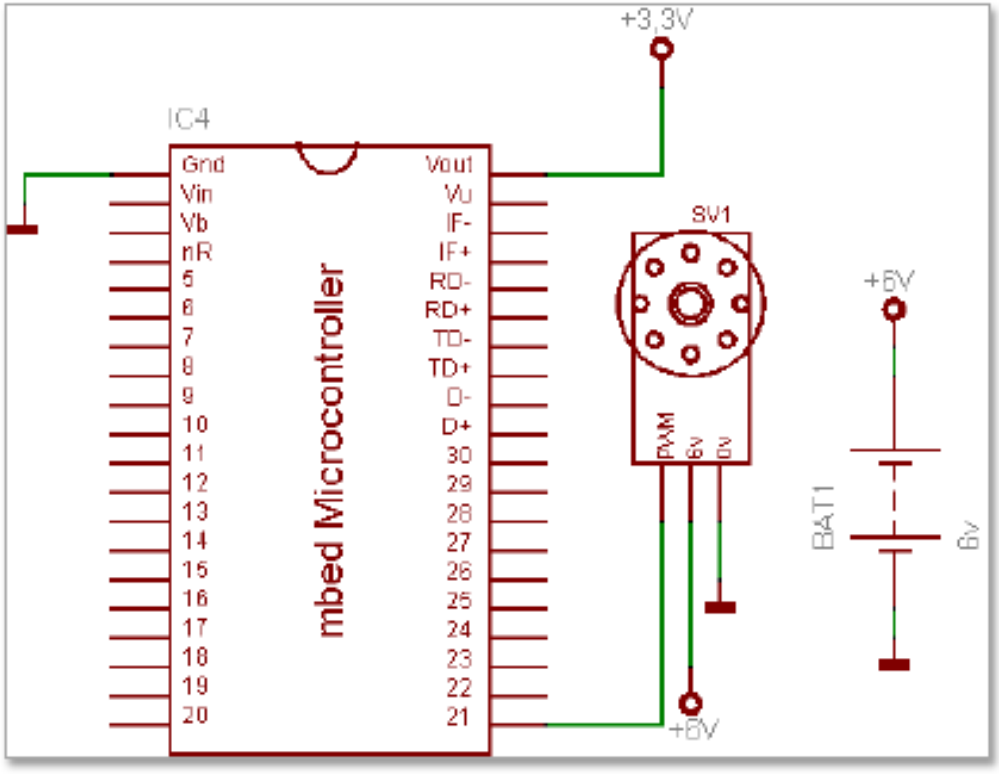
# Controlling servo motor position with PWM

- The servo requires a higher current than the USB standard can provide, and so it is essential that you power the servo using an external power supply
- A 4xAA (6V) battery pack meets the supply requirement of the servo



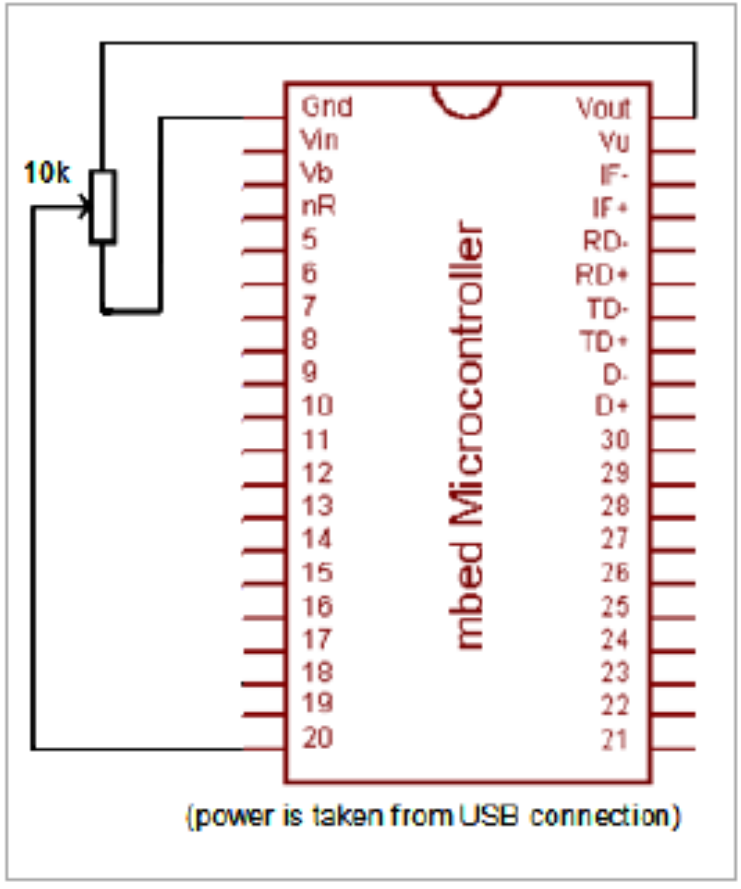
# Controlling servo motor position with PWM

- Exercise 4: Connect the servo to the mbed as indicated. Set the PWM period to 20 ms
- Try a number of different duty periods and observe the servo's motion



# Open loop control of a servo motor

- Exercise 5: Controlling servo position with the potentiometer
- Connect a servo to the mbed with a potentiometer connected to pin 20
- Write a program which allows the potentiometer to control servo position
- Scale input values so that the full range of potentiometer adjustment leads to the full range of servo position change

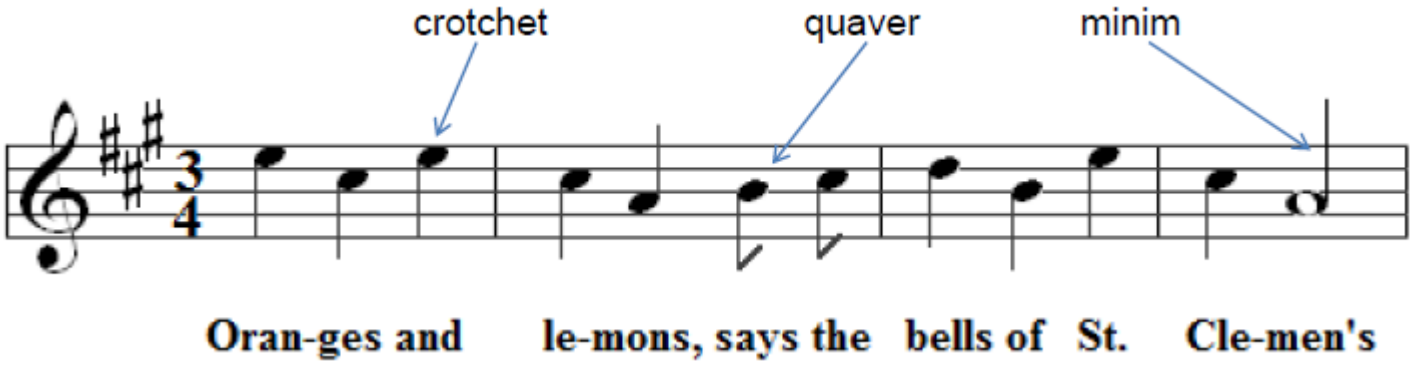


# Outputting to a piezo buzzer

- We can connect the PWM output to a piezo buzzer to make sound
- If the duty cycle is set to 50% the PWM frequency will define the pitch of the sound heard
- Exercise 6: Connect a piezo buzzer one of the PWM outputs. Set duty cycle to 50% and frequency to 500 Hz. Ensure that you can create sound with the PWM. Change the frequency of the PWM output and notice the difference in sound with changes in frequency.

# Outputting to a piezo buzzer

- Exercise 7: We are going to use the PWM to play the start of an old English folk tune ‘Oranges and Lemons’. If you’re a reader of music you will recognise this in the Figure below
- You just need to know that any note which is a minim lasts twice as long as a crotchet, which in turn lasts twice as long as a quaver



# Outputting to a piezo buzzer

- The following table converts the musical piece into frequencies and beat lengths

| Word/syllable | Musical note | Frequency (Hz) | Beats |
|---|---|---|---|
| oran | E | 659 | 1 |
| ges | C# | 554 | 1 |
| and | E | 659 | 1 |
| le | C# | 554 | 1 |
| mons | A | 440 | 1 |
| says | B | 494 | ½ |
| the | C# | 554 | ½ |
| bells | D | 587 | 1 |
| of | B | 494 | 1 |
| st | E | 659 | 1 |
| clem | C# | 554 | 1 |
| en's | A | 440 | 2 |

# Outputting to a piezo buzzer

- The following program implements the ‘Oranges and Lemons‘ musical output

```
// Oranges and Lemons program
#include "mbed.h"

PwmOut buzzer(p21);
float frequency[]={659,554,659,554,550,494,554,587,494,659,554,440};
//frequency array
float beat[]={1,1,1,1,1,0.5,0.5,1,1,1,1,2};
//beat array
int main() {
    while (1) {
        for (int i=0; i<=11; i++) {
            buzzer.period(1/(frequency[i]));    // set PWM period
            buzzer=0.5;                         // set duty cycle
            wait(0.5*beat[i]);                  // hold for beat period
        }
    }
}
```

- Experiment with adding multipliers to the frequency and beat values to change the speed and pitch of the piece.

# Summary

- The concept of pulse width modulation (PWM)
- Applications using pulse width modulation
- Evaluating pulse width modulation on the mbed
- Controlling LED brightness with PWM
- Controlling servo position with PWM
- Outputting to a piezo buzzer

# Pulse Width Modulation

Train The Trainer - IAC

# Modular design and Programming Techniques

**Train The Trainer – IAC**

# Agenda

- Developing advanced embedded systems
- Functions and subroutines
- Working with 7-segment displays
- Building mbed projects with functions
- Modular programming
- Using header files
- Creating a modular program

# Developing advanced embedded systems

- When working on large, multi-functional projects it is particularly important to consider the design and structure of the software. It is not possible to program all functionality into a single loop!

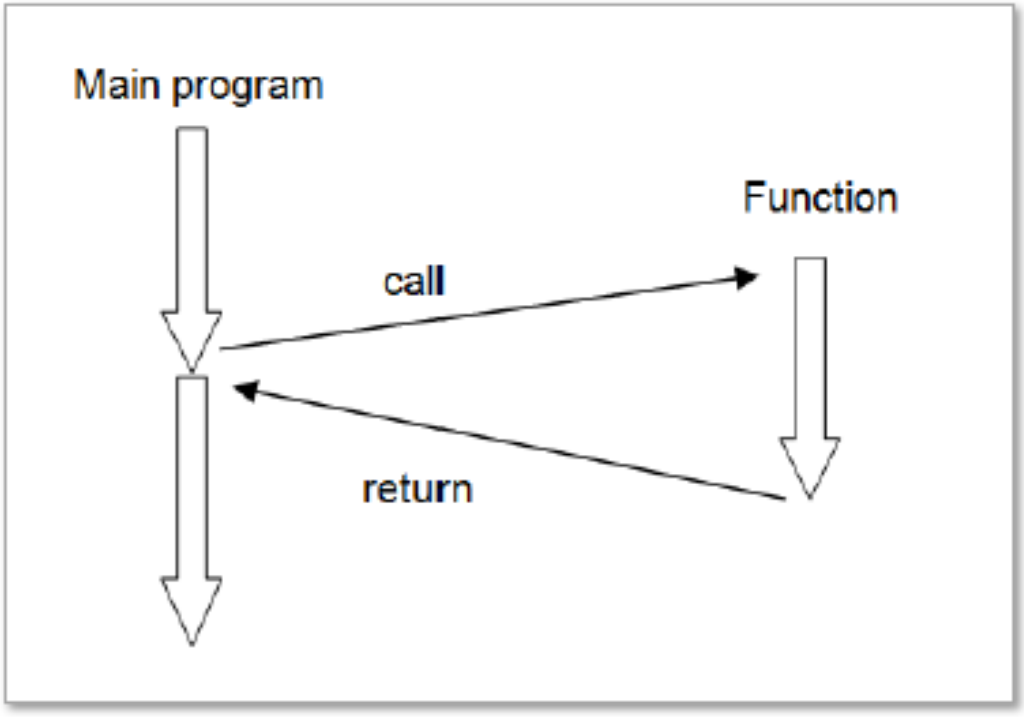
- The techniques used in large multifunctional projects also bring benefits to individuals who work on a number of similar design projects and regularly reuse code.

- Good practice for embedded system design includes:
  - Code that is readable, structured and documented
  - Code that can be tested in a modular form
  - Reuses existing code utilities to keep development time short
  - Support multiple engineers working on a single project
  - Allow for future upgrades to be implemented

- There are a number of C/C++ programming techniques which enable these design requirements to be considered…

# Functions and subroutines

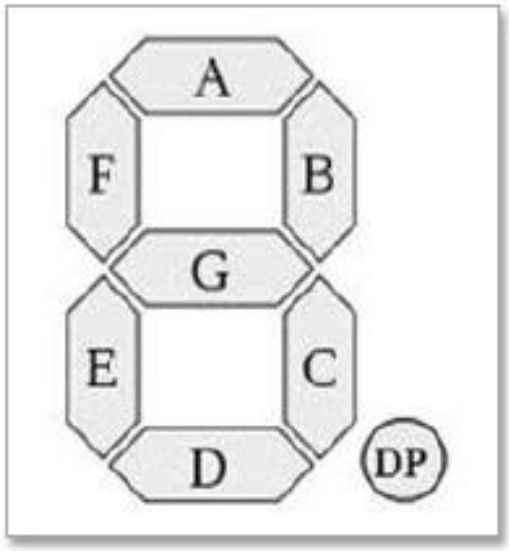
- A function (sometimes called a subroutine) is a portion of code within a larger program
- A function performs a specific task and is relatively independent of the main code



# Functions and subroutines

- To demonstrate the use of functions, we will look at examples using a standard 7-segment LED display.
- Note that the 8-bit byte for controlling the individual LED's on/off state is configured as:

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



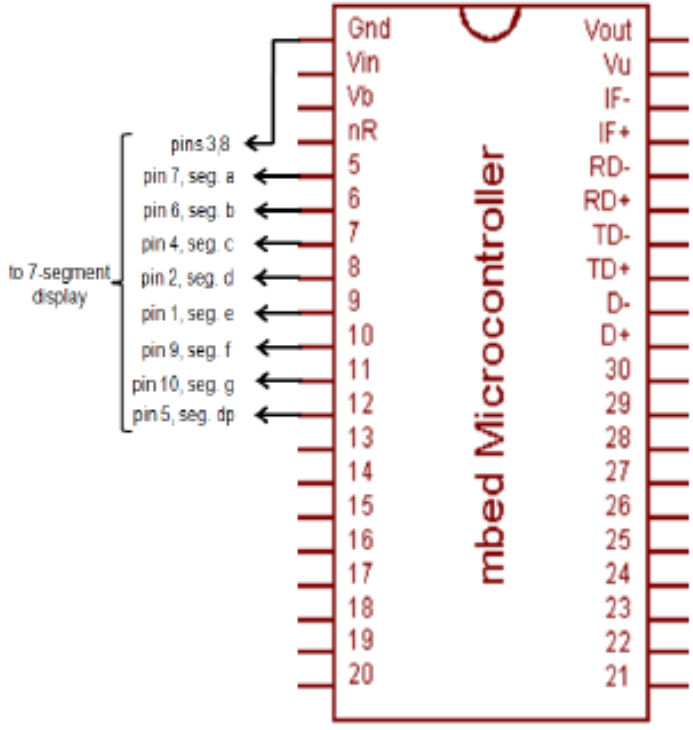

| A | 0x00 | 0x01 | 0x02 | 0x03 | 0x04 | 0x05 | 0x06 | 0x07 | 0x08 | 0x09 |
|---|---|---|---|---|---|---|---|---|---|---|
| B | | | | | | | | | | |
| (MSB) | 0011 | 0000 | 0101 | 0100 | 0110 | 0110 | 0111 | 0000 | 0111 | 0110 |
| (LSB) | 1111 | 0110 | 1011 | 1111 | 0110 | 1101 | 1101 | 0111 | 1111 | 1111 |
| 7-seg |  |  |  |  |  |  |  |  |  |  |

# Working with 7-segment displays

- A 7-segment display is actually just 8 LEDs in a single package. We can therefore connect each LED pin to an mbed pin to display a chosen number.

- For example, the code here uses a digital output for each LED segment and switches the correct LEDs on and off to display the number 3.

```
#include "mbed.h"

DigitalOut A(p5);
DigitalOut B(p6);
DigitalOut C(p7);
DigitalOut D(p8);
DigitalOut E(p9);
DigitalOut F(p10);
DigitalOut G(p11);
DigitalOut DP(p12);

int main() {
    A=1;
    B=1;
    C=1;
    D=1;
    E=0;
    F=0;
    G=1;
    DP=0;
}
```



# Working with 7-segment displays

- You may see that this code can get a little intensive; it is a very simple operation but:
  - we have defined a large number of digital outputs
  - we have to set each output to a very specific value in order to display a single number
- There is an issue with scalability here:
  - What if we want to change the number regularly? Or output a number based on the result of a calculation?
  - What if we need more digits, i.e. decimal places or numbers with tens, hundreds, thousands etc?

```
#include "mbed.h"

DigitalOut A(p5);
DigitalOut B(p6);
DigitalOut C(p7);
DigitalOut D(p8);
DigitalOut E(p9);
DigitalOut F(p10);
DigitalOut G(p11);
DigitalOut DP(p12);

int main() {
    A=1;
    B=1;
    C=1;
    D=1;
    E=0;
    F=0;
    G=1;
    DP=0;
}
```

# Working with 7-segment displays

- If we want our LED display to continuously count from 0-9, our code might look something like this →
- Exercise 1: connect a 7-segment display to the mbed and verify that the code example continuously counts from 0-9

```
// program code for Exercise 1
#include "mbed.h"
DigitalOut A(p5);       // segment A
DigitalOut B(p6);       // segment B
DigitalOut C(p7);       // segment C
DigitalOut D(p8);       // segment D
DigitalOut E(p9);       // segment E
DigitalOut F(p10);      // segment F
DigitalOut G(p11);      // segment G
DigitalOut DP(p12);     // segment DP

int main() {
    while (1) {                                     // infinite loop
        A=1; B=1; C=1; D=1; E=1; F=1; G=0; DP=0;  // set LEDs '0'
        wait(0.2);
        A=0; B=1; C=1; D=0; E=0; F=0; G=0; DP=0;  // set LEDs '1'
        wait(0.2);
        A=1; B=1; C=0; D=1; E=1; F=0; G=1; DP=0;  // set LEDs '2'
        wait(0.2);
        A=1; B=1; C=1; D=1; E=0; F=0; G=1; DP=0;  // set LEDs '3'
        wait(0.2);
        A=0; B=1; C=1; D=0; E=0; F=1; G=1; DP=0;  // set LEDs '4'
        wait(0.2);
        A=1; B=0; C=1; D=1; E=0; F=1; G=1; DP=0;  // set LEDs '5'
        wait(0.2);
        A=1; B=0; C=1; D=1; E=1; F=1; G=1; DP=0;  // set LEDs '6'
        wait(0.2);
        A=1; B=1; C=1; D=0; E=0; F=0; G=0; DP=0;  // set LEDs '7'
        wait(0.2);
        A=1; B=1; C=1; D=1; E=1; F=1; G=1; DP=0;  // set LEDs '8'
        wait(0.2);
        A=1; B=1; C=1; D=1; E=0; F=1; G=1; DP=0;  // set LEDs '9'
        wait(0.2);
    }
}
```

# Working with 7-segment displays

- Before moving on to functions, we can simplify our code using a BusOut object.
- The BusOut object allows a number of digital outputs to be configured and manipulated together.
- We can therefore define a BusOut object for our 7-segment display and send the desired data byte value in order to display a chosen number.
- Exercise 2: Verify that the code here performs the same functionality as that in the previous exercise

```
// program code for Exercise 2
#include "mbed.h"

BusOut Seg1(p5,p6,p7,p8,p9,p10,p11,p12); // ABCDEFGDP

int main() {
    while (1) {          // infinite loop
        Seg1=0x3F;       // 00111111 binary LEDs to '0'
        wait(0.2);
        Seg1=0x06;       // 00000110 binary LEDs to '1'
        wait(0.2);
        Seg1=0x5B;       // 01011011 binary LEDs to '2'
        wait(0.2);
        Seg1=0x4F;       // 01001111 binary LEDs to '3'
        wait(0.2);
        Seg1=0x66;       // 01100110 binary LEDs to '4'
        wait(0.2);
        Seg1=0x6D;       // 01101101 binary LEDs to '5'
        wait(0.2);
        Seg1=0x7D;       // 01111101 binary LEDs to '6'
        wait(0.2);
        Seg1=0x07;       // 00000111 binary LEDs to '7'
        wait(0.2);
        Seg1=0x7F;       // 01111111 binary LEDs to '8'
        wait(0.2);
        Seg1=0x6F;       // 01101111 binary LEDs to '9'
        wait(0.2);
    }
}
```

# Working with 7-segment displays

- There are some issues with this current method for coding the 7-segment display:
  - If we wanted to add a second 7-segment display and count from 0-99, we would clearly have a problem – we'd need to write lots of extra code!
  - There is also very little flexibility with this coding method – if we want to change the functionality slightly (for example to display every third number or to change the timing) we have to make quite a lot of changes
  - There is a better way to program this type of functionality; using functions

# C function syntax

- The correct C function syntax is as follows:

```
Return_type function_name (variable_type_1 variable_name_1, variable_type_2 variable_name_2,…)
{
... C code here
... C code here
}
```

- As with variables, all functions must be declared at the start of a program
- The declaration statements for functions are called prototypes
- The correct format for a function prototype is the same as in the function itself, as follows:

```
Return_type function_name (variable_type_1 variable_name_1, variable_type_2 variable_name_2,…)
```

# Designing a C function

- It is beneficial for us to design a C function that inputs a count variable and returns the 8-bit value for the corresponding 7-segment display, for example:

```
char SegConvert(char SegValue) {          // function 'SegConvert'
    char SegByte=0x00;
    switch (SegValue) {                  //DPGFEDCBA
        case 0 : SegByte = 0x3F;break;   // 00111111 binary
        case 1 : SegByte = 0x06;break;   // 00000110 binary
        case 2 : SegByte = 0x5B;break;   // 01011011 binary
        case 3 : SegByte = 0x4F;break;   // 01001111 binary
        case 4 : SegByte = 0x66;break;   // 01100110 binary
        case 5 : SegByte = 0x6D;break;   // 01101101 binary
        case 6 : SegByte = 0x7D;break;   // 01111101 binary
        case 7 : SegByte = 0x07;break;   // 00000111 binary
        case 8 : SegByte = 0x7F;break;   // 01111111 binary
        case 9 : SegByte = 0x6F;break;   // 01101111 binary
    }
    return SegByte;
}
```

- Note that this function uses a C ‘switch/case’ statement which performs like an ‘If’ operation with a number of possible conditions.

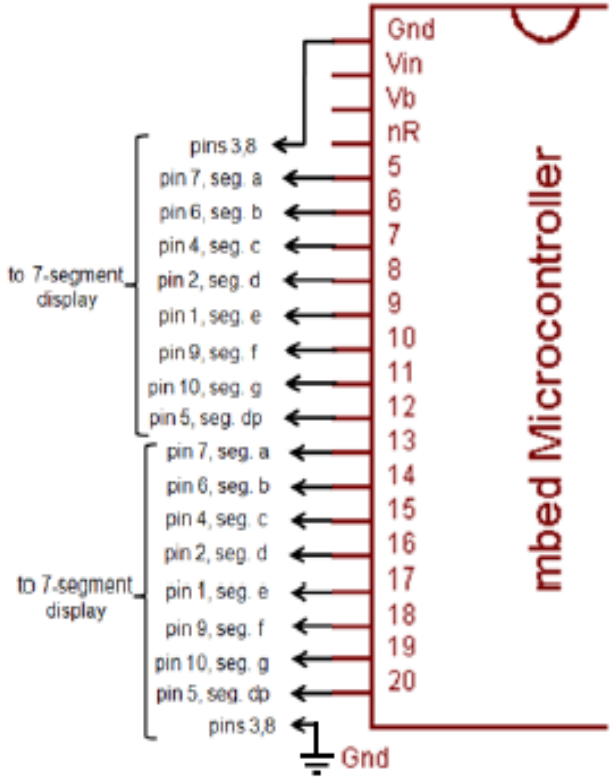
# Implementing a C function

- Exercise 3: Verify that the following program, using the SegConvert function, performs the same functionality as in Exercises 1 and 2.

```
#include "mbed.h"
BusOut Seg1(p5,p6,p7,p8,p9,p10,p11,p12);// A,B,C,D,E,F,G,DP
char SegConvert(char SegValue);          // function
prototype
int main() {                             // main program
    while (1) {                          // infinite loop
        for (char i=0;i<10;i++) {
            Seg1=SegConvert(i);
            wait(0.2);
        }
    }
}
char SegConvert(char SegValue) {    // function 'SegConvert'
    char SegByte=0x00;
    switch (SegValue) {                  //DPGFEDCBA
        case 0 : SegByte = 0x3F;break;   // 00111111 binary
        case 1 : SegByte = 0x06;break;   // 00000110 binary
        case 2 : SegByte = 0x5B;break;   // 01011011 binary
        case 3 : SegByte = 0x4F;break;   // 01001111 binary
        case 4 : SegByte = 0x66;break;   // 01100110 binary
        case 5 : SegByte = 0x6D;break;   // 01101101 binary
        case 6 : SegByte = 0x7D;break;   // 01111101 binary
        case 7 : SegByte = 0x07;break;   // 00000111 binary
        case 8 : SegByte = 0x7F;break;   // 01111111 binary
        case 9 : SegByte = 0x6F;break;   // 01101111 binary
    }
    return SegByte;
}
```

# Reusing functions to reduce programming effort

- Exercise 4: To explore the power of function reuse, add a second 7-segment display to pins 13-20. You can now update the main program code to call the SegConvert function a second time, to implement a counter which counts from 0-99.




```
// main program code for  Exercise 4

#include "mbed.h"
BusOut Seg1(p5,p6,p7,p8,p9,p10,p11,p12); // A,B,C,D,E,F,G,DP
BusOut Seg2(p13,p14,p15,p16,p17,p18,p19,p20);
char SegConvert(char SegValue);    // function prototype

int main() {                              // main program
    while (1) {                           // infinite loop
        for (char j=0;j<10;j++) {          // counter loop 1
            Seg2=SegConvert(j);           // tens column
            for (char i=0;i<10;i++) { // counter loop 2
                Seg1=SegConvert(i);  // units column
                wait(0.2);
            }
        }
    }
}

// SegConvert function here...
```

# Summary of C functions

- Functions can be used to:
  - Process and manipulate data; we can input data values to a function, which returns manipulated data back to the main program. For example, code mathematical algorithms, look up tables and data conversions, as well as control features which may operate on a number of different and parallel data streams
  - Allow clean, efficient and manageable code to be designed
  - Allow multiple engineers to develop software features independently, and hence enable the practice of modular coding
- Functions can also be reused, which means that engineers don't have to re-write common code every time they start a new project
- Notice that in every program we also have a main() function, which is where our main program code is defined

# Building complex mbed projects with functions

- Before discussing modular coding, we will design a more advanced mbed project using a number of functions
- Exercise 5: Write a program which reads a two digit number from a host terminal keyboard and outputs that number to two 7-segment displays
  - Four function prototypes are declared prior to the main program function:

```
void SegInit(void);                 // function to initialise 7-seg displays

void HostInit(void);                // function to initialise the host terminal

char GetKeyInput(void);             // function to get a keyboard input from the terminal

char SegConvert(char SegValue);     // function to convert a number to a 7-segment byte
```

  - We will also use the mbed serial USB interface to communicate with the host PC and two 7-segment displays as in the previous exercises

# Building mbed projects with functions

- Create a new project and add the following to your main.cpp file:

```
// main program code for Exercise 5
#include "mbed.h"
Serial pc(USBTX, USBRX);                          // comms to host PC
BusOut Seg1(p5,p6,p7,p8,p9,p10,p11,p12);          // A,B,C,D,E,F,G,DP
BusOut Seg2(p13,p14,p15,p16,p17,p18,p19,p20);     // A,B,C,D,E,F,G,DP

void SegInit(void);                    // function prototype
void HostInit(void);                   // function prototype
Char GetKeyInput(void);                // function prototype
char SegConvert(char SegValue);        // function prototype

char data1, data2;                     // variable declarations

int main() {                           // main program
    SegInit();                         // call function to initialise the 7-seg displays
    HostInit();                        // call function to initialise the host terminal
    while (1) {                        // infinite loop
        data2 = GetKeyInput();         // call function to get 1st key press
        Seg2=SegConvert(data2);        // call function to convert and output
        data1 = GetKeyInput();         // call function to get 2nd key press
        Seg1=SegConvert(data1);        // call function to convert and output
        pc.printf("  ");               // display spaces between 2 digit numbers
    }
}

// add function code here...
```

# Building mbed projects with functions

- The following functions need to be added after your main program code:

```
// functions for Exercise 5
void SegInit(void) {
    Seg1=SegConvert(0);         // initialise to zero
    Seg2=SegConvert(0);         // initialise to zero
}

void HostInit(void) {
    pc.printf("\n\rType two digit numbers to be displayed on the 7-seg display\n\r");
}

char GetKeyInput(void) {
    char c = pc.getc();       // get keyboard data (note numerical ascii range 0x30-0x39)
    pc.printf("%c",c);        // print ascii value to host PC terminal
    return (c&0x0F);          // return value as non-ascii (bitmask c with value 0x0F)
}

// copy SegConvert function here too...
```

- You will also need to copy in the code for the SegConvert function
- Your code should now compile and run

# Modular programming

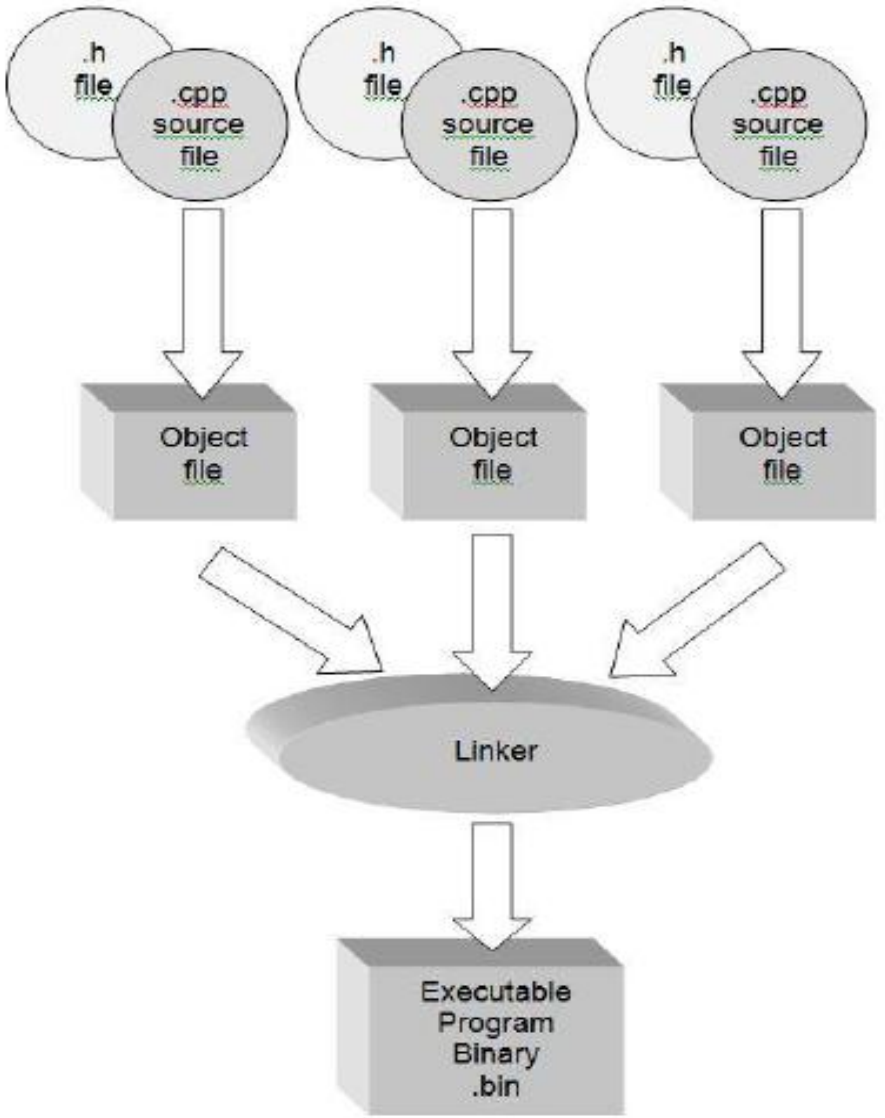
- Large projects in C and C++ need splitting into a number of different files. This approach improves readability and maintenance.
- For example:
  - The code for an embedded system might have one C file for the control of the attached peripherals and a different file for controlling the user input.
  - It doesn’t make sense to combine these two code features in the same source file.
  - If one of the peripherals is updated, only that piece of the code needs to be modified.
  - All the other source files can be carried over without change.
  - Working this way enables a team of engineers to work on an single project by each taking responsibility for a software feature.

# Modular programming

- Modular coding uses header files to join multiple files together.
- In general we use a main.cpp file to call and use functions, which are defined in feature specific .cpp files.
- Each .cpp definition file should have an associated declaration file, we call this the ‘header file’.
- Header files have a .h extension and typically include declarations only, for example compiler directives, variable declarations and function prototypes.
- A number of header files also exist from within C. These can be used for more advanced data manipulation or arithmetic. For example, math.h can be included to more easily perform trigonometric functions.

# Modular programming

- To fully understand the design approach to modular coding, it helps to understand the way programs are pre-processed, compiled and linked to create a binary execution file for the microprocessor.
- First a pre-processor looks at a particular source (.cpp) file and implements any pre-processor directives and associated header (.h) files.
  - Pre-processor directives are denoted with a '#' symbol.
- The compiler then generates an object file for the particular source code.
- Object and library files are then linked together to generate an executable binary (.bin) file.



# Modular programming

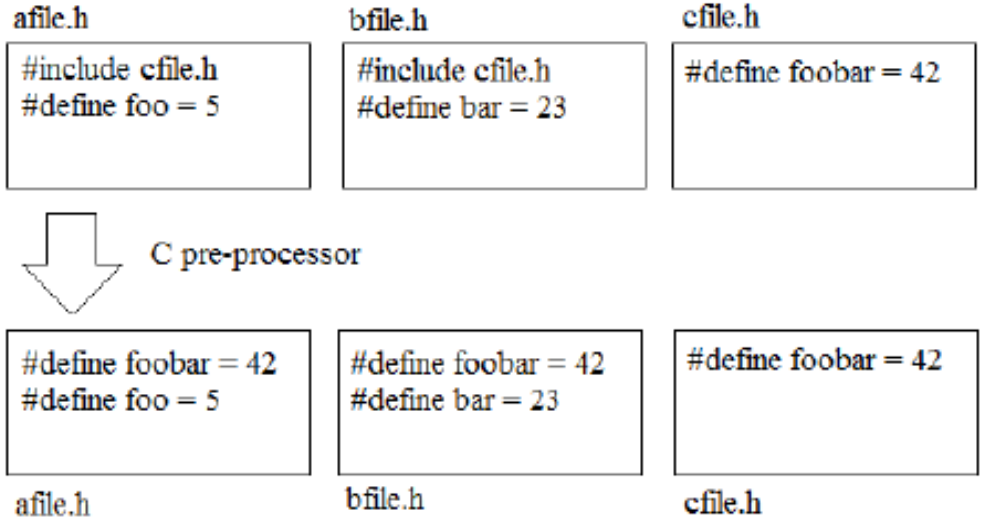
- We commonly use the #include directive to tell the pre-processor to include any code or statements contained within an external header file.
- When including predefined C header files, we must put the filename in <>, for example:

```
#include <string.h>
```

- When we include our own header files, we put the filename in quotations, for example:

```
#include "myfile.h"
```

- It is important to note that #include acts as a cut and paste feature. If you include "afile.h" and "bfile.h" where both files also include "cfile.h" You will have two copies of the contents of “cfile.h".
- The compiler will therefore see instances of multiple declarations of *foobar and highlight an error.*



# Using header files

- We can use the #ifndef pre-processor directive to ensure that header code is only ever included once by the linker.
- #ifndef means literally ‘if not defined’.
- We can define a pre-processor variable (using #define) at the start of a header file, and then only include that header code if the variable has not previously been defined.
- This way the header code is only included once for linking.
- As good practice it is therefore recommended to use the following template for header files:

```
// template for .h file
#ifndef VARIABLENAME_H          // if VARIABLENAME_H has not previously been defined
#define VARIABLENAME_H          // define it now

// header declarations here…

#endif                          // end of the if directive
```

# Using header files

- Header files usually contain
  - #include statements for built in C libraries and bespoke function libraries
  - Function prototypes
- Additionally, we should include details of any mbed Objects which are required to be manipulated from outside the source file. For example:
  - a file called functions.cpp may define and use a DigitalOut object called ‘RedLed’.
  - If we want any other source files to manipulate RedLed, we must also declare the Object in the header file using the extern type, as follows:

```
// functions.h file
#ifndef FUNCTIONS_H                    // if FUNCTIONS_H has not previously been defined
#define FUNCTIONS_H                    // define it now

extern DigitalOut RedLed;              // external object RedLed is defined in functions.cpp
// other header declarations here…

#endif                                 // end of the if directive
```

  - Note that we don’t need to define the specific mbed pins here, as these will have already been specified in the object declaration in functions.cpp

# Creating a modular program

- Exercise 6: Create the same keyboard controlled, 7-segment display project as in Exercise 5 using modular coding techniques, i.e. with multiple source files as follows:
  - main.cpp – contains the main program function
  - HostIO.cpp – contains functions and objects for host terminal control
  - SegDisplay.cpp – contains functions and objects for 7-segment display output
- We also need the following associated header files:
  - HostIO.h
  - SegDisplay.h
- Note that, by convention, the main.cpp file does not need a header file

# Creating a modular program

- Create a new project and add the required modular files
  - To add new files, right click on your project and select ‘New File…’
- The main.cpp file will hold the same main function code as before, but with #includes to the new header files

```
// main.cpp file for Exercise 6
#include "mbed.h"
#include "HostIO.h"
#include "SegDisplay.h"

char data1, data2;                      // variable declarations

int main() {                            // main program
    SegInit();                          // call function to initialise the 7-seg displays
    HostInit();                         // call function to initialise the host terminal
    while (1) {                         // infinite loop
        data2 = GetKeyInput();          // call function to get 1st key press
        Seg2  = SegConvert(data2);      // call function to convert and output
        data1 = GetKeyInput();          // call function to get 2nd key press
        Seg1  = SegConvert(data1);      // call function to convert and output
        pc.printf("  ");               // display spaces between 2 digit numbers
    }
}
```

# Creating a modular program

- The SegInit and SegConvert functions are to be ‘owned’ by SegDisplay.cpp, as are the BusOut objects named ‘Seg1’ and ‘Seg2’.
- The SegDisplay.cpp file should therefore be as follows:

```
// SegDisplay.cpp file for Exercise 6

#include "SegDisplay.h"
BusOut Seg1(p5,p6,p7,p8,p9,p10,p11,p12);      // A,B,C,D,E,F,G,DP
BusOut Seg2(p13,p14,p15,p16,p17,p18,p19,p20); // A,B,C,D,E,F,G,DP

void SegInit(void) {
    Seg1=SegConvert(0);         // initialise to zero
    Seg2=SegConvert(0);         // initialise to zero
}

char SegConvert(char SegValue) {          // function 'SegConvert'
    char SegByte=0x00;
    switch (SegValue) {                   //DP G F E D C B A
        case 0 : SegByte = 0x3F; break;   // 0 0 1 1 1 1 1 1 binary
        case 1 : SegByte = 0x06; break;   // 0 0 0 0 0 1 1 0 binary
        case 2 : SegByte = 0x5B; break;   // 0 1 0 1 1 0 1 1 binary
        case 3 : SegByte = 0x4F; break;   // 0 1 0 0 1 1 1 1 binary
        case 4 : SegByte = 0x66; break;   // 0 1 1 0 0 1 1 0 binary
        case 5 : SegByte = 0x6D; break;   // 0 1 1 0 1 1 0 1 binary
        case 6 : SegByte = 0x7D; break;   // 0 1 1 1 1 1 0 1 binary
        case 7 : SegByte = 0x07; break;   // 0 0 0 0 0 1 1 1 binary
        case 8 : SegByte = 0x7F; break;   // 0 1 1 1 1 1 1 1 binary
        case 9 : SegByte = 0x6F; break;   // 0 1 1 0 1 1 1 1 binary
    }
    return SegByte;
}
```

# Creating a modular program

- Note that SegDisplay.cpp file has a #include to the SegDisplay.h header file.
- SegDisplay.h should be as follows:

```
// SegDisplay.h file for Exercise 6

#ifndef SEGDISPLAY_H
#define SEGDISPLAY_H

#include "mbed.h"

extern BusOut Seg1;                 // allow Seg1 to be manipulated by other files
extern BusOut Seg2;                 // allow Seg2 to be manipulated by other files

void SegInit(void);                 // function prototype
char SegConvert(char SegValue);     // function prototype

#endif
```

# Creating a modular program

- The DisplaySet and GetKeyInput functions are to be ‘owned’ by HostIO.cpp, as is the Serial USB interface object named ‘pc’.
- The HostIO.cpp should therefore be as follows:

```
// HostIO.cpp code for Exercise 6

#include "HostIO.h"
#include "SegDisplay.h"        // allow access to functions and objects in SegDisplay.cpp

Serial pc(USBTX, USBRX);       // communication to host PC

void HostInit(void) {
    pc.printf("\n\rType two digit numbers to be displayed on the 7-seg display\n\r");
}

char GetKeyInput(void) {
    char c = pc.getc();         // get keyboard data (note numerical ascii range 0x30-0x39)
    pc.printf("%c",c);          // print ascii value to host PC terminal
    return (c&0x0F);            // return value as non-ascii (bitmask c with value 0x0F)
}
```

# Creating a modular program

- Note that HostIO.cpp file has #includes to both the HostIO.h and the SegDisplay.h header files.
- HostIO.h should be as follows:

```
// HostIO.h file for Exercise 6

#ifndef HOSTIO_H
#define HOSTIO_H

#include "mbed.h"
extern Serial pc;                   // allow pc to be manipulated by other files

void HostInit(void);                // function prototype
char GetKeyInput(void);             // function prototype

#endif
```

- Your Exercise 6 program should now compile and run

# Extended Task

- Exercise 7:
- Create a modular project which uses a host terminal application and a servo motor.
- The user can input a value between 1-9 which will move the servo motor to a specified position. An input of 1 moves the servo to 90 degrees left and 9 moves the servo to 90 degrees the right. Numbers in between 1-9 move the servo to a relative position, for example the value 5 will point the servo to the centre.
- You can reuse the GetKeyInput function from the previous exercise.
- You may also need to create a look up table function to convert the input value to a sensible PWM duty cycle value associated with the desired servo position.

# Summary

- Developing advanced embedded systems
- Functions and subroutines
- Working with 7-segment displays
- Building mbed projects with functions
- Modular programming
- Using header files
- Creating a modular program

# Modular design and Programming Techniques

**Train The Trainer - IAC**

# Parallel data and communication

Train The Trainer – IAC

# Agenda

- Using parallel digital outputs with the BusOut object
- Working with a parallel LCD display
- Hardware integration
- Using modular coding to interface the LCD display
- Initialising the LCD display
- Sending parallel display data to the LCD
- Adding data to a specified location
- Using the mbed TextLCD library
- Displaying variable data on the LCD

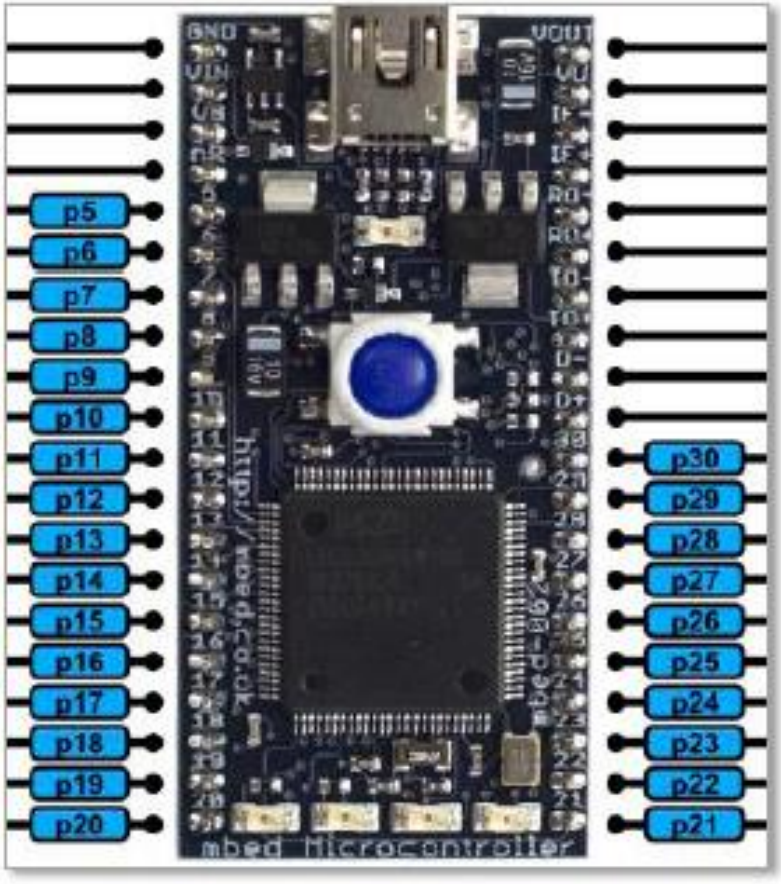
# Using parallel digital outputs with BusOut

- A digital bus is a system to transfer data between components, or between controllers.
- Buses can be parallel, carrying data on multiple wires, or serial, carrying data sequentially on a single wire.
- Conventional PCI, Parallel ATA and PCMCIA are examples of parallel buses.
- Examples of serial interfaces include PCIe, Ethernet, FireWire, USB, I2C, and SPI.
- The BusOut interface is used to create a number of parallel DigitalOut pins that can be written as one numeric value.
- The BusOut interface can be used to set the state of the output pins, and also read back the current output state.

# Using parallel digital outputs with BusOut

- On the mbed, the four on-board LEDs can be used to create a BusOut interface and have been specially configured to operate with no need for extra wires or connections.

- The mbed also has 26 digital IO pins (pins 5-30) which can be configured as a digital bus for use with the BusOut and BusIn interfaces.



# Using parallel digital outputs with BusOut

- The BusOut library functions are shown in the table below

| BusOut | A digital output bus, used for setting the state of a collection of pins |
|---|---|
| BusOut | Create a BusOut object, connected to the specified pins |
| write | Write the value to the output bus |
| read | Read the value currently output on the bus |
| operator= | A shorthand for write |
| operator int() | A shorthand for read |

# Using parallel digital outputs with BusOut

- We can use digital outputs to switch the on-board LEDs in a specific order.
- For example, we can use the code shown to produce a light that moves horizontally across the 4 on-board LEDs.
- The code is very simple, but it could become quite complex if we require more outputs, or to perform more on/off configurations of the LEDs.

```
#include "mbed.h"

DigitalOut led1(LED1);
DigitalOut led2(LED2);
DigitalOut led3(LED3);
DigitalOut led4(LED4);

int main() {
    while(1) {
            led1 = 1;
            led2 = 0;
            led3 = 0;
            led4 = 0;
            wait(0.25);
            led1 = 0;
            led2 = 1;
            led3 = 0;
            led4 = 0;
            wait(0.25);
            led1 = 0;
            led2 = 0;
            led3 = 1;
            led4 = 0;
            wait(0.25);
            led1 = 0;
            led2 = 0;
            led3 = 0;
            led4 = 1;
            wait(0.25);
        }
    }
```

# Using parallel digital outputs with BusOut

- **Exercise 1:** Using digital outputs, create a program to produce a “Knightrider” LED sweep effect with the on-board LEDs.

```
#include "mbed.h"
DigitalOut led1(LED1);
DigitalOut led2(LED2);
DigitalOut led3(LED3);
DigitalOut led4(LED4);

int main() {
    while(1) {
        led1 = 1; led2 = 0; led3 = 0; led4 = 0;
        wait(0.25);
        led1 = 0; led2 = 1; led3 = 0; led4 = 0;
        wait(0.25);
        led1 = 0; led2 = 0; led3 = 1; led4 = 0;
        wait(0.25);
        led1 = 0; led2 = 0; led3 = 0; led4 = 1;
        wait(0.25);
        led1 = 0; led2 = 0; led3 = 1; led4 = 0;
        wait(0.25);
        led1 = 0; led2 = 1; led3 = 0; led4 = 0;
        wait(0.25);
        }
}
```

# Using parallel digital outputs with BusOut

- **Exercise 2:** Using the BusOut object, create a program to produce a “Knightrider” sweep effect with the on-board LEDs.
- Verify that this program behaves the same as the previous exercise.

```
#include "mbed.h"

BusOut myleds(LED4, LED3, LED2, LED1);
char x=1;
int main() {
    while(1) {
        for(int i=0; i<3; i++) {          // x = a << b then x = a*2^b;
            x = x << 1;                  // x=1,2,4,8 or x=0001,0010,0100,1000
            myleds=x;                    // sweep left
            wait(0.2);
        }
        for(int i=0; i<3; i++) {          // x = a >> b then x = a/2^b;
            x = x >> 1;                  // x=8,4,2,1 or x=1000,0100,0010,0001
            myleds=x;                    // sweep right
            wait(0.2);
        }
    }
}
```

- Note: Shift operators, << and >>, are used to multiply and divide by two.

# Working with a parallel LCD display

- We will use the 2x16 character Powertip PC1602F LCD, though a number of similar LCD displays can be found with the same hardware configuration and functionality.


- The following must be achieved in order to interface the LCD:
  - Hardware integration: we will need to connect the LCD to the correct mbed pins.
  - Modular coding: as there are many processes that need to be completed, it makes sense to define LCD functions in modular files.
  - Initialising the LCD: a specific sequence of control signals must be sent to the LCD in order to initialise it.
  - Outputting data: we will need to understand how the LCD converts control data into legible display data.

# Hardware integration

- The PC1602F display is a 2x16 character display with an on board data controller chip and an integrated backlight.
- The LCD display has 16 connections as shown here.

| Pin Number | Pin Name | Function |
|---|---|---|
| 1 | $V_{SS}$ | Power supply (GND) |
| 2 | $V_{DD}$ | Power supply (5V) |
| 3 | $V_0$ | Contrast adjust |
| 4 | RS | Register select signal |
| 5 | R/W | Data read / write |
| 6 | E | Enable signal |
| 7 | DB0 | Data bus line bit 0 |
| 8 | DB1 | Data bus line bit 0 |
| 9 | DB2 | Data bus line bit 0 |
| 10 | DB3 | Data bus line bit 0 |
| 11 | DB4 | Data bus line bit 0 |
| 12 | DB5 | Data bus line bit 0 |
| 13 | DB6 | Data bus line bit 0 |
| 14 | DB7 | Data bus line bit 0 |
| 15 | A | Power supply for LED back light (5V) |
| 16 | K | Power supply for LED back light (GND) |

# Hardware integration

- The Powertip PC1602F datasheet is available from here:

  http://www.rapidonline.com/netalogue/specs/57-0911.pdf

- Operation and interfacing the LCD is summarised as follows:
  - The display is initialised by sending control instructions to the relevant configuration registers in the LCD. This is done by setting RS, R/W and E all low, then sending the correct data through bits DB0-DB7.
  - We will use the LCD in 4-bit mode which means that we only need to use the final 4-bits of the data bus (DB4-DB7). This means we can control the LCD with only 7 digital lines, rather than 11 lines which are required for 8-bit mode.
  - After each data byte has been sent, the Enable bit must be toggled on then off again, this tells the LCD that data is ready and should be processed.
  - Once the LCD has been initialised, display data can be sent by setting the RS bit. Again, after each byte of display data has been sent, the Enable bit should be toggled to process the data.

# Hardware integration

- We obviously need a digital mbed pin to attach to each of the LCD data pins. We need 4 digital outputs to send the 4-bit instruction and display data and 3 digital outputs to manipulate the RS, R/W and E control flags.

- We can connect the PC1602F to the mbed using the following interface configuration:
- Note: in general, we only use the LCD in write mode, so we tie R/W permanently to ground (mbed pin 1).

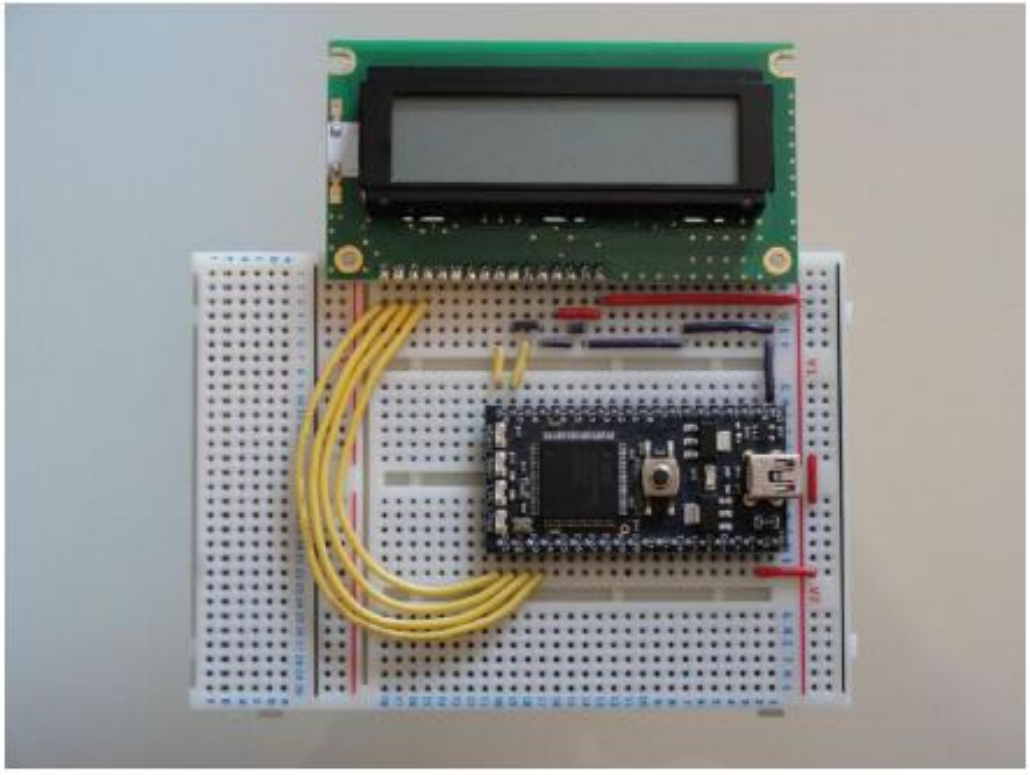
| mbed Pin number | LCD Pin Number | LCD Pin Name | Power Connections |
|---|---|---|---|
| 1 | 1 | $V_{SS}$ | 0V |
| 39 | 2 | $V_{DD}$ | 5V |
| 1 | 3 | $V_0$ | 0V |
| 19 | 4 | RS | |
| 1 | 5 | R/W | 0V |
| 20 | 6 | E | |
| 21 | 11 | DB4 | |
| 22 | 12 | DB5 | |
| 23 | 13 | DB6 | |
| 24 | 14 | DB7 | |
| 39 | 15 | A | 5V |
| 1 | 16 | K | 0V |

# Hardware integration

- Note that the PC1602F has a non-conventional pin layout which reads from left to right:

| 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 16 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| DB7 | DB6 | DB5 | DB4 | DB3 | DB2 | DB1 | DB0 | E | R/W | RS | $V_0$ | $V_{DD}$ | $V_{SS}$ | A | K |

The hardware setup is as shown.



# Using modular coding to interface the LCD

- We will use three files for the LCD application:
  - A main code file (main.cpp) which can call functions defined in the LCD feature file.
  - An LCD feature file (LCD.cpp) which will include all the functions for initialising and sending data to the LCD.
  - An LCD header file (LCD.h) which will be used to declare data and function prototypes.

# Using the LCD display

- We will declare the following functions in our LCD header file
  - toggle_enable - function to toggle the enable bit
  - LCD_init - function to initialise the LCD
  - display_to_LCD - function to display characters on the LCD

- Our LCD.h file should therefore be as follows:

```
// LCD.h file

#ifndef LCD_H
#define LCD_H

#include "mbed.h"

void display_to_LCD(char value);     //function to display characters on the LCD
void toggle_enable(void);            //function to toggle the enable bit
void LCD_init(void);                 //function to initialise the LCD

#endif
```

# Initialising the LCD display

- In LCD.cpp:
  - We define the digital IO classes for the mbed. We need one digital output for each of RS and E and we will use the mbed BusOut class for the 4-bit data.
  - We send a number of 4-bit data packets to the LCD in order to initialise it and to display alphanumeric characters on the display.
  - After each data packet has been sent, the LCD requires the Enable bit to be toggled (i.e. sent high and then low with a pause in between). This is done by the **toggle_enable** function.

```
// define mbed objects
DigitalOut RS(p19);
DigitalOut E(p20);
BusOut data(p21, p22, p23, p24);
```

```
/**** toggle enable function
void toggle_enable(void){
    E=1;
    wait(0.001);
    E=0;
    wait(0.001);
}
```

# Initialising the LCD display

- A specific initialisation procedure must be followed in order for the PC1602F display to operate correctly. Please refer to the PC1602F datasheet for more specific configuration details.
- We will code the initialisation routine using the **LCD_init** function:
  - In order to initialise we first need to wait a short period (approximately 20ms), set the RS and E lines to zero and then send a number of configuration messages to set up the LCD functionality.
  - To set the LCD **function mode** we send a binary value of 0010 1000 (0x28 hex) to the LCD data pins, we define 4-bit mode, 2 line display and 5x7 dot characters.
  - Note that as we are using 4-bit mode, we need to send two pieces of 4-bit data for each instruction; effectively setting one 8-bit register by sending two 4-bit packets of data.

```
// Function Mode
data=0x2;
toggle_enable();
data=0x8;
toggle_enable();
```

# Initialising the LCD display

- The LCD **display mode** control register must also be set during initialisation.

```
// Display Mode
data=0x0;
toggle_enable();
data=0xF;
toggle_enable();
```

- Here we need to send a command to switch the display on, and to determine the cursor function. Value 0x0F will switch the display on with a blinking cursor.

- Before data can be written to the display, the display must first be cleared and the cursor reset to the first character in the first row. This is done by the **clear display** command with a value 0x01.

```
// Clear display
data=0x0;
toggle_enable();
data=0x1;
toggle_enable();
```

# Sending parallel display data to the LCD

- Display data is sent to the LCD screen by the display_to_LCD function. This function performs the following tasks:
  - Setting the RS flag to 1 (data setting).
  - Sending a data byte describing the ascii character to be displayed.
  - Toggle the enable flag.
  - The character displayed to the LCD is described by an 8-bit hexadecimal ascii value. The complete ascii table is included with the LCD datasheet.

```
/**** display ****/
void display_to_LCD(char value ){

  RS=1;

  //***** display character

  data=value>>4;      // upper 4 bits
  toggle_enable();
  data=value&0x0F;    // lower 4 bits
  toggle_enable();

}
```

# Sending parallel display data to the LCD

- If we wish to display the word “HELLO”, for example, the hexadecimal ascii values required are as follows: 0x48, 0x45, 0x4C, 0x4C and 0x4F.
- Other ascii values can be found in the table below:

| | | LSB | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 0x0 | 0x1 | 0x2 | 0x3 | 0x4 | 0x5 | 0x6 | 0x7 | 0x8 | 0x9 | 0xA | 0xB | 0xC | 0xD | 0xE | 0xF |
| MSB | 0x0 | | | | | | | | | | | | | | | | |
| | 0x1 | | | | | | | | | | | | | | | | |
| | 0x2 | | ! | " | # | $ | % | & | ' | ( | ) | * | + | , | - | . | / |
| | 0x3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; | < | = | > | ? |
| | 0x4 | @ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| | 0x5 | P | Q | R | S | T | U | V | W | X | Y | Z | [ | \ | ] | ^ | _ |
| | 0x6 | ` | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
| | 0x7 | p | q | r | s | t | u | v | w | x | y | z | { | \| | } | ~ | |

# Sending parallel display data to the LCD

- Digital IO and LCD functions are therefore defined in LCD.cpp as below 21

```
// LCD.cpp file

#include "LCD.h"

DigitalOut RS(p19);
DigitalOut E(p20);
BusOut data(p21, p22, p23, p24);

void toggle_enable(void){
  E=1;
  wait(0.001);
  E=0;
  wait(0.001);
}

//initialise LCD function
void LCD_init(void){
  wait(0.02);
  RS=0;
  E=0;

  //function mode
  data=0x2;
  toggle_enable();
  data=0x8;
  toggle_enable();

// continued...
```

```
//... LCD.cpp continued...

  //display mode
  data=0x0;
  toggle_enable();
  data=0xF;
  toggle_enable();

  //clear display
  data=0x0;
  toggle_enable();
  data=0x1;
  toggle_enable();
}

//display function
void display_to_LCD(char value ){
  RS=1;
  data=value>>4;
  toggle_enable();
  data=value&0x0F;
  toggle_enable();
}
```

# Using the LCD display

- Exercise 3: Connect the LCD to an mbed and construct a new program with the files main.cpp, LCD.cpp and LCD.h ,as described in the previous slides.
- Add the following code to your main.cpp file and compile and run on the mbed:

```
#include "LCD.h"

int main() {
    LCD_init();
    display_to_LCD(0x48);    // 'H'
    display_to_LCD(0x45);    // 'E'
    display_to_LCD(0x4C);    // 'L'
    display_to_LCD(0x4C);    // 'L'
    display_to_LCD(0x4F);    // 'O'
}
```

- Verify that the word 'HELLO' is correctly displayed with a flashing cursor.

# Using the mbed TextLCD library

- The mbed TextLCD library is more advanced than the simple functions we have created.
  - The TextLCD library performs the laborious LCD setup routine for us
  - The TextLCD definition also tells the LCD object which pins are used for which functions
- The pin definition is defined in the following manner:

```
TextLCD lcd(int rs, int e, int d0, int d1, int d2, int d3);
```

- We need to ensure that our pins are defined in the same order. For our hardware setup this will be:

```
TextLCD lcd(p19, p20, p21, p22, p23, p24);
```

- We use printf statements to display characters on the LCD screen.

# Using the mbed TextLCD library

- Exercise 4: Compile a “Hello World” example using the mbed library, which makes use of an alphanumeric LCD much simpler and quicker to program.

```
#include "mbed.h"
#include "TextLCD.h"
TextLCD lcd(p19, p20, p21, p22, p23, p24); //rs,e,d0,d1,d2,d3

int main() {
    lcd.printf("Hello World!");
}
```

- Import the mbed TextLCD.h library file to your project (right click and select ‘import library’).
- This library is effectively a file full of specific LCD functions already written for you. Use the following link for the library file:

  http://mbed.org/users/simon/libraries/TextLCD/livod0

# Using the mbed TextLCD library

- The cursor can be moved to a chosen position to allow you to choose where to display data, for example

```
lcd.locate(3,1);
```

- The display is laid out as 2 rows (0-1) of 16 columns (0-15). The locate function defines the column first followed by the row.
  - The above example moves the cursor to the 4th column and 2nd line
  - Any printf statements after the locate command will be printed at the new cursor location.

- We can also clear the screen with the following command:

```
lcd.cls();
```

# Using the mbed TextLCD library

- We display data on the screen using the standard printf statement too. If we want to display the value of an integer to the screen, we need to:
  - –declare the variable
  - –give the variable a value
  - –display the variable to the screen by using the printf statement,

```
x = 1028
lcd.printf("%i",x);
```

- Note that the “%i” command is used to indicate that an integer is to be output, and the integer name follows.

# Displaying variables on the LCD

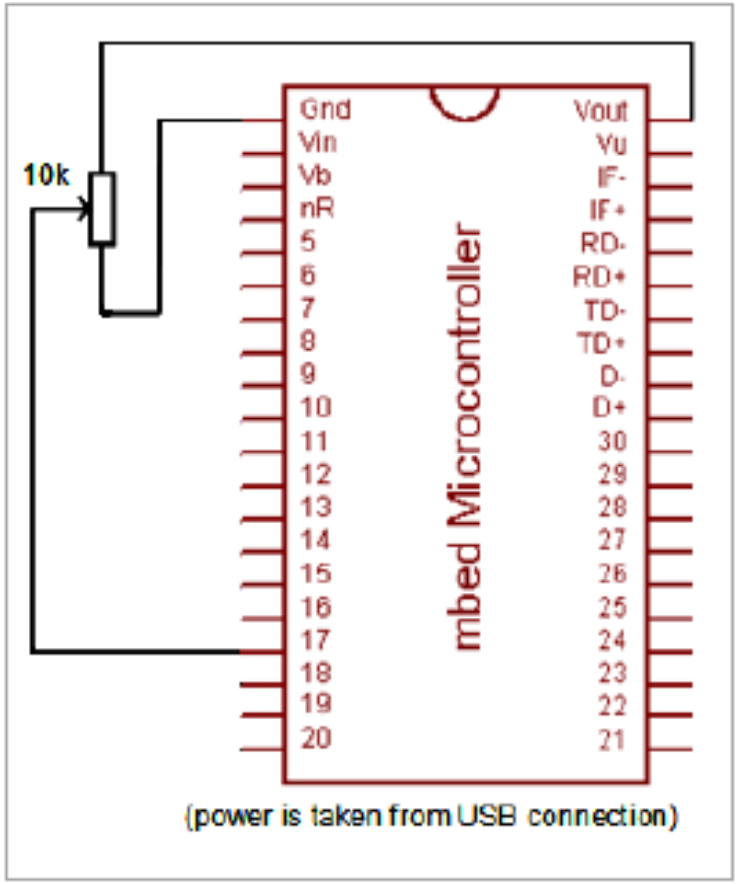
- Exercise 5: display a continuous count variable on the LCD display by implementing the following code:

```
#include "mbed.h"
#include "TextLCD.h"
TextLCD lcd(p19, p20, p21, p22, p23, p24); // rs, e, d0, d1, d2, d3
int x=0;
int main() {
    lcd.printf("LCD Counter");
    while (1) {
        lcd.locate(5,1);
        lcd.printf("%i",x);
        wait(1);
        x++;
    }
}
```

- Don’t forget to import the TextLCD library!
- Increase the speed of the counter and investigate how the cursor position changes as the count value increases.

# Displaying analog input data on the LCD

- Exercise 6: Display the analog value to the screen.
  - You will need to use a potentiometer to provide an analog input.
  - The analog input variable has a floating point value between 0 and 1, where 0 is 0V and 1 represents 3.3V.
  - We will multiply the analog input value by 100 to make it a percentage between 0-100%.
  - An infinite loop is used so that the screen continuously updates automatically. To do this it is necessary to clear the screen and add a delay to set the update frequency.



# Displaying analog input data on the LCD

- Add the following to your main.cpp . Your code should now compile and run:

```
#include "mbed.h"
#include "TextLCD.h"

TextLCD lcd(p19, p20, p21, p22, p23, p24); //rs,e,d0, d1,d2,d3
AnalogIn Ain(p17);
int percentage;

int main() {
    while(1){
        percentage=Ain*100;
        lcd.printf("%i",percentage);
        wait(0.002);
        lcd.cls();
    }
}
```

- The analog value should change as you move the position of the potentiometer.

# Extended task

- Exercise 7: Create a program to make the mbed and display act like a standard voltmeter. Potential difference should be measured between 0 - 3.3 V and display this to the screen similar to that shown below:


- Note of the following:
  - You will need to convert the 0.0 - 1.0 analog input value to a value which represents 0 - 3.3 Volts.
  - You will need to use an infinite loop to allow the voltage value to continuously update as the potentiometer position changes.
  - Check your display with the reading from an actual voltmeter – is it accurate?

# Summary

- Using parallel digital outputs with the BusOut object
- Working with a parallel LCD display
- Hardware integration
- Using modular coding to interface the LCD display
- Initialising the LCD display
- Sending parallel display data to the LCD
- Adding data to a specified location
- Using the mbed TextLCD library
- Displaying variable data on the LCD

# Parallel data and communication

Train The Trainer - IAC

# Serial communications with I2C

**Train The Trainer – IAC**

# Agenda

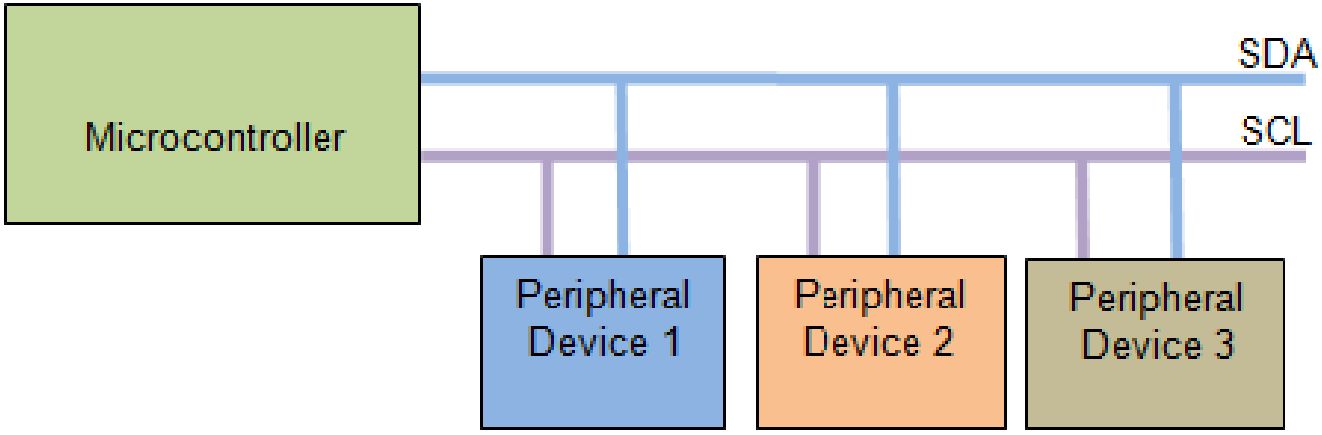
- Introducing I2C
- Evaluating simple I2C communications
- I2C on the mbed
- Working with the TMP102 I2C temperature sensor
- Working with the SRF08 ultrasonic rangefinder
- Interfacing multiple devices on a single I2C bus
- Extended exercises

# Introducing I2C

- The name I2C is shorthand for Standard Inter-Integrated Circuit bus
- I2C is a serial data protocol which operates with a master/slave relationship
- I2C only uses two physical wires, this means that data only travels in one direction at a time.

# Introducing I2C

- The I2C protocol is a two-wire serial bus:



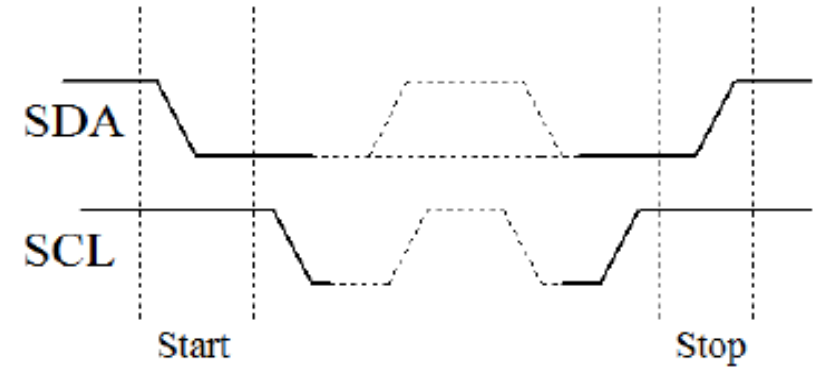

- The I2C communication signals are serial data (SDA) and serial clock (SCL)
  - These two signals make it possible to support serial communication of 8-bit data bytes, 7-bit device addresses as well as control bits
  - Using just two wires makes it cheap and simple to implement in hardware

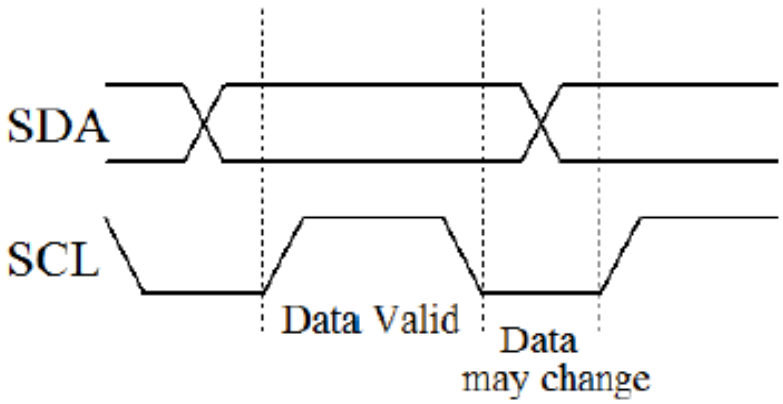
# Evaluating simple I2C communications

- I2C has a built-in addressing scheme, which simplifies the task of linking multiple devices together.
  - In general, the device that initiates communication is termed the ‘master’. A device being addressed by the master is called a ‘slave’.
  - Each I2C-compatible slave device has a predefined device address. The slaves are therefore responsible for monitoring the bus and responding only to data and commands associate with their own address.
  - This addressing method, however, limits the number of identical slave devices that can exist on a single I2C bus, as each device must have a unique address. For some devices, only a few of the address bits are user configurable.

# Evaluating simple I2C communications

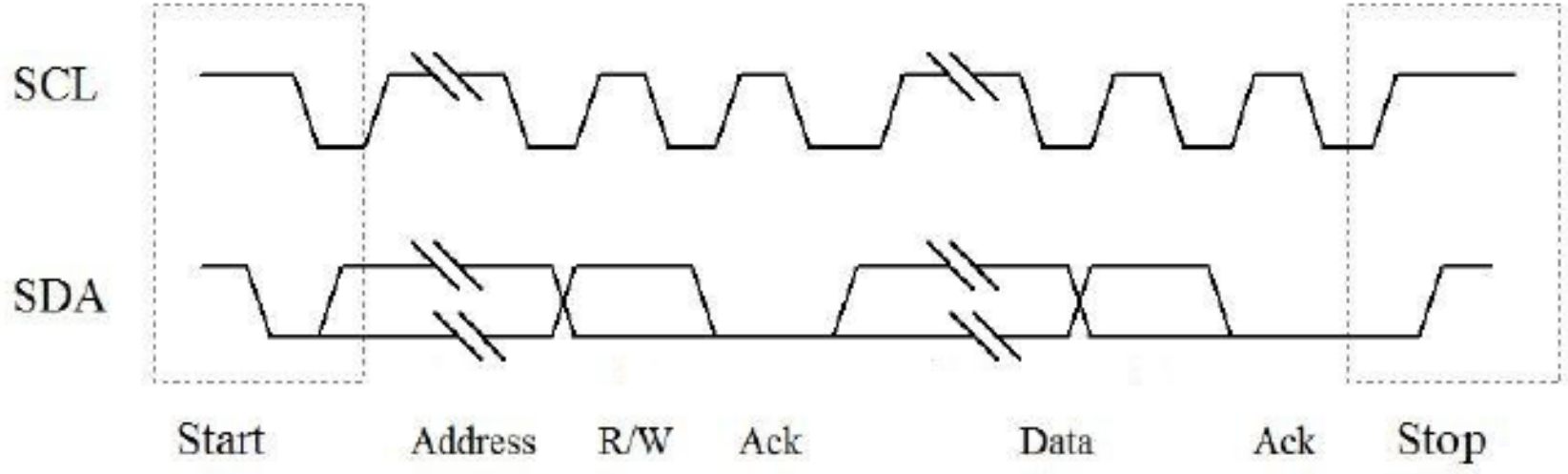
- A data transfer is made up of the Master signalling a Start Condition, followed by one or two bytes containing address and control information.
- The Start condition is defined by a high to low transition of SDA when SCL is high.
- A low to high transition of SDA while SCL is high defines a Stop condition
- One SCL clock pulse is generated for each SDA data bit, and data may only change when the clock is low.



# Evaluating simple I2C communications

- The byte following the Start condition is made up of seven address bits, and one data direction bit (Read/Write)
- All data transferred is in units of one byte, with no limit on the number of bytes transferred in one message.
- Each byte must be followed by a 1-bit acknowledge from the receiver, during which time the transmitter relinquishes SDA control.



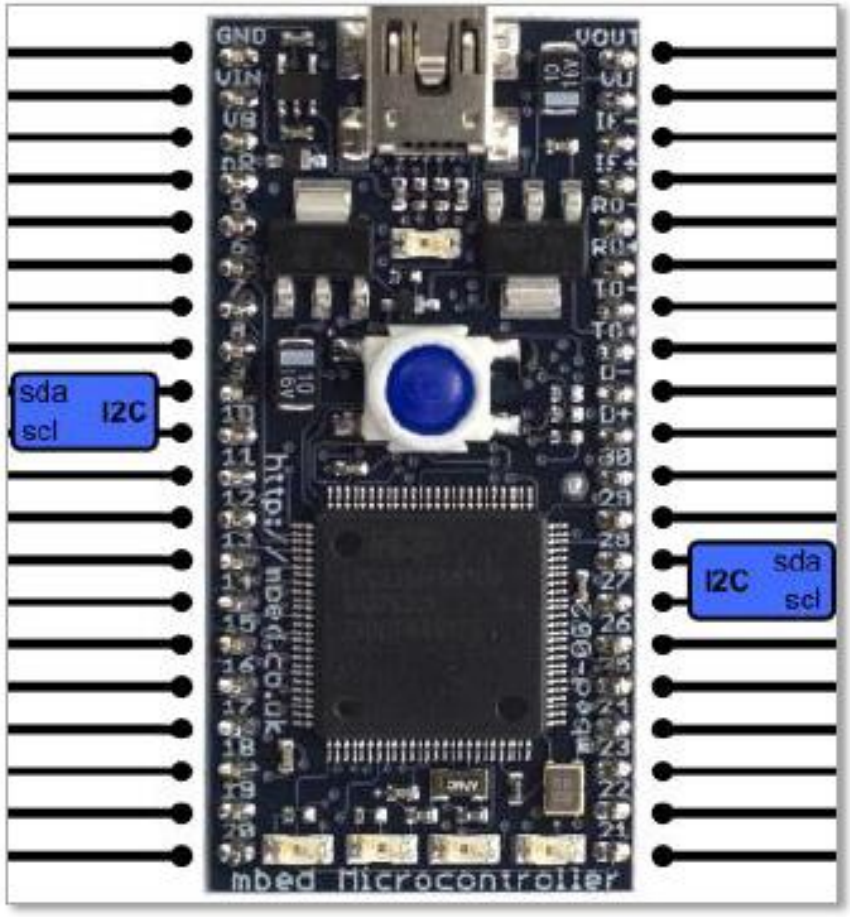
# I2C on the mbed

- The mbed I2C library functions are shown in the table below:

| I2C | An I2C Master, used for communicating with I2C slave devices |
|---|---|
| **Functions** | **Usage** |
| I2C | Create an $I^2C$ Master interface, connected to the specified pins |
| frequency | Set the frequency of the $I^2C$ interface |
| read | Read from an $I^2C$ slave |
| read | Read a single byte from the $I^2C$ bus |
| write | Write to an $I^2C$ slave |
| write | Write single byte out on the $I^2CC$ bus |
| start | Creates a start condition on the $I^2C$ bus |
| stop | Creates a stop condition on the $I^2C$ bus |

# I2C on the mbed

- The I2C Interface can be used on mbed pins p9/p10 and p28/p27
- Note also that the SDA and SCL data signals each need to be ‘pulled up’ to 3.3V, with a resistor value which can be optimised for the exact circuit configuration; in this setup we choose 2.2 kΩ resistor



# Evaluating the TMP102 I2C temperature sensor

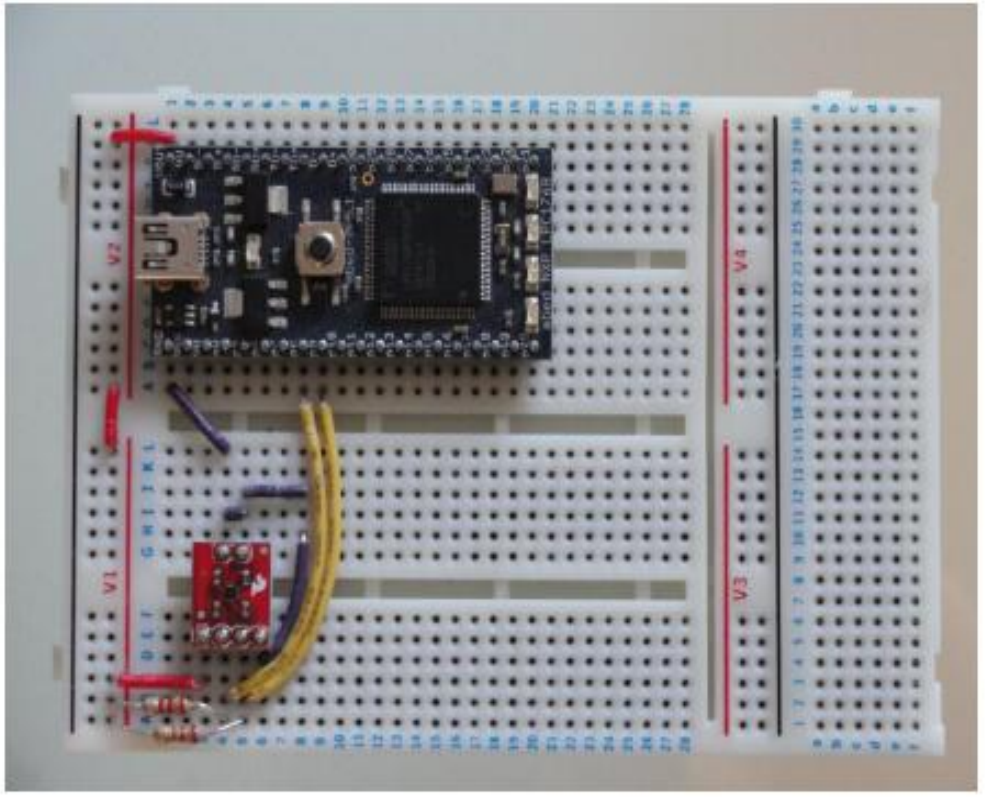
- Configuration and data register details are given in the TMP102 data sheet

  *http://focus.ti.com/lit/ds/sbos397b/sbos397b.pdf*
- To configure the temperature sensor we need to:
  - Use arrays of 8-bit values for the data variables, because the I2C bus can only communicate data in one bytes.
  - Set the configuration register; we first need to send a data byte of 0x01 to specify that the Pointer Register is set to ‘Configuration Register’.
  - Send two bytes of data to perform the configuration. A simple configuration value to initialise the TMP102 to normal mode operation is 0x60A0.
- To read the data register we need to:
  - To read the data register we need to set the pointer value to 0x00.
  - To print the data we need to convert the data from a 16-bit data reading to an actual temperature value. The conversion required is to shift the data right by 4 bits (its actually only 12-bit data held in two 8-bit registers) and to multiply by the 1-bit resolution which is 0.0625 degrees C per LSB.

# Interfacing the TMP102 with the mbed

- The TMP102 can be connected to the mbed as shown:

| Signal | TMP102 Pin | Mbed Pin | Notes |
|---|---|---|---|
| Vcc (3.3V) | 1 | 40 | |
| Gnd (0V) | 4 | 1 | |
| SDA | 2 | 9 | 2.2kΩ pull-up to 3.3V |
| SCL | 3 | 10 | 2.2kΩ pull-up to 3.3V |



# Working with the TMP102 I2C temperature sensor

- Exercise 1: Connect the temperature sensor to an I2C bus. Verify that the correct data can be read by continuously display updates of temperature to the screen.
  - Test that the temperature increases when you press your finger against the sensor. You can even try placing the sensor on something warm, for example a pocket hand warmer, in order to check that it reads temperature correctly.

# Working with the TMP102 I2C temperature sensor

- The following program will configure the TMP102 sensor, read data, convert data to degrees Celsius and then display values to the screen every second:

```
#include "mbed.h"
I2C tempsensor(p9, p10); //sda, sc1
Serial pc(USBTX, USBRX); //tx, rx
const int addr = 0x90;
char config_t[3];
char temp_read[2];
float temp;
int main() {
    config_t[0] = 0x01;                    //set pointer reg to 'config register'
    config_t[1] = 0x60;                    // config data byte1
    config_t[2] = 0xA0;                    // config data byte2
    tempsensor.write(addr, config_t, 3);
    config_t[0] = 0x00;                    //set pointer reg to 'data register'
    tempsensor.write(addr, config_t, 1);   //send to pointer 'read temp'
    while(1) {
        wait(1);
        tempsensor.read(addr, temp_read, 2);       //read the two-byte temp data
        temp = 0.0625 * (((temp_read[0] << 8) + temp_read[1]) >> 4);  //convert data
        pc.printf("Temp = %.2f degC\n\r", temp);
    }
}
```

# Evaluating the SRF08 ultrasonic rangefinder

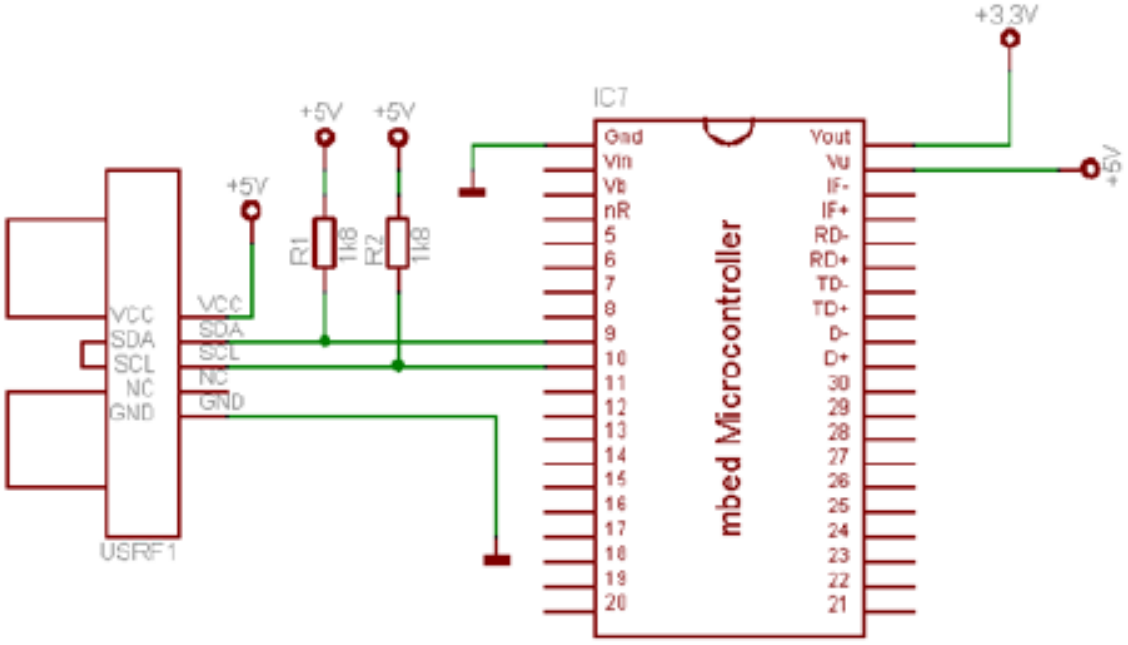
- Configuration and data register details are given in the SRF08 data sheet:

  *http://www.rapidonline.com/netalogue/specs/78-1086.pdf*

- The following information summarises the configuration and data read procedures for the SRF08:
  - The rangefinder I2C address is 0xE0.
  - The pointer value for the command register is 0x00.
  - A data value of 0x51 to the command register initialises the range finder to operate and return data in cm.
  - A pointer value of 0x02 prepares for 16-bit data (i.e. two bytes) to be read.

# Interfacing the SRF08 with the mbed

- The SRF08 ultrasonic range finder is an I2C device which measures distance and proximity of items
- The SRF08 can be connected to the mbed as shown:



# Interfacing the SRF08 with the mbed

- Exercise 2: Configure the SRF08 ultrasonic rangefinder to update distance data to the screen. The following code will perform this operation:

```
#include "mbed.h"
I2C rangefinder(p9, p10); //sda, sc1
Serial pc(USBTX, USBRX); //tx, rx
const int addr = 0xE0;
char config_r[2];
char range_read[2];
float range;
int main() {
    while (1) {
        config_r[0] = 0x00;                         //set pointer reg to 'cmd register'
        config_r[1] = 0x51;                         // config data byte1
        rangefinder.write(addr, config_r, 2);
        wait(0.07);
        config_r[0] = 0x02;                         //set pointer reg to 'data register'
        rangefinder.write(addr, config_r, 1);       //send to pointer 'read range'
        rangefinder.read(addr, range_read, 2);      //read the two-byte range data
        range = ((range_read[0] << 8) + range_read[1]);
        pc.printf("Range = %.2f cm\n\r", range);   //print range on screen
        wait(0.05);
    }
}
```

# Interfacing the SRF08 with the mbed

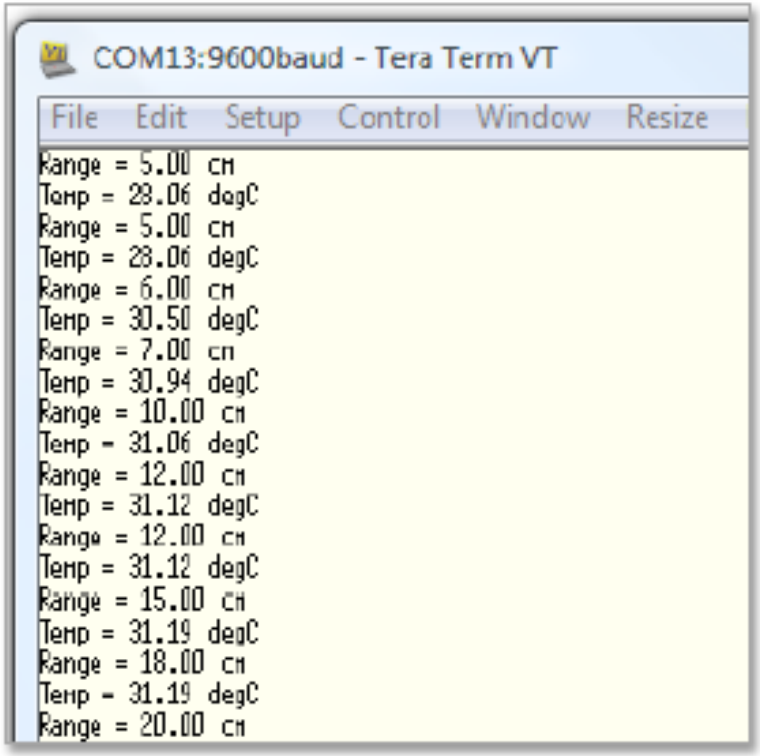
- Exercise 3:
- Evaluate the SRF08 datasheet to understand the different control setup and data conversion options.
- Now configure the SRF08 to accurately return data in inches measured to a maximum range of 10 feet.
- Note:

  1 inch = 25.4 mm

  1 foot = 12 inches 17

# Interfacing multiple devices on an I2C bus

- Exercise 4: Connect both the temperature sensor and the range finder to the same I2C bus and verify that the correct data can be read by addressing commands appropriately. Update your screen readout to continuously display updates of temperature and detected range.



# Extended Exercises

- Exercise 5: Use an ultrasonic range finder to control the position of a servo motor. When items are distant, the servo will take a 0 degrees position, but as objects come within range of the SRF08, the servo will move towards a 180 degrees position.

- Exercise 6: Communicate between two mbeds on an I2C bus. Program the master mbed to count a value from 0 to 15 and to broadcast this count value on the I2C bus. The slave mbed should be programmed to read the I2C data at regular intervals and set the onboard LEDs to display the binary count value.

# Summary

- Introducing I2C
- Evaluating simple I2C communications
- I2C on the mbed
- Working with the TMP102 I2C temperature sensor
- Working with the SRF08 ultrasonic rangefinder
- Interfacing multiple devices on a single I2C bus
- Extended exercises

# Serial communications with I2C

**Train The Trainer - IAC**

# Serial communications with SPI

**Train The Trainer – IAC**

# Agenda

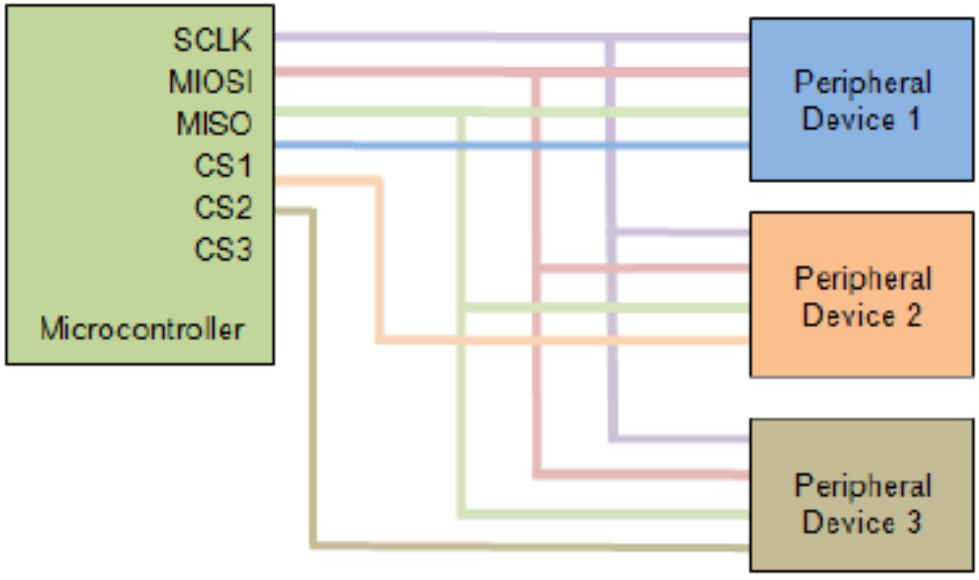
- Introducing SPI
- Evaluating simple SPI communications
- SPI on the mbed
- Evaluating the ADXL345 SPI accelerometer
- Interfacing the ADXL345 with the mbed
- Interfacing the Nokia 6610 display with the mbed
- Interfacing multiple devices on a single SPI bus
- Digital spirit level design challenge

# Introducing SPI

- Serial Peripheral Interface Bus (SPI) is a serial data protocol which operates with a master/slave relationship
- When the master initiates communication and selects a slave device, data can be transferred in either or both directions simultaneously
- A master must send a byte to receive a byte - transmitting a dummy byte is a typical way to make a read transaction

# Introducing SPI

- The SPI protocol uses four signals: clock (SCLK); master data output, slave data input (MOSI); master data input, slave data output (MISO); and slave chip select (CS)
- When multiple slave devices exist, the master must output a unique CS signal for each slave



# Evaluating simple SPI communications

- SPI is suited to transferring data streams, for example data communication between microprocessors or data transfer from an analogue-to-digital converter
- It can achieve higher data rates than I2C, as it doesn't send address or control information, and doesn't include an acknowledge bit for every byte
- SPI data communication is ideal for use in digital signal processing applications, where data is regularly exchanged between the master and slave

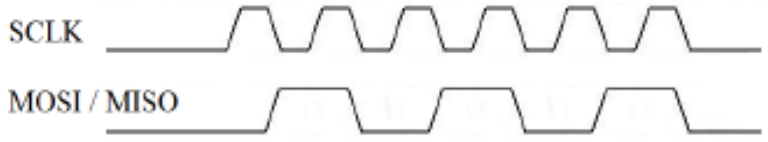
# Evaluating simple SPI communications

- A SPI data transfer is initiated by the master device as follows:
  - The master first configures the SPI clock (SCLK) to be a frequency supported by the recipient slave device (up to 70 MHz)
  - The master sets the chip select signal (CS) of the intended recipient slave to 0 V
  - The master then starts the clock pulses on SCLK to indicate that data is to be transferred
  - The master simultaneously sends data as consecutive bits on MOSI
  - The number of bits in each data frame can be configured, but is usually between 4 and 16 bits
  - The slave returns data in the same manner on MISO

# Evaluating simple SPI communications

- The master must also configure the clock's polarity (CPOL) and phase (CPHA) as follows:

| CPOL | | |
|---|---|---|
| 0 | Clock active high (off =0) |  |
| 1 | Clock active low (off=1) |  |

| CPHA | | |
|---|---|---|
| 0 | Clock out of phase with data (CPOL = 0) |  |
| 1 | Clock in phase with data (CPOL = 0) |  |

# Evaluating simple SPI communications

- We therefore have four SPI Modes as follows:

| Mode | CPOL | CPHA |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 2 | 1 | 0 |
| 3 | 1 | 1 |

- In general, SPI devices are designed to operate in one of the four modes and this will described in the device datasheet

# SPI on the mbed

- The mbed SPI library functions are shown in the table below:

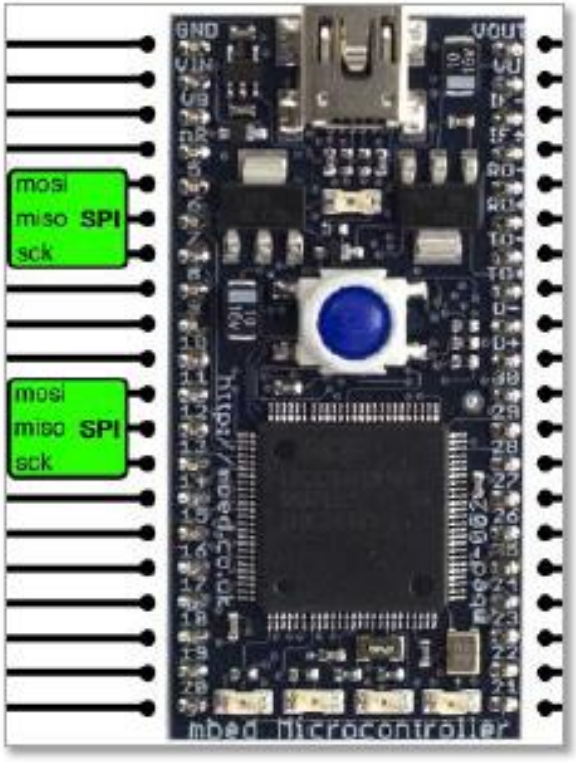
| SPI | A SPI Master, used for communicating with SPI slave devices Functions |
|---|---|
| **Functions** | **Usage** |
| SPI | Create a SPI master connected to the specified pins format |
| format | Configure the data transmission mode and data length |
| frequency | Set the SPI bus clock frequency |
| write | Write to the SPI Slave and return the response |

Note: this table is for the SPI master library. There is also a SPI slave library which is is used for communicating with a SPI Master device. The SPISlave library is not covered in these slides.

# SPI on the mbed

- The SPI Interface can be used on pins p5/p6/p7 and p11/p12/p13

- Default settings of the SPI interface on the mbed:
  - Default clock frequency of 1 MHz
  - Default data length of 8 bits
  - Default mode of 0



# Evaluating the ADXL345 SPI accelerometer

- Configuration and data register details are given in the ADXL345 datasheet

  *http://www.analog.com/static/imported-files/data_sheets/ADXL345.pdf*

- The accelerometer is actually analogue in nature, measuring acceleration on 3 axes.
- This kind of accelerometer has an internal capacitor mounted in the plane of each axis. Acceleration causes the capacitor plates to move, hence changing the output voltage proportionally to the acceleration or force.
- The ADXL345 accelerometer converts the analogue voltage fluctuations to digital and outputs this over a digital communication protocol.
- The ADXL345 can be configured to communicate in SPI and I2C modes.

# Evaluating the ADXL345 SPI accelerometer

- To configure the accelerometer for SPI communication we need to:
  - Set the SPI frequency to 1-5 MHz and SPI Mode 3
  - Set the Data Format by writing to register 0x31; a data byte of 0x0B will configure the accelerometer data to be in the range ±16g and to a resolution of 0.004g/LSB (1g is the value of acceleration due to earth's gravity, i.e. 9.81ms-2)
  - Set the device into 'measure' mode by sending a data value 0x08 to the Power Control register (address 0x2D)

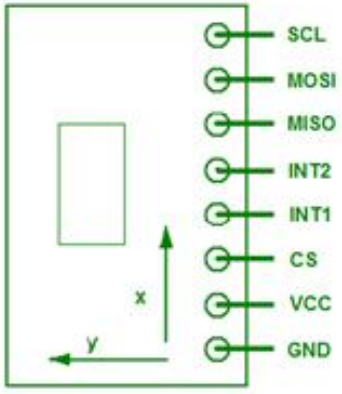
- When writing to the ADXL345 we must follow the following sequential procedure
  - Set CS low
  - Send register address byte
  - Send data byte
  - Set CS high

# Evaluating the ADXL345 SPI accelerometer

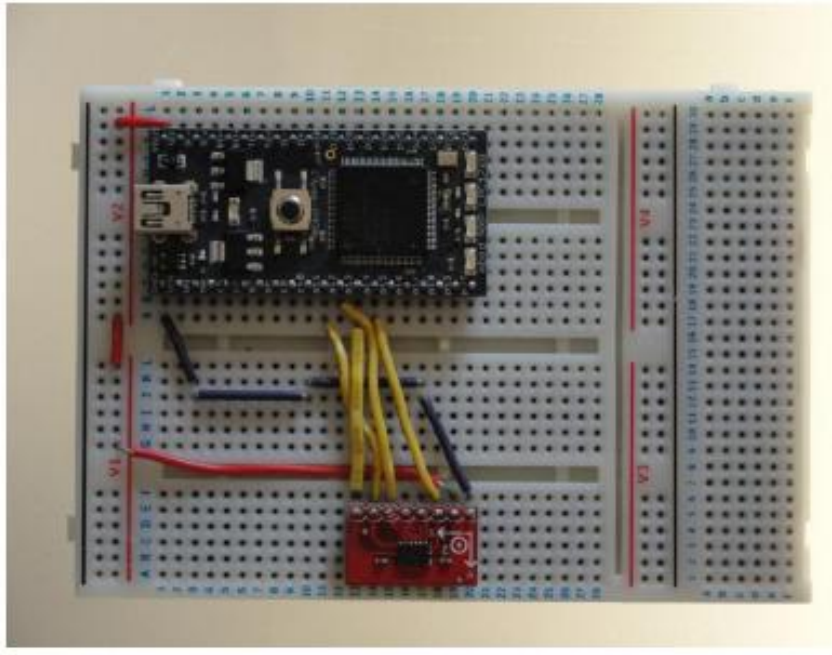
- To read 3-axis data back from the ADXL345 we must:
  - Set the read/write bit high (bit 7 of the address byte)
  - Set the multibyte-data flag high (bit 6 of the address byte)
  - Set CS low
  - Send the configured address byte for register 0x32
  - Read 6 bytes of data back from the ADXL345 by writing 6 dummy bytes of 0x00
  - Set CS high
- The 6 returned data bytes contain the most significant and least significant bytes for each of the three measurement axes (x, y and z)
- We therefore need to convert the data to floating point ‘g’ values by combining the relevant data bytes and multiplying by the configured data resolution.

# Interfacing the ADXL345 SPI accelerometer with the mbed

- The ADXL345 can be connected to the mbed as shown:




| ADXL345 signal name | mbed pin |
|---|---|
| Vcc | 40 |
| Gnd | 1 |
| SCL | 13 |
| MOSI | 11 |
| MISO | 12 |
| CS | 14 |



# Interfacing the ADXL345 SPI accelerometer with the mbed

- **Exercise 1:** Configure the ADXL345 accelerometer to continuously output 3-axis data to the terminal screen.
  - Set up the SPI accelerometer as described in the previous slides
  - Initiate a data read by setting CS high and writing the configured address value
  - Use a 6 element ‘char’ buffer to read the raw data back from the accelerometer
  - Combine the relevant most significant and least significant bytes into three twos- complement, 16-bit integer values (to keep things simple, you can make use of the 16int_t data type for this to function correctly)
  - Convert the 16-bit data to floating point by multiplying each axis reading by the configured data resolution
  - Output and display data to a host terminal application

# Interfacing the ADXL345 SPI accelerometer with the mbed

```
// solution for SPI Exercise 1
#include "mbed.h"
SPI acc(p11,p12,p13);                   // setup SPI interface on pins 11,12,13
DigitalOut cs(p14);                     // use pin 14 as chip select
Serial pc(USBTX, USBRX);                // setup USB interface to host terminal
char buffer[6];                         // raw data array type char
signed short data[3];                   // acc data is signed 16 bit from -32,768 to +32,767
float x, y, z;                          // floating point data
int main() {
    cs=1;
    acc.format(8,3);                    // 8 bit data, Mode 3
    acc.frequency(2000000);             // 2MHz clock rate
    cs=0;
    acc.write(0x31);                    // data format register
    acc.write(0x0B);                    // format +/-16g, 0.004g/LSB
    cs=1;
    cs=0;
    acc.write(0x2D);                    // power ctrl register
    acc.write(0x08);                    // measure mode
    cs=1;
    while (1) {                         // infinite loop
        wait(0.2);
        cs=0;
        acc.write(0x80|0x40|0x32);                  // RW bit high, MB bit high, plus address
        for (int i = 0;i<=5;i++) {
            buffer[i]=acc.write(0x00);              // read back 6 data bytes
        }
        cs=1;
        data[0] = buffer[1]<<8 | buffer[0];         // combine MSB and LSB
        data[1] = buffer[3]<<8 | buffer[2];
        data[2] = buffer[5]<<8 | buffer[4];
        x=0.004*data[0]; y=0.004*data[1]; z=0.004*data[2];      // convert to floating point
        pc.printf("x = %+1.2fg\t y = %+1.2fg\t z = %+1.2fg\n\r", x, y,z); //print to screen
    }
}
```

# Interfacing the ADXL345 SPI accelerometer with the mbed

- **Exercise 2:** Experiment with advanced configuration parameters for the ADXL345.
  - For example, the following configuration code will set the data rate to measure at 3200 Hz:

```
cs=0;
acc.write(0x2C);        // data rate register
acc.write(0x0F);        // set to 3200Hz
cs=1;
```

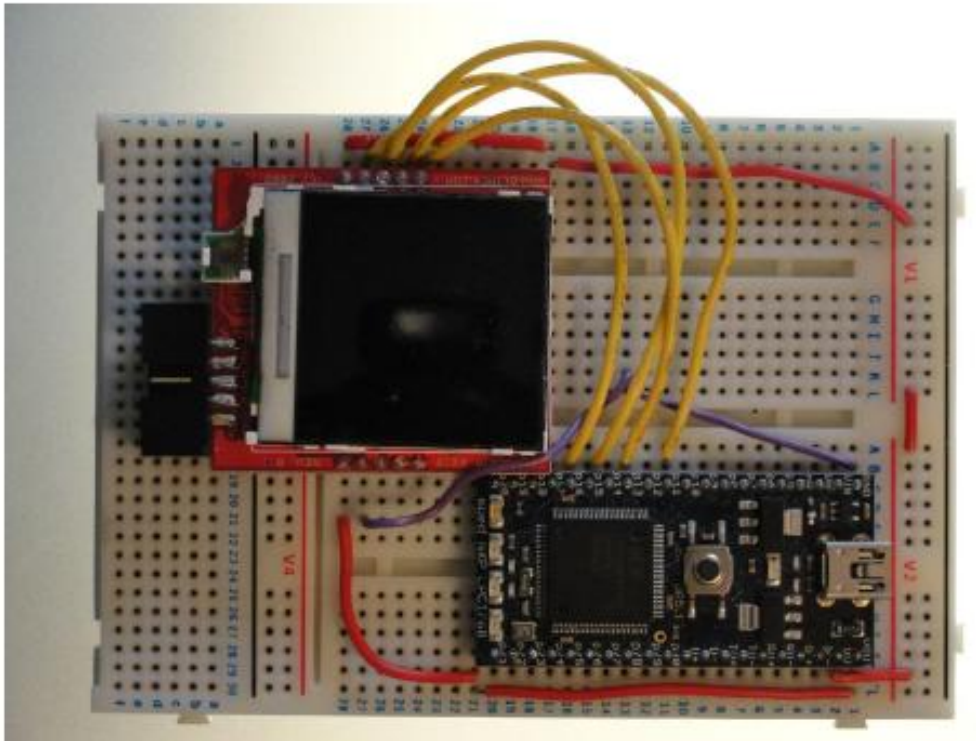
  - Similarly, the following configuration will set the measurement range to ±2g:

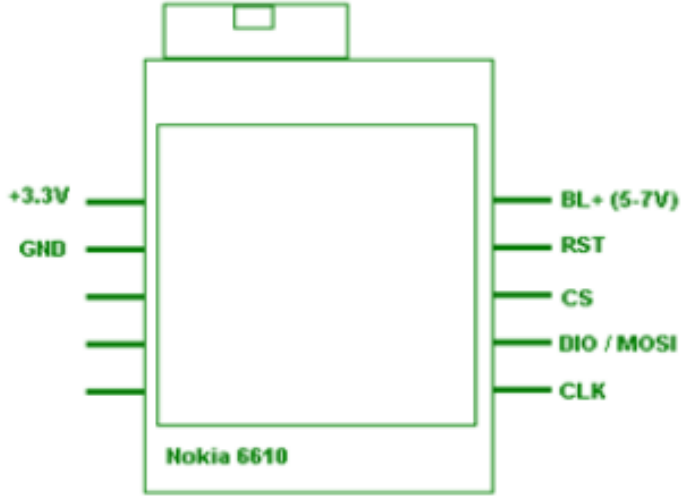
```
cs=0; // active low
acc.write(0x31);        // data format register
acc.write(0x08);        // format +/-2g, 0.004g/LSB
cs=1;
```

  - A number of other features including tap detection, freefall detection and threshold exceedance interrupts can also be configured with the ADXL345

# Interfacing the Nokia 6610 LCD display with the mbed

- The mobile screen has 10 pins and can be connected to the mbed as shown:






| Nokia 6610 Pin | Mbed Pin |
|---|---|
| +3.3V | 40 |
| GND | 1 |
| BL+ | 39 |
| RST | 16 |
| CS | 15 |
| MOSI | 11 |
| CLK | 13 |

# Interfacing the Nokia 6610 LCD display with the mbed

- **Exercise 3:** Display text typed into the terminal application on to the LCD screen. Use the # key to clear the screen of the text you have written.
  - You'll need to define the following screen_setup function to set a background colour and clear the screen :

```
void screen_setup(void) {          // define a function called screen_setup
    lcd.background(0x0000FF);      // set the background colour
    lcd.cls();                     // clear the screen
}
```

  - You'll also need to import the MobileLCD library from *http://mbed.co.uk/projects/cookbook/svn/MobileLCD/tests/MobileLCD*

  - Note also that the cursor can be moved to a chosen position to allow you to choose where to display data, for example:

```
lcd.locate(3,1);                   // move cursor to row 1 column 3
```

# Interfacing the Nokia 6610 LCD display with the mbed

```
// solution for SPI Exercise 3
#include "mbed.h"
#include "MobileLCD.h"                    //include the Nokia display library

MobileLCD lcd(p11, p12, p13, p15, p16); //mosi,miso,clk,cs,rst
Serial pc(USBTX, USBRX);                  // host terminal comms setup
char c;                                   // char variable for keyboard input
void screen_setup(void);                  // function proptotype

int main() {
    pc.printf("\n\rType something to be displayed:\n\r");
    screen_setup();                       // call the screen setup function

    while(1){
      c = pc.getc();                      // c = character input from computer keyboard
      wait(0.001);
      if (c=='#'){                        // perform the following if "#" is pressed
        screen_setup();                   // call the screen setup function
        lcd.locate(0,0);                  // move the cursor back to row 0 column 0
      }
      else{
        lcd.printf("%c",c);               // print character on the LCD screen
        pc.printf("%c",c);               // print character on the terminal screen
      }
    }
}

void screen_setup(void) {                 // define a function called screen_setup
    lcd.background(0x0000FF);             // set the background colour
    lcd.cls();                            // clear the screen
}
```

# Interfacing the Nokia 6610 LCD display with the mbed

- **Exercise 4:** Experiment with the fill and pixel commands to draw on the LCD Display. For example:
  - Use the following fill functions in order to fill some areas of the LCD display:

```
lcd.fill(2, 51, 128, 10, 0x00FF00);  //fill an area between the defined pixels
lcd.fill(50, 1, 10, 128, 0xFF0000);
```

  - Draw on the screen pixel by pixel. The following loop will create the function for a sine wave and print this to the screen:

```
for(int i=0; i<130; i++) {
    //draw a black pixel at the location described by the following formula
    lcd.pixel(i, 80 + sin((float)i / 5.0)*10, 0x000000);
}
```

  - Note the use of colour as a single 24-bit value for red, green and blue (8-bits each)

0xFF0000 = red 0x00FF00 = green 0x0000FF = blue 0x000000 = black 0xFFFFFF = white

# Interfacing multiple devices on a single SPI bus

- **Exercise 5:** You can use a single SPI data bus to control the ADXL345 accelerometer and the Nokia 6610 display at the same time. Write a program so that the x, y and z data appear and update on the LCD display, rather than on a host pc.

- You will need a separate slave chip select signal for each device.

# Digital spirit level design challenge

- **Exercise 6:** Design, build and test a digital spirit level based on the mbed microcontroller, the ADXL345 accelerometer and the Nokia 6610 display

- You may wish to consider the following:
  - Design your display to show a pixel or image moving around the LCD screen with respect to the orientation of the accelerometer
  - Improve your display output to include accurate measurements of the 2-plane orientation angles in degrees from the horizontal (i.e. horizontal = $0^0$ )
  - You may need to include some calibration or filtering techniques to ensure smooth and accurate functionality

# Summary

- Introducing SPI
- Evaluating simple SPI communications
- SPI on the mbed
- Evaluating the ADXL345 SPI accelerometer
- Interfacing the ADXL345 with the mbed
- Interfacing the Nokia 6610 display with the mbed
- Interfacing multiple devices on a single SPI bus
- Digital spirit level design challenge

# Serial Communications with SPI

Train The Trainer - IAC

# Timers and Interrupts

**Train The Trainer – IAC**

# Agenda

- Time and event management in embedded systems
- An introduction to timers
- Using the mbed Timer object
- Using multiple timers
- Using the mbed Ticker object
- Hardware interrupts
- External interrupts on the mbed
- Switch debouncing for interrupt control
- Extended exercises

# Time and event management in embedded systems

- Many embedded systems need high precision timing control and the ability to respond urgently to critical requests
- For example:
  - A video camera needs to capture image data at very specific time intervals, and to a high degree of accuracy, to enable smooth playback
  - A automotive system needs to be able to respond rapidly to a crash detection sensor in order to activate the passenger airbag
- Interrupts allow software processes to be halted while another, higher priority section of software executes
- Interrupt routines can be programmed to execute on timed events or by events that occur externally in hardware
- Routines executed by events that occur from an external source (e.g. a mouse click or input from another program) can be referred to as '**event driven**'.

# An introduction to timers

- Interrupts in embedded systems can be thought of as functions which are called by specific events rather than directly in code.
- The simplest type of interrupt is one which automatically increments a counter at a periodic interval, this is done behind the scenes while the software is operating.
- Most microcontrollers have built in timers or real-time-interrupts which can be used for this purpose.
- The main code can then be executed at specified time increments by evaluating the counter value.
- For example, we can set some pieces of software to operate every 10ms and others to operate every 100ms. We call this scheduled programming.

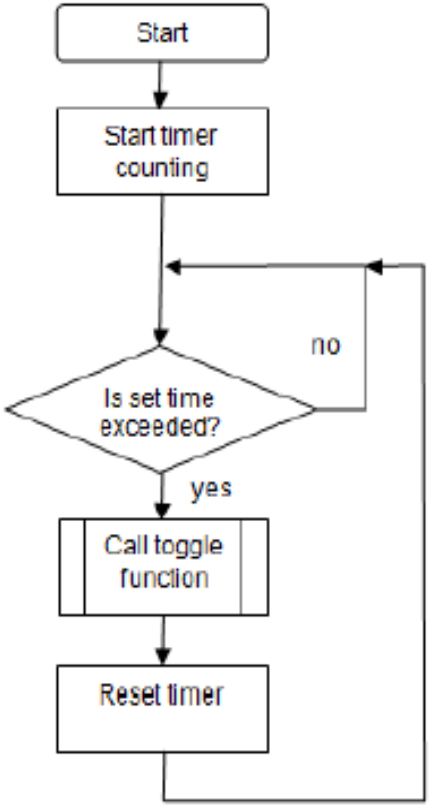
# Using the mbed Timer object

- We can use the mbed Timer object to perform scheduled programming:

| Timer | A general purpose timer |
|---|---|
| **Functions** | **Usage** |
| start | Start the timer |
| stop | Stop the timer |
| reset | Reset the timer to 0 |
| read | Get the time passed in seconds |
| read_ms | Get the time passed in mili-seconds |
| read_us | Get the time passed in micro-seconds |

# A simple timer routine

- **Exercise 1:** Create a square wave output using scheduled programming and verify the timing accuracy with an oscilloscope.




```
#include "mbed.h"

Timer timer1;            // define timer object
DigitalOut output1(p5); // digital output
void task1(void);        // task function prototype

//*** main code
int main() {
    timer1.start();      // start timer counting
    while(1) {
        if (timer1.read_ms()>=200)    // read time in ms
        {
            task1();                  // call task function
            timer1.reset();           // reset timer
        }
    }
}

void task1(void){           // task function
  output1=!output1;         // toggle output
}
```

# Using multiple timers

- With scheduled programs we often need to execute different sections of code at different rates.
- Consider an automotive system:
  - The engine spark, valve and fuel injection system needs to be controlled and executed at a high speed, perhaps every 1 ms or less given that the engine revolves at anything up to 8,000 revs per minute.
  - The fuel tank level monitoring system needs to report the fuel level less often, perhaps every 1000 ms is sufficient.
- There is no point in executing both the injection management and the fuel level management systems at the same rate.
- For this reason we can use synchronous programs to improve efficiency

# Using multiple timers

- **Exercise 2:** Add a second timer which will run at a different rate, you can use an LED or an oscilloscope on the mbed pins to check that the two timers are executing correctly.

```
//*** main code
int main() {
    timer1.start();     // start timer1 counting
    timer2.start();     // start timer2 counting
    while(1) {
        if (timer1.read_ms()>=200)   // read time
        {
            task1();          // call task1 function
            timer1.reset();          // reset timer
        }
        if (timer2.read_ms()>=1000)  // read time
        {
            task2();          // call task2 function
            timer2.reset();          // reset timer
        }
    }
}

// continued...
```

```
// ...continued

//*** task functions
void task1(void){
  output1=!output1;     // toggle output1
}

void task2(void){
  output2=!output2;     // toggle output2
}
```

- Note: You will need to define a second timer object, digital output and task function prototype.

# Challenges with timer interrupts

- With scheduled programming, we need to be careful with the amount of code and how long it takes to execute.

- For example, if we need to run a task every 1 ms, that task must take less than 1 ms second to execute, otherwise the timing would overrun and the system would go out of sync.
  - How much code there is will determine how fast the processor clock needs to be.
  - We sometimes need to prioritize the tasks, does a 1ms task run before a 100ms task? (because after 100ms, both will want to run at the same time).
  - This also means that pause, wait or delays (i.e. timing control by ‘polling’) cannot be used within scheduled program designs.

# Using the mbed Ticker object

- The Ticker interface is used to setup a recurring interrupt to repeatedly call a designated function at a specified rate.
- Previously we used the Timer object, which required the main code to continuously analyse the timer to determine whether it was the right time to execute a specified function.
- An advantage of the ticker object is that we don’t need to read the time, so we can execute other code while the ticker is running in the background and calling the attached function as necessary.

# Using the mbed Ticker object

- The mbed ticker object can also be used for scheduled programming.

| Ticker | A Ticker is used to call a function at a recurring interval |
|---|---|
| **Functions** | **Usage** |
| attach | Attach a function to be called by the Ticker, specifying the interval in seconds |
| attach | Attach a member function to be called by the Ticker, specifying the interval in seconds |
| attach_us | Attach a function to be called by the Ticker, specifying the interval in micro-seconds |
| attach_us | Attach a member function to be called by the Ticker, specifying the interval in micro-seconds |
| detach | Detach the function |

# Using the mbed Ticker object

- Exercise 3: Use two tickers to create square wave outputs.
- Use an LED or an oscilloscope on the mbed pins to check that the two tickers are executing correctly.

```
#include "mbed.h"
Ticker flipper1;
Ticker flipper2;
DigitalOut led1(p5);
DigitalOut led2(p6);

void flip1() {                          // flip 1 function
    led1 = !led1;
}
void flip2() {                          // flip 2 function
    led2 = !led2;
}

int main() {
    led1 = 0;
    led2 = 0;

    flipper1.attach(&flip1, 0.2); // the address of the
                                  // function to be attached
                                  // and the interval (sec)
    flipper2.attach(&flip2, 1.0);

    // spin in a main loop
    // flipper will interrupt it to call flip

    while(1) {
        wait(0.2);
    }
}
```

# Hardware interrupts

- Microprocessors can be set up to perform specific tasks when hardware events are incident.
- This allows the main code to run and perform its tasks, and only jump to certain subroutines or functions when something physical happens.
  - i.e. a switch is pressed or a signal input changes state.
- Interrupts are used to ensure adequate service response times in processing.
- The only real disadvantage of interrupt systems is the fact that programming and code structures are more detailed and complex.

# External interrupts on the mbed

- External interrupts on the mbed:

| InterruptIn | A digital interrupt input, used to call a function on a rising or falling edge |
|---|---|
| **Functions** | **Usage** |
| InterruptIn | Create an InterruptIn connected to the specified pin |
| rise | Attach a function to call when a rising edge occurs on the input |
| rise | Attach a member function to call when a rising edge occurs on the input |
| fall | Attach a function to call when a falling edge occurs on the input |
| fall | Attach a member function to call when a falling edge occurs on the input |
| mode | Set the input pin mode |

- Note: any digital input can be an interrupt except pin 19 and pin 20

# External interrupts on the mbed

- **Exercise 4:** Use the mbed InterruptIn library to toggle an LED whenever a digital pushbutton input goes high.

```
#include "mbed.h"

InterruptIn button(p18);     // Interrupt on digital pushbutton input p18
DigitalOut led1(p5);         // digital out to p5

void toggle(void);           // function prototype

int main() {
    button.rise(&toggle);    // attach the address of the toggle
}                                        // function to the rising edge

void toggle() {
    led1=!led1;
}
```

- You may notice some issues with this simple program, what are they?

# Switch debouncing for interrupt control

- Exercise 4 doesn't work quite as expected; it is possible for the button to become unresponsive or out of synch with the LED.
- This is because of a common issue called switch or button bouncing. When the button is pressed it doesn't cleanly switch from low to high, there is some 'bounce' in between as shown below:


- It is therefore easy to see how a single button press can cause multiple interrupts and hence the LED can get out of synch with the button.
- We therefore need to 'debounce' the switch with a timer feature.

# Switch debouncing for interrupt control

- **Exercise 5:** Use the mbed InterruptIn library to toggle an LED whenever a digital input goes high, implementing a debounce counter to avoid multiple interrupts.

```
#include "mbed.h"

InterruptIn button(p18);     // Interrupt on digital pushbutton input p18
DigitalOut led1(p5);         // digital out to p5
Timer debounce;              // define debounce timer

void toggle(void);           // function prototype

int main() {
    debounce.start();
    button.rise(&toggle);       // attach the address of the toggle
}                                           // function to the rising edge

void toggle() {
if (debounce.read_ms()>200)     // only allow toggle if debounce timer
    led1=!led1;                             // has passed 200 ms
    debounce.reset();           // restart timer when the toggle is performed
}
```

# Switch debouncing for interrupt control

- By removing the bounces, the system acts as though the switch has an ideal response.
- This can be done by a variety of hardware and software methods.


- An example of a classic hardware debouncer would be two cross-coupled NAND gates form a very simple Set-Reset (SR) latch.
- Another example of a software debouncer would be to look for a number of sequential readings of the switch, e.g. if the input changes from 0 to 1 and then continues to read 1 for the next ten samples then the switch has been pressed.

# Extended exercises

- **Exercise 6:** Using an oscilloscope evaluate the debounce characteristic of your pushbutton. What is the ideal debounce time for your pushbutton? Note that longer debounce times reduce the capability for fast switching, so if fast switching is required a different type of pushbutton might be the only solution

- **Exercise 7:** Combine the timer and hardware interrupt programs to show that a scheduled program and an event driven program can operate together. Flash two LEDs at different rates but allow a hardware interrupt to sound a buzzer if a pushbutton is pressed.

- **Exercise 8:** Accelerometer chips such as the ADXL345 have interrupt output flags to enable an interrupt based on an excessive acceleration (as used in vehicle airbag systems). Investigate and experiment with the ADXL345 interrupt feature to sound a buzzer when a high impact is seen.

# Summary

- Time and event management in embedded systems
- An introduction to timers
- Using the mbed Timer object
- Using multiple timers
- Using the mbed Ticker object
- Hardware interrupts
- External interrupts on the mbed
- Switch debouncing for interrupt control
- Extended exercises

# Timers and Interrupts

**Train The Trainer - IAC**

# Memory and data management

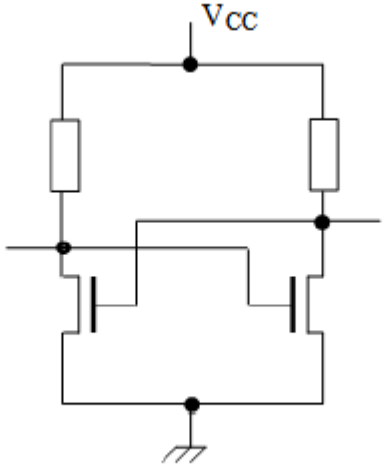
Train The Trainer – IAC

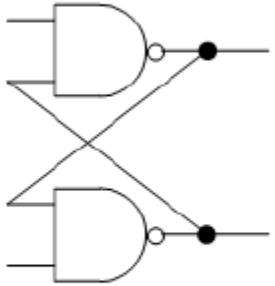
# Agenda

- Electronic memory types
- Volatile memory
- Non-volatile memory
- Memory function types
- Using data files with the mbed
- Using formatted data
- Extended exercises

# Electronic memory types

- A simple one-bit memory is a coin
  - It is “bistable”, i.e. stable in two positions, with either “heads” facing up, or “tails”
  - Consider “heads” represents logic 1, and “tails” logic 0
  - With 8 coins, an 8-bit number can be represented and stored.
- An alternative to the coin is to use an electronic bistable (“flip-flop”) circuit
  - Circuits b and c are also stable in only two states, each can be used to store one bit of data

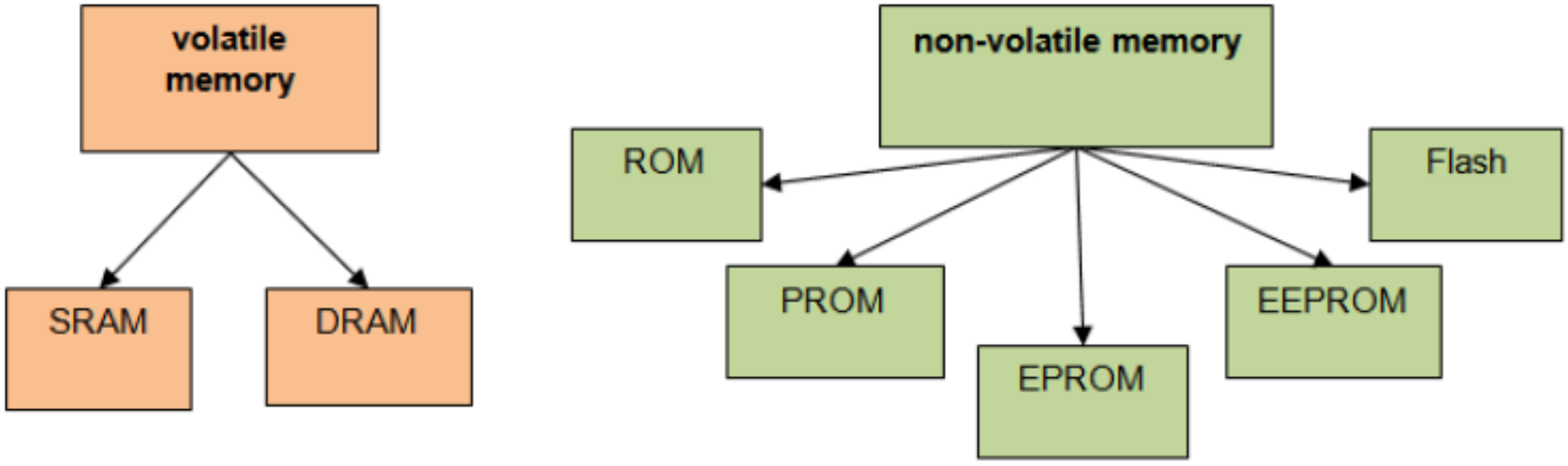


a) a coin b) two transistors and two resistors c) two NAND gates

# Electronic memory types

- Volatile memory requires power to maintain the stored information.
- Non-volatile memory can maintain stored information, even without a power supply.



# Volatile memory

- **Random Access Memory (RAM**) is an example of volatile memory

- **SRAM (Static RAM**) stores data using the state of a flip flop.
  - There are usually six transistors in a single SRAM unit
  - It is low power, easy to interface and relatively inexpensive

- **DRAM (Dynamic Ram**) uses one transistor and one capacitor to store one bit of data
  - DRAM can therefore take up a reduced silicon area
  - But it also requires refresh logic to recharge the capacitor every 10-100 ms
  - Power requirements are also higher in relation to SRAM

# Non-volatile memory

- **Read Only Memories (ROM)** and **PROMs (Programmable ROM)** can only ever be programmed once.
- **Electrically Programmable Read Only Memory (EPROM)** is non-destructive using a trapped charge to represent a stored memory value.
  - The programming and erasing process is quite involved however, requiring a high voltage to program and a UV light source to erase
- With **Electrically Erasable and Programmable Read Only Memory (EEPROM),** bytes of data are individually writeable, readable and erasable.
- **Flash** memory, is a type of EEPROM but without the ability to erase individual data bytes
  - Multiple bytes have to be erased at any one time, ‘in a flash’. This leads to flash memory having a very high density and low cost.
  - Flash and EEPROM wear out however and can only handle approximately 100,000 write-erase cycles

# Memory function types

- Microprocessors use memory for holding the program code (program memory) and the working data (data memory) in an embedded system
- When the processor is powered up the program data needs to be there and ready. Non-volatile memory is generally required for program memory
- There is often no need to retain data when the product is switched off, therefore volatile memory is traditionally preferred for data memory

# Using data files with the mbed

- With C/C++ we can open files, read and write data and also scan through files to specific locations, even searching for particular types of data.

- We can store data in files (as chars) or we can store words and strings (as character arrays).

- We can save data files to a specific area of flash memory installed on the mbed. This memory is a page-erase data flash IC which is external to the LPC1768 microprocessor.

- When interfacing the mbed data memory, we use the standard C/C++ commands as defined by the C Standard Input and Output Library (stdio.h).

# Using data files with the mbed

| Function | Format | Summary Action |
|---|---|---|
| fopen | `FILE * fopen ( const char * filename, const char * mode);` | opens the file of name filename |
| fclose | `int fclose ( FILE * stream );` | closes a file |
| fgetc | `int fgetc ( FILE * stream );` | gets a character from a stream |
| fgets | `char * fgets ( char * str, int num, FILE * stream );` | gets a string from a stream |
| fputc | `int fputc ( int character, FILE * stream );` | writes a character to a stream |
| fputs | `int fputs ( const char * str, FILE * stream );` | writes a string to a stream |
| fseek | `int fseek ( FILE * stream, long int offset, int origin );` | moves file pointer to specified location |

A summary of useful stdio functions for implementing file access on the mbed

str - An array containing the null-terminated sequence of characters to be written.

stream - Pointer to a FILE object that identifies the stream where the string is to be written.

# Using data files with the mbed

- The compiler must be told where to store data files. This can be done by using the mbed ‘LocalFileSystem’ declaration:

```
LocalFileSystem local("local");
```

- This allows programs to read and write files on the same disk drive that is used to program the mbed Microcontroller.

- Once created, the standard C file access functions are used to open, read and write files.

# Using data files with the mbed

- We can open a file with the following command:

```
FILE* pFile = fopen("/local/datafile.txt","w");
```

- This assigns a pointer with a shorthand name (pFile) to the file at the specific location given in the fopen statement.
- We also tell the compiler whether we want read or write access to the file. This is done by the “w” syntax. If we put “r” this would make the file open as read only.
- Note also that if the file doesn’t already exist the fopen command will automatically create the file in the specified location.
- When we have finished using a file for reading or writing it is good practice to close the file, for example:

```
fclose(pFile);
```

# Using data files with the mbed

## Writing data to a file

- If we are looking to store numerical data we can do this in a simple way by storing individual 8-bit data values (chars). The fputc command allows this, as follows:

```
char write_var=0x0F;
fputc(write_var, pFile);
```

- This stores the variable write_var to the data file.

## Reading data from a file

- We can read data from a file to a variable as follows:

```
read_var = fgetc(pFile);
```

- Using the stdio.h commands it is also possible to read and write words and strings and search or move through files looking for particular data elements.

# Using data files with the mbed

- **Exercise 1:** Compile a program that creates a data file and writes the value 0x23 to that file. The file is saved on the mbed. The program then opens and reads back the data value and displays it to the screen in a terminal application.
- Open the created file datafile.txt in a word processor, you should see a hash character (#) in the top left corner (the ASCII character for 0x23 is the hash character).

```
#include "mbed.h"
Serial pc(USBTX,USBRX);              // setup terminal link
LocalFileSystem local("local");      // define local file system
int write_var;
int read_var;                        // create data variables

int main ()
{
  FILE* File1 = fopen("/local/datafile.txt","w");       // open file
  write_var=0x23;                                       // example data
  fputc(write_var, File1);                              // put char (data value) into file
  fclose(File1);                                        // close file

  FILE* File2 = fopen ("/local/datafile.txt","r");      // open file for reading
  read_var = fgetc(File2);                              // read first data value
  fclose(File2);                                        // close file
  pc.printf("input value = %i \n",read_var);            // display read data value
}
```

# Using data files with the mbed

- **Exercise 2:** Compile a program that creates a file and writes text data to that file. The file is saved on the mbed. The program then opens and reads back the text data and displays it to the screen in a terminal application.

```
#include "mbed.h"
Serial pc(USBTX,USBRX);                    // setup terminal link
LocalFileSystem local("local");            // define local file system
char write_string[64];                     // character array up to 64 characters
char read_string[64];                      // create character arrays(strings)
int main ()
{
  FILE* File1 = fopen("/local/textfile.txt","w");      // open file access
  fputs("lots and lots of words and letters", File1);  // put text into file
  fclose(File1);                                       // close file

  FILE* File2 = fopen ("/local/textfile.txt","r");     // open file for reading
  fgets(read_string,256,File2);                        // read first data value
  fclose(File2);                                      // close file
  pc.printf("text data: %s \n",read_string);           // display read data string
}
```

- Open the file textfile.txt, you should see the correct text data.

# Using formatted data

- The fprintf command is used to format data when working with files.
- fprintf has very similar syntax to printf, except that the filename pointer is also required.
- An example of fprintf is to log specific events to a data file and include variable data values such as time, sensor input data and output control settings.
- **Exercise 3** uses the fprintf statement in an interrupt controlled toggle switch project. Each time the toggle switch pushbutton is pressed, the LED changes state. Also on each toggle event the file log.txt is updated to include the time elapsed since the previous toggle button press and the current LED state.
- **Exercise 3** also implements a simple debounce timer to avoid multiple interrupts and file write operations.

# Using formatted data

- **Exercise 3:** Compile a program that uses fprintf statement in an interrupt controlled toggle switch project. On each toggle event and LED switches state and the file log.txt is updated to include the time elapsed since the previous toggle button press and the current LED state.

```
#include "mbed.h"
InterruptIn button(p30);           // Interrupt on digital input p30
DigitalOut led1(LED1);             // digital out to onboard LED1
Timer debounce;                    // define debounce timer
LocalFileSystem local("local");    // define local file system
void toggle(void);                 // function prototype
int main() {
  debounce.start();                // start debounce timer
  button.rise(&toggle);            // attach the toggle function to the rising edge
}
void toggle() {                    // perform toggle if debounce time has elapsed
  if (debounce.read_ms()>200)  {
    led1=!led1;                                        // toggle LED
    FILE* Logfile = fopen ("/local/log.txt","a");      // open file for appending
    fprintf(Logfile,"time=%.3fs: setting led=%d\n\r",debounce.read(),led1.read());
    fclose(Logfile);                                   // close file
    debounce.reset();                                  // reset debounce timer
  }
}
```

# Extended exercises

- **Exercise 4:** Create a program which prompts the user to type some text data into a terminal application. When the user presses return the text is captured and stored in a file on the mbed.
- **Exercise 5:** Create a program which records 5 seconds of analogue data to the screen and a text file. Use a potentiometer to generate the analogue input data.
  - You will need to specify a sample period to capture and store an analogue value every, say, 100 ms. Ensure that your data file records time and voltage data, you can then plot a chart of the measured data to visualise the input readings.
- **Exercise 6:** Create a new project similar to that in Exercise 5, but replace the analogue input with a sensor (e.g. a temperature sensor or accelerometer). Using a suitable sample frequency, log actual sensor data to a formatted data file.

# Summary

- Electronic memory types
- Volatile memory
- Non-volatile memory
- Memory function types
- Using data files with the mbed
- Using formatted data
- Extended exercises

# Memory and data management

Train The Trainer - IAC