ENGINEERING @ SP

ET1010 / ET0884

MICROCONTROLLER APPLICATIONS

(Version 17.1)

School of Electrical & Electronic Engineering

ENGINEERING @ SP

The Singapore Polytechnic's Mission

As a polytechnic for all ages
we prepare our learners to be
life ready, work ready, world ready
for the transformation of Singapore

The Singapore Polytechnic's Vision

Inspired Learner. Serve with Mastery. Caring Community.

A caring community of inspired learners committed to serve with mastery.

The SP CORE Values

- Self-Discipline
- Personal Integrity
- Care & Concern
- Openness
- Responsibility
- Excellence

For any queries, please contact:

Name: Dr. M Fikret Ercan
Room: T827
Email: M_Fikret_ERCAN@sp.edu.sg
Tel: 68790693

SINGAPORE POLYTECHNIC | SP

SP SCHOOL OF ELECTRICAL & ELECTRONIC ENGINEERING

Industry-Collaborated TECH HUB

Diploma in Aerospace Electronics

Aerospace Engineering Hub

Themed Project : UAV-augmented Smart Environment
Facilities : A320 Flight Simulator
Industries : Aerospace, Air Transport
Industry Partners : AIRBUS, Rolls-Royce
Industry Co-location Lab : ST Engineering Aerospace

Diploma in Computer Engineering

5G & AIoT Technology Hub

Themed Project : Smart Building & Facilities
Facilities : AIoT Nerve Centre, 5G Garage
SP Connected Solutions Lab
Industries : ICT & Media, Smart City
Industry Partners : ADVANTECH, CONTEC, ENVISION, HUAWEI
Industry Co-location Lab : BOSCH Invented for life, ERICSSON, Singtel

Diploma in Engineering with Business

Technology to Business Hub

Themed Project : Technology To Business Venture
Facilities : IdeaBox, T2B Hub
Industries : Entrepreneurship & Start-up
Industry Co-location Lab : SPgroup, GEX VENTURES

APOL, BevEat, ItsGreat, Codesurance, OCTOPUS8, Trackdat, REAL IMPACT, SMARTDB, TOP TASTE, Vaster

T18A

T9 level 3 & T14 level 3

T14 level 3

T10 level 3

T8 level 2 & T12 level 1

T12/T12A level 2

T12 level 6

T8 level 2 & T12 level 1

Diploma in Electrical & Electronic Engineering

Robotics, Automation & Control Hub

Themed Project : Smart Workflow Solution for I4.0
Facilities : SP-SICK Smart Sensor Lab, SP-Rexroth I4.0 Innovation Lab
Industries : Logistics, Electronics (Manufacturing), Electronics & Chemicals (Manufacturing)
Industry Partners : ABB, BECKHOFF, SEW EURODRIVE, Weidmüller, ZEBRA
Industry Co-location Lab : rexroth A Bosch Company, SICK Sensor Intelligence.

Power & Sustainable Energy Hub

Themed Project : Zero-Energy Space
Facilities : High Voltage Training, LE & LET Training, Energy Efficiency Lab
Industries : Energy & Power
Industry Partners : CHINT, ebmpapst The engineer's choice, SUNLIGHT A CHINT COMPANY

Semicon & Electronics Hub

Themed Project : Semicon-Enabled AI Applications
Facilities : Nanofab & IC Design Labs, Infineon Innovation Lab
Industries : Semiconductor & Electronics
Industry Co-location Lab : Infineon, LG, SSIA

Biomedical Engineering Hub

Themed Project : Engaging Rehabilitation
Facilities : Biomedical Engineering Lab
Industries : Healthcare
Industry Partners : GE Healthcare, GOLD LITE, Khoo Teck Puat Hospital, NATIONAL INSTRUMENTS

Autonomous & Transportation Systems Hub

Themed Project : Autonomous EV / Rapid Simulator System
Facilities : RSS Lab for RTT Training, Siemens Digital Experience Centre, SP-TIBCO Adv Mfg Analytics Lab
Industries : Smart Transport, Land Transport
Industry Partners : ST Engineering Land Systems, SMRT, SIIX, SUTD, GreenPower Accredited Renewable Energy
Industry Co-location Lab : Land Transport Authority, SIEMENS Ingenuity for life, TIBCO

CONTENTS

Module Overview

1 Introduction

This module uses Microchip PIC18 microcontroller to introduces basic microcontroller concepts such as IO ports and device interfacing, ADC, Timers, Interrupts and Serial Port.

Students are assumed to know how to write simple programs in C language.

2 Module Aims

This module introduces the use of Microcontrollers for applications. Students are taught how a microcontroller works, the programming and use of the microcontroller for applications. In addition, the student will learn basic analogue and digital support circuitry, sensors and actuators/displays required for a microcontroller-based application. This module allows students to develop a project conceived around a microcontroller system with sensors and output devices.

Chapter 1 – Introduction to Micro-controller

Chapter Overview

1.1 What is a micro-controller?
1.2 Main parts of a micro-controller
1.3 How is a micro-controller different from a micro-processor?
1.4 Common characteristics of a micro-controller
1.5 Typical applications of a micro-controller
1.6 Embedded systems
1.7 A simple application – "pedestrian crossing traffic light control" (scenario + block diagram + flowchart)
1.8 The development tools and programming languages for a micro-controller
1.9 The topics to be covered in subsequent chapters

1.1 What is a micro-controller?

- According to Wikipedia, the free encyclopedia, a micro-controller (MCU or μC) is **a small computer on a single integrated circuit (IC) consisting internally of a relatively simple CPU, clock, timers, I/O ports and memory** (program memory in the form of flash, and a small amount of data memory in the form of RAM).

- Micro-controllers are designed for **small or dedicated applications**.

- Thus, in contrast to the micro-processor used in personal computers (PC) and other high-performance or general purpose applications, **simplicity** is emphasized in micro-controllers.

- Some micro-controllers may use four-bit words and operate at clock rate as low as 4 kHz, as this is adequate for many typical applications, enabling **low power consumption** (milliwatts or microwatts)

- They will generally have the ability to **retain functionality while waiting for an event** such as a button press or other interrupt; power consumption while sleeping (CPU clock and most peripherals off) may just be nanowatts, making many of them well suited for **long lasting battery applications**.

- Other micro-controllers may serve performance-critical roles, where they may need to act more like a **digital signal processor** (DSP), with higher clock speeds and power consumption.

- Micro-controllers are used in **automatically controlled products and devices**, such as automobile engine control systems, implantable medical devices, remote controls, office machines, appliances, power tools and toys.

- By reducing the **size and cost** compared to a design that uses a separate **micro-processor, memory, and input/output devices**, micro-controllers make it economical to digitally control even more devices and processes.

- **Mixed signal** micro-controllers are common, integrating **analogue components** needed to control non-digital electronic systems.

1.2 Main parts of a micro-controller

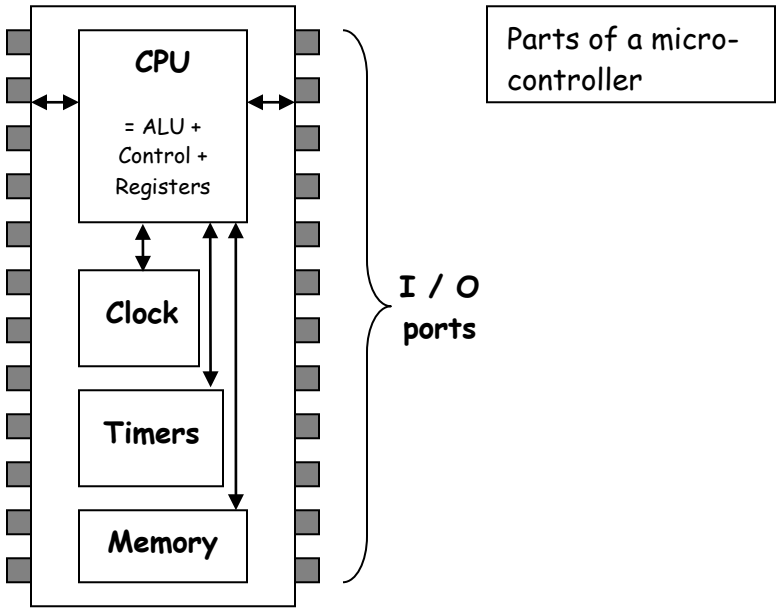


Figure 1.1 – Main parts of a micro-controller

- The **memory** holds the instructions or data. The memory is typically RAM (random access memory – which is volatile i.e. data is lost when power is turned off) and / or FLASH (a type of EEPROM – non-volatile, and can be "block erased")

- The **I/O ports** allow interactions with the outside world.

- The **CPU** has logic that fetches instruction & data, executes the instruction and stores the result back to memory.

1.3 How is a micro-controller different from a micro-processor?

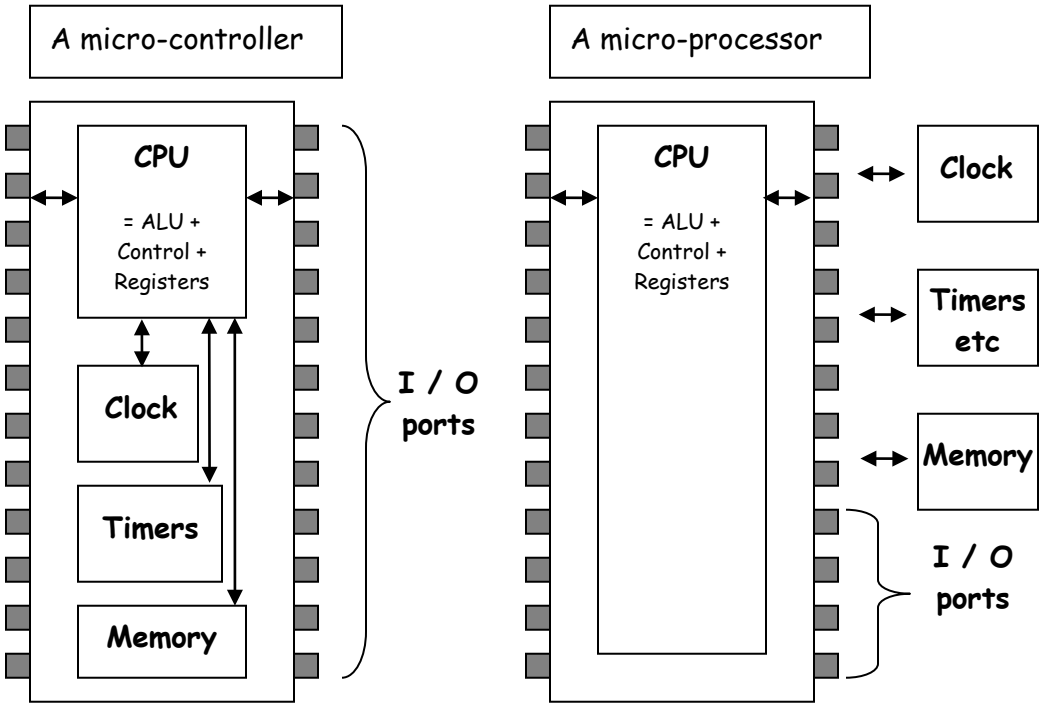


Figure 1.2 – Micro-controller vs. micro-processor

- As can be seen from the diagram above, a **micro-processor** often requires **other components** to form a functional system e.g. external clock source (in the form of a crystal oscillator *), peripheral devices such as timers, and memory.

- Thus, a **micro-controller** based system is often **smaller** in **size** and **lower** in **cost**.

- As mentioned in a previous section, **micro-processors** are used in **personal computers** (PC) which are "**general purpose**" systems, while **micro-controller** are often used in **dedicated** systems.

- (*) Some micro-controllers come with **on-chip oscillators**, while others require external oscillators.

- The figure below shows a **desk top PC** (Personal Computer) with the monitor as output and the keyboard and mouse as inputs. Inside the **CPU** (Central Processing Unit) is a "**mother board**" – a PCB (Printed Circuit Board) with a

micro-processor (e.g. Intel or AMD), some peripheral devices (e.g. memory, timer) and connections for the various inputs and outputs. The various **programs** running on the micro-processor – operating system, web browser, email software, word processor, spreadsheet, games – allows the PC to be used for a variety of purposes.

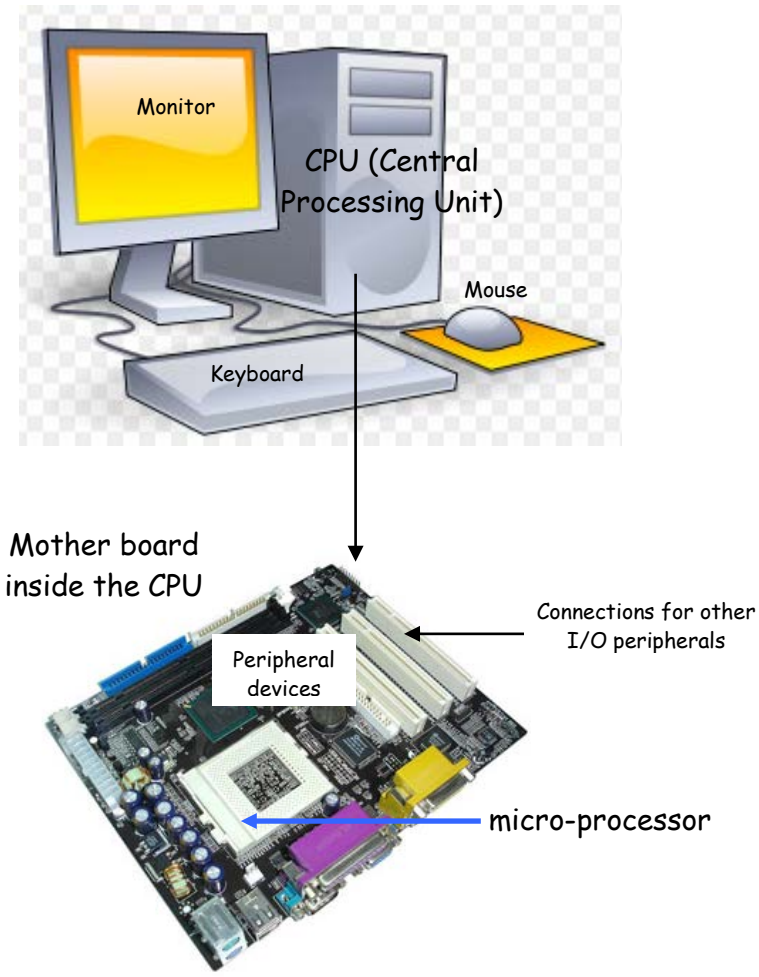


Figure 1.3 – Micro-processor as part of a personal computer

1.4 Common characteristics of a micro-controller

☐ According to HowStuffWorks.com, micro-controllers share the following characteristics:

☐ Micro-controllers are **embedded** inside some other device (often a consumer product) so that they can control the features or actions of the product. Another name for a micro-controller, therefore, is "embedded controller".

☐ Micro-controllers are **dedicated** to one task and run one specific program. The program is stored in non-volatile memory (e.g. ROM / flash) and generally does not change.

☐ Micro-controllers are often **low-power** devices. A desktop PC might consume 50 watts of electricity but a battery-operated micro-controller might consume 50 milliwatts.

☐ A micro-controller works with fixed **input and output devices**. It reads the status of the input devices and sends control signals to the output devices. For example, a microwave oven controller takes input from a keypad, displays output on an LCD display and controls a relay that turns the microwave generator on and off.

☐ A micro-controller is often **small** and **low cost**. The components are chosen to minimize size and to be as inexpensive as possible.

☐ Compare the various micro-controller characteristics above with the micro-processor inside a desktop PC. How are they different? 

	Micro-controller	Micro-processor inside a desktop PC
Dedicated or general purpose		
Power consumption		
Input and output devices		
Size		
Cost		

1.5 Typical applications of a micro-controller

☐ Micro-controllers are found in almost all "**smart**" electronic devices.

☐ From microwave ovens to automotive braking systems, they are around us doing jobs that make our lives more convenient and safer.

☐ Typically, a micro-controller receives **inputs** from sensors (e.g. a temperature sensor), makes some intelligent **decisions** (e.g. is the temperature too high?), and then drives some **outputs** / mechanism to cause something to happen (e.g. turn on a fan).

☐ Look around you and name several devices where micro-controller may be found: 

Applications of micro-controller
1.
2.
3.

☐ Suppose you have been asked to design an "intruder alert gadget" to protect a home against intruders while the owner is away. Before leaving home the owner can activate the gadget by pressing a button. He must then leave within a minute. Upon returning home, he opens the door and triggers a "motion sensor", the gadget will prompt him to key in the correct password within 30 seconds, failing which an alarm will be activated. For this gadget, which of the following input / output devices will be required? Please tick.

Input / output devices required for "intruder alert gadget"

- ☐ Some buttons
- ☐ Some LED's
- ☐ A motion sensor
- ☐ An LCD
- ☐ A keypad
- ☐ A buzzer

1.6 Embedded systems

☐ An embedded system is a computer system designed to perform one or a few **dedicated functions**, often with real-time computing constraints i.e. the system must respond in time to external stimuli. The system often includes hardware and **mechanical parts**.

☐ By contrast, a general-purpose computer, such as a PC, is designed to be flexible and to meet a wide range of end-user needs.

☐ Embedded systems are controlled by a micro-controller or a digital signal processor (DSP).

☐ Since an embedded system is dedicated to handle a particular task, design engineers can **optimize** it reducing the size and cost of the product and increasing the reliability and performance.

☐ Physically, embedded systems range from portable devices such as MP3 players, to large stationary installations like traffic lights.

1.7 A simple application – "pedestrian crossing traffic light control"

☐ A pedestrian crossing is shown below. A micro-controlled based embedded system is required to control the traffic and pedestrian lights.

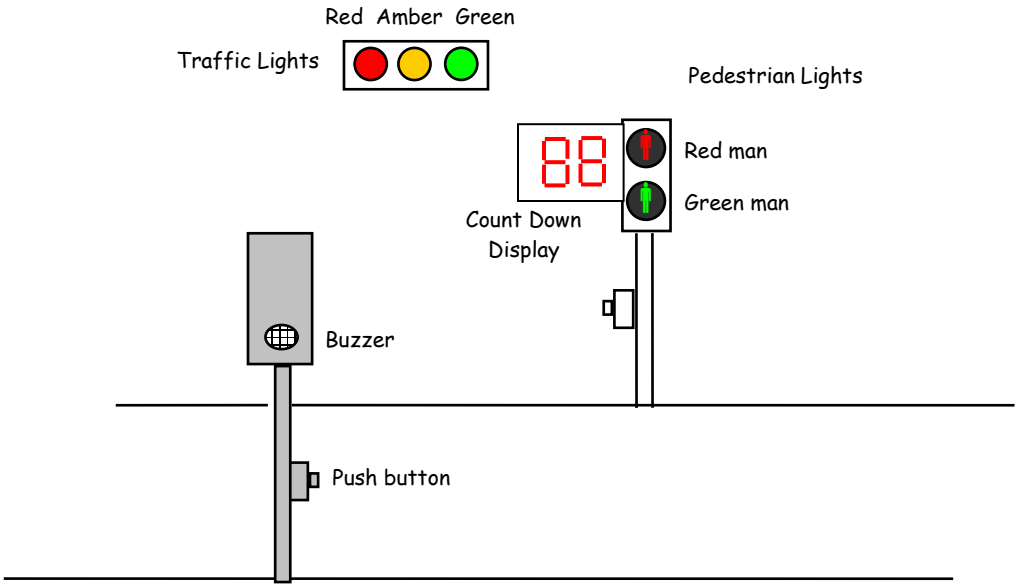


Figure 1.4 – Pedestrian crossing traffic light control

☐ Can you name the inputs to and outputs from the system?

Inputs	Outputs
1.	1.
	2.
	3.

☐ Complete the **block diagram** of the traffic light control below:

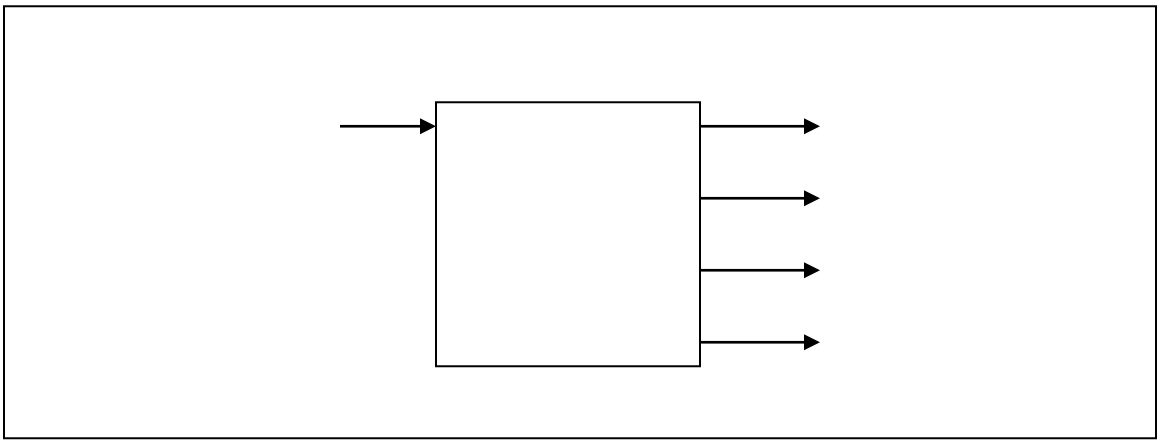


Figure 1.5 – Block diagram for traffic light control

☐ Complete the **flow chart** of the traffic light control below:

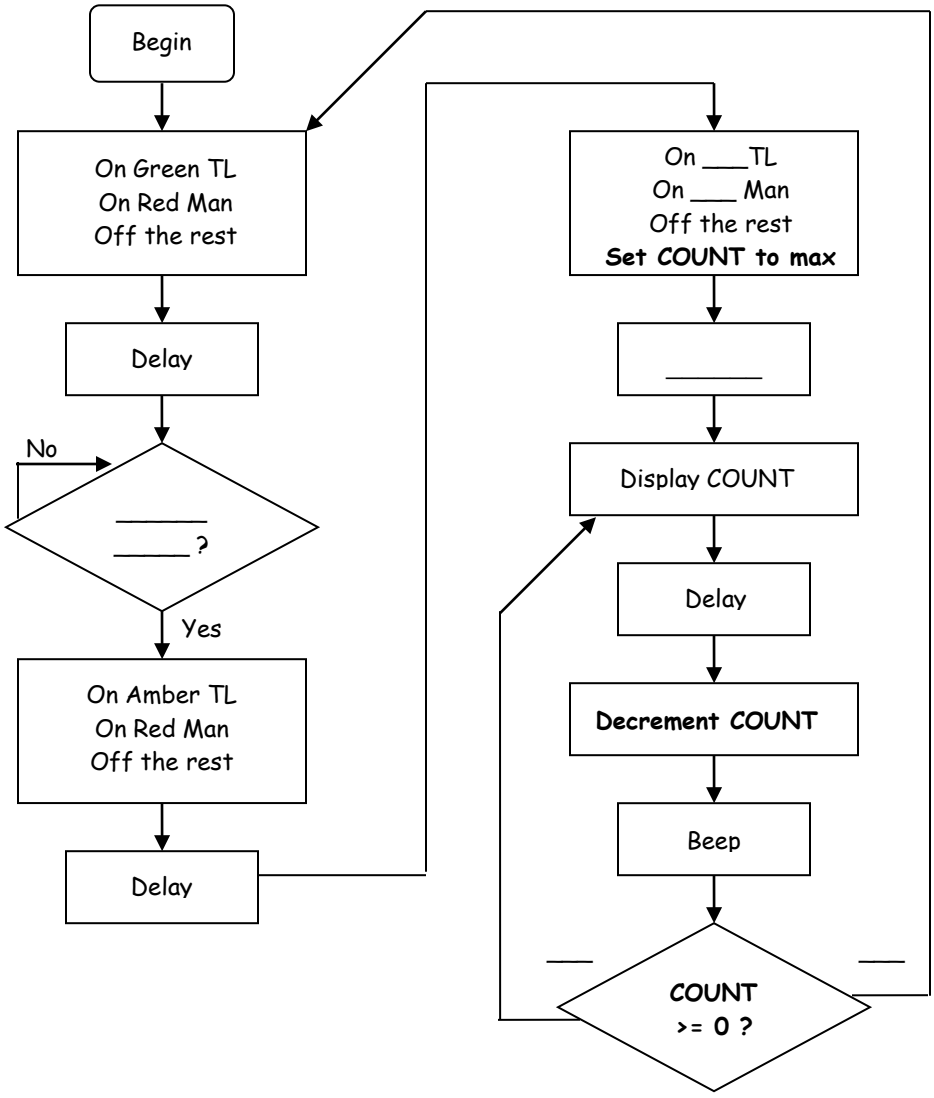


Figure 1.6 – Flow chart for traffic light control

1.8 The development tools and programming languages for a micro-controller

- To develop a "micro-controller-based electronic system", both **hardware** and **software** development work are involved.

- In terms of hardware development, the designer/engineer first understands / selects /procures suitable **input/output devices** (i.e. sensors / actuators) as well as a **micro-controller**.

- He then designs an electronic **circuit**, perhaps with the use of a **CAD** (Computer-Aided-Design) tool (which allows him to captures the **schematic**, performs some **simulation**, and then **layout** the PCB i.e. Printed Circuit Board) and **prototypes** it (on a "**bread-board**", or on a "**strip-board**", or on a **PCB**). If several **sub-circuits** are used, some **interfacing** may be required.

- However, a micro-controller cannot work unless it is "programmed". To **program** the micro-controller, an **IDE** (Integrated Development Environment) is first used to create a "**project**" i.e. to specify the name of the project, what micro-controller will be used, what programming language will be used…

- A program is then written, using either a "**high-level language**" e.g. C-language (which you learnt in another module e.g. Structured Programming) or an "**assembly language**".

- A program written in a high-level language is easier to understand, for instance, you know what this means:

```
if (temp > 30)
        on_motor;
```

- On the other hand, a segment of assembly code would look like "Greek" to most of us:

```
BSF     03h, 5      ; go to Bank 1
MOVLW   06h         ; put 00110 into W
MOVWF   85h         ; move 00110 into TRISA
BCF     03h, 5      ; come back to Bank 0
```

- The micro-controller cannot understand the high-level language or the assemble language – it can only comprehend "**machine code**" i.e. a string of 1's and 0's e.g. 10101011111010100000100101110…….

- On a computer, the string of 1's and 0's are stored as hexadecimal numbers i.e. in a **hex file**.

- So, the program written in high level language must first be "compiled" i.e. "converted" into the machine code using a "**Compiler**".

- Likewise, the program written in assembly language must first be "assembled" into the machine code using an "**Assembler**".

- The machine code is then "**downloaded**" (usually via some cable) from the PC into the micro-controller memory.

- Even with the circuits and program ready, the electronic system seldom works the first time. So, some "**trouble-shooting**" will usually be required.

1.9 The topics to be covered in subsequent chapters

- In the following chapters you will learn the **more useful features** of a micro-controller "**PIC18F4550**" from **Microchip**.

- Some useful features we will cover include **I/O ports** and input/output **device interfacing**, **analogue to digital converter**, how to use the **C-18 compiler**, **timer**, **interrupt** etc.

- During the practical lessons, a series of **experiments** will be carried out to familiarise you with those common features.

- After that, you will also work in a **team** to **conceive / design / implement** a useful and interesting micro-controller application **project**.

- As it is difficult for students (especially those new to micro-controller) to fabricate a **micro-controller board** that works, pre-fabricated boards will be made available – you only fabricate the input/output circuits required for your particular project, and interface these to the micro-controller board. Of course, you have to write the program.

- For your info, many engineers working in the industry also buy "**prototyping kits**" i.e. pre-fabricated micro-controller boards to kick-start their projects. They may fabricate their own customized boards at a later stage in the project.

Chapter 2 – Microchip's PIC18F4550 – an Overview.

Chapter Overview

2.1 Why Microchip's PIC18F4550?
2.2 Key features of PIC18F4550
2.3 Variations of PIC18F4550
2.4 Block diagram & on-chip peripherals
2.5 Pin diagram
2.6 A basic PIC-based circuit
2.7 A simple application – "zebra crossing light control" (scenario + block diagram + flowchart + circuit diagrams of I/O devices + C-program)
2.8 The steps to create a project, to write & compile a C program and to download a hex file into the PIC18F4550.

2.1 Why Microchip's PIC18F4550?

☐ If you **Google** "microcontroller", you will discover that many manufacturers are making thousands of different micro-controllers.

☐ Among the more **popular** ones are 68HC11 (by Motorola), 8051 (by Intel), ARM processors, Hitachi H8, various PIC's (by Microchip), PowerPC (by Apple-IBM-Motorola alliance) etc.

☐ It is not easy to decide which micro-controller is the best for a course or for use in a project.

☐ What would be your **key considerations** in choosing a micro-controller for project? 

The micro-controller should be

☐ _____

☐ _____

☐ _____

☐ _____

☐ In this course, we have decided to use **Microchip's PIC18F4550** for a few reasons:

- It can be **programmed by a USB cable**, so we can do away with expensive programmer. After all, it costs less than $1 to make a USB cable.
- The **software tools** IDE (Integrated Development Environment), the C-compiler, as well as the utility program to download a hex file into the micro-controller are all available on the Internet, **free of charge**.

2.2 Key features of PIC18F4550

☐ The **key features** of PIC18F4550 include the following:

Parameter Name	**Value**
Program Memory Type	**Flash**
Program Memory (KB)	32
CPU Speed (MIPS)	12
RAM Bytes	2,048
Data EEPROM (bytes)	256
Digital Communication Peripherals	1-A/E/USART, 1-MSSP(SPI/I2C)
Capture/Compare/PWM Peripherals	1 CCP, 1 ECCP
Timers	1 x 8-bit, 3 x 16-bit
ADC	13 ch, 10-bit
Comparators	2
USB (ch, speed, compliance)	1, Full Speed, USB 2.0 (12Mbits/s)
Temperature Range (C)	-40 to 85
Operating Voltage Range (V)	2 to 5.5
Pin Count	40

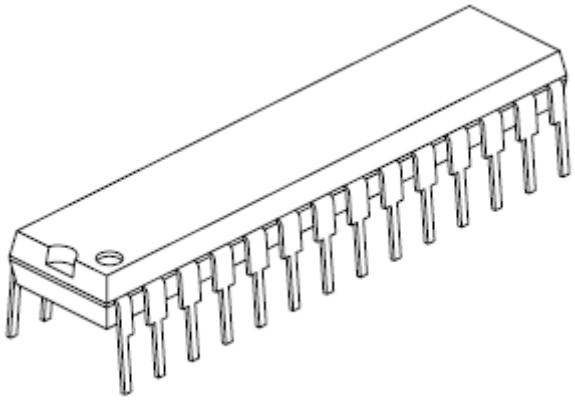
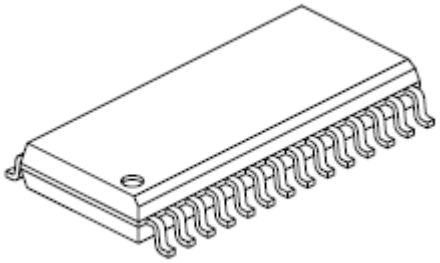
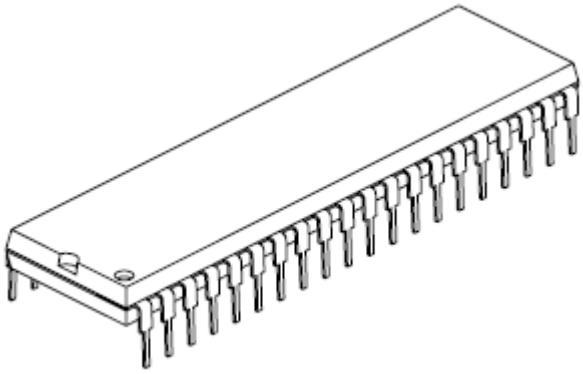
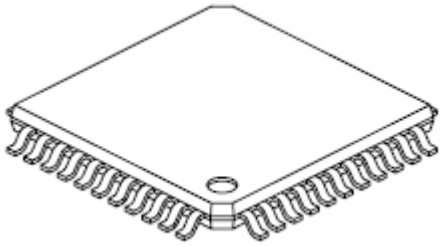
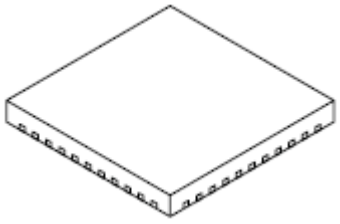
☐ The **USB** connectivity, large amounts of **RAM** memory for buffering and **Flash** program memory make it ideal for embedded control and monitoring applications that require **periodic** connection with a PC via USB for data upload / download and/or firmware **updates**.

☐ Do you know what the feature below means? 

CPU Speed = 12 MIPS means the micro-controller _____

2.3 Variations of PIC18F4550

☐ PIC18F4550 comes in variety. In terms of **packaging**, you can choose from these:

 28-Lead Skinny Plastic Dual In-Line (SP) – 300 mil Body [SPDIP]	 28-Lead Plastic Small Outline (SO) – Wide, 7.50 mm Body [SOIC]
 40-Lead Plastic Dual In-Line (P) – 600 mil Body [PDIP]	 44-Lead Plastic Thin Quad Flatpack (PT) – 10x10x1 mm Body, 2.00 mm Footprint [TQFP]
 44-Lead Plastic Quad Flat, No Lead Package (ML) – 8x8 mm Body [QFN]	

☐ The SPDIP & PDIP packages are more suitable for prototyping using bread-board, strip-board & **thru-hole** PCB. However, SOIC, TQFP & QFN packages will make the final product/gadget **smaller**.

☐ In terms of **features**, you can choose from these:

Features	PIC18F2455	PIC18F2550	PIC18F4455	PIC18F4550
Program Memory (Bytes)	24576	32768	24576	32768
Program Memory (Instructions)	12288	16384	12288	16384
Interrupt Sources	19	19	20	20
I/O Ports	Ports A, B, C, (E)	Ports A, B, C, (E)	Ports A, B, C, D, E	Ports A, B, C, D, E
Capture/Compare/PWM Modules	2	2	1	1
Enhanced Capture/Compare/ PWM Modules	0	0	1	1
Parallel Communications (SPP)	No	No	Yes	Yes
10-Bit Analog-to-Digital Module	10 input channels	10 input channels	13 input channels	13 input channels
Packages	28-pin PDIP 28-pin SOIC	28-pin PDIP 28-pin SOIC	40-pin PDIP 44-pin TQFP 44-pin QFN	40-pin PDIP 44-pin TQFP 44-pin QFN

☐ You will notice that the 2550 / 4550 have more **program memory** compared to the 2455 / 4455. The bigger PIC's (4455 / 4550) of course have **more input/output pins**.

☐ Do you know the difference between **parallel communication** and **serial communication?** 

In parallel communication, k bits of data can be sent simultaneously over k wires.	In serial communication, k bits of data can be sent _____ over ____ wire.

☐ Microchip has hundreds of different PIC's. Check out www.microchip.com. The above 4 are featured in the **same datasheet** and have many things in common.

☐ Choosing a suitable PIC to use for a project / product will only come with experience. The choice often depends on **features** currently required,

future **enhancements** planned for the product/gadget, prototyping / manufacturing **constraints**, **price** and **availability** etc. Frequently, engineers / designers look for something "good enough", not necessarily "optimal".

2.4 Block diagram & on-chip peripherals

- As can be seen from the block diagram, PIC18F4550 has many on-chip peripherals.

- In this course we will study those that are most useful for projects e.g. **I/O, ADC, Timer, Interrupt, USART**.

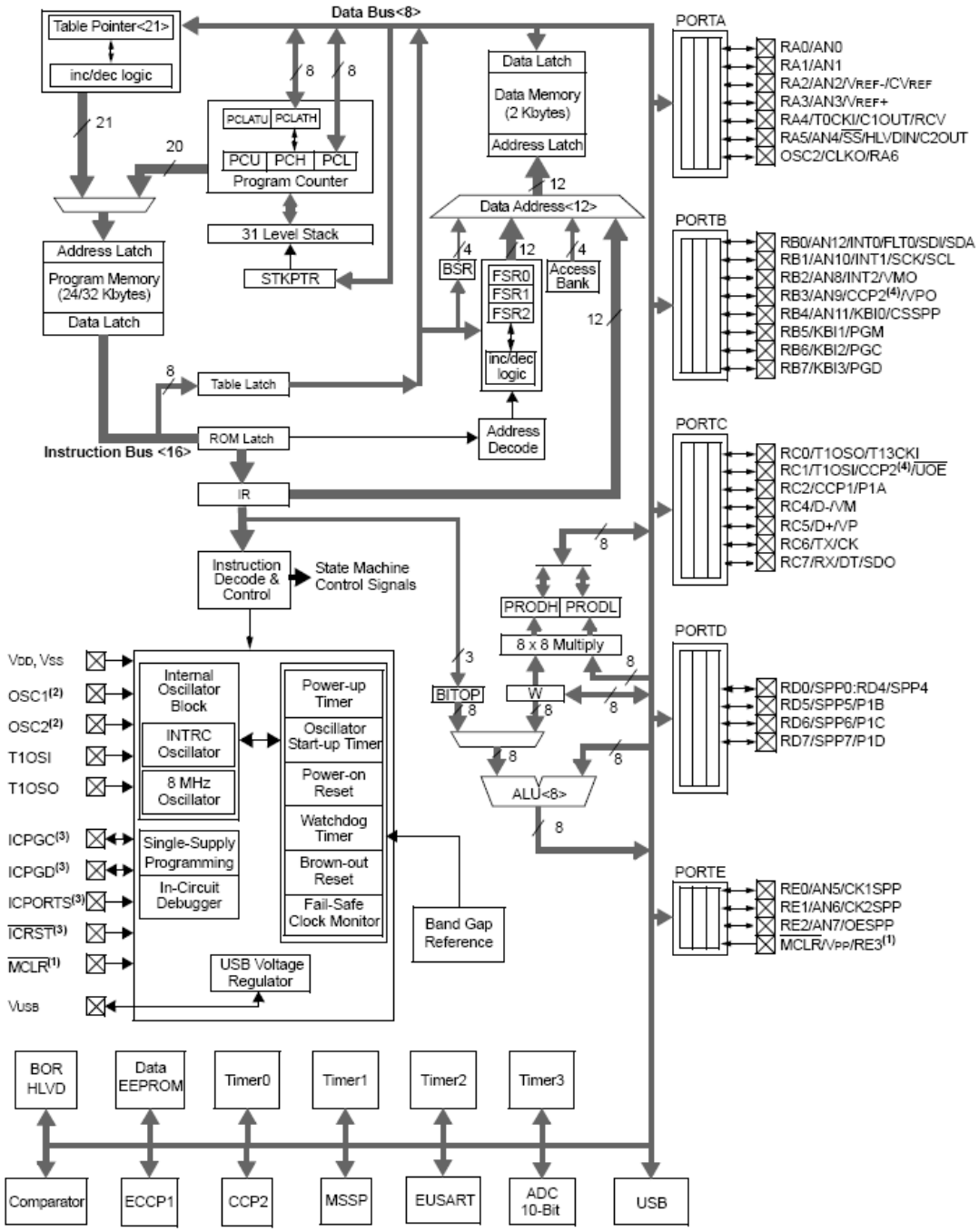


Figure 2.1 – PIC18F4550 block diagram

2.5 Pin diagram

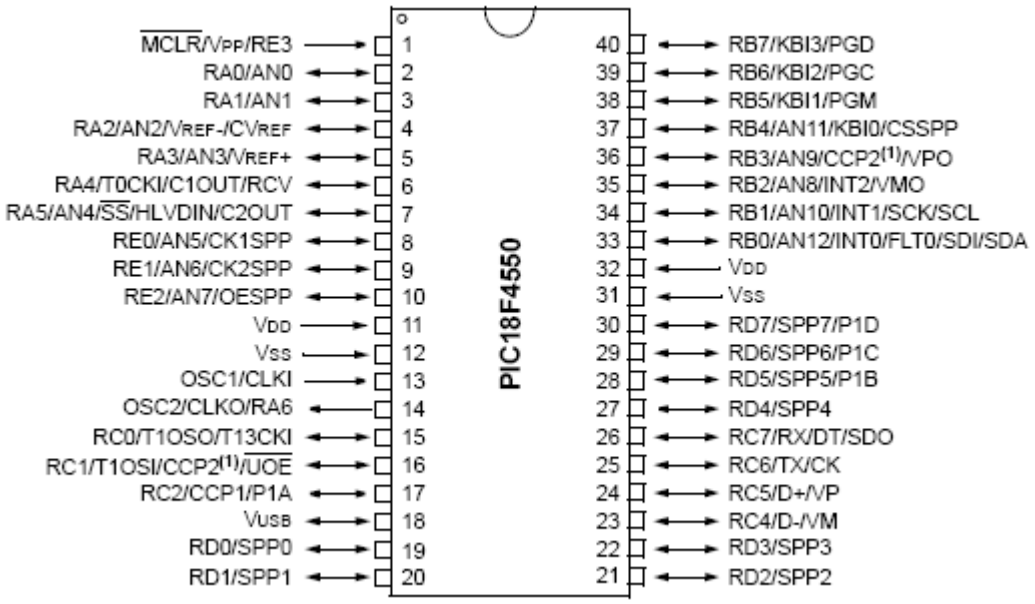


Figure 2.2 – PIC18F4550 pin diagram

☐ You can see that many pins have **multiple functions**. For instance, the pin 33 is RB0 (Port B Pin 0), AN12 (analogue input 12), INT0 (external interrupt 0), FLT0, SDI and SDA, i.e. a total of 6 functions multiplexed on one pin.

☐ Study the pin diagram above and answer the following questions:

At which pins should the power supply (5 volts & ground) be connected?	5 volt @ ____ & ____ ground @ ____ & ____
At which pins should the crystal/oscillator be connected to supply the clock to the PIC?	____ & ____
At which pins should the reset button be connected?	____

2.6 A basic PIC-based circuit

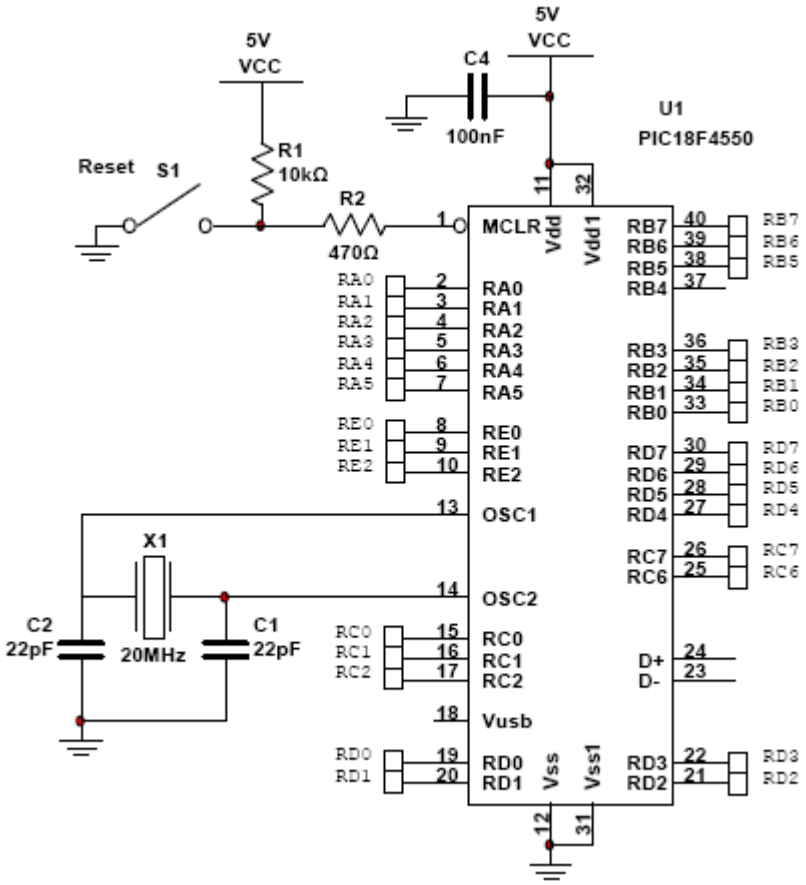


Figure 2.3 – A basic PIC18F4550-based circuit

- As you can see, a **5 volt** supply & **ground** need to be connected to pins 11, 12, 31, 32 as shown. A 100nF **capacitor** is connected across 5V and ground.

- A 20MHz **crystal** together with two 22pF **capacitors** are connected at pins 13 & 14 to provide the clock source. This clock signal will go through a PLL (Phase Locked Loop) to generate a 48MHz clock required for USB operation. For this module, you can take Fosc as 48MHz.

- A simple **reset circuitry** (consisting of a **switch** and two **resistors**: 10kOhm & 470Ohm) at pin 1 completes the picture.

- Of course, more connections are usually needed, depending on the application.

- For the Micro-controller board used during the practical lessons in this module, the pins 18, 23, 24, 37 are also connected so that a hex file can be downloaded to the PIC18F4550 via the **USB** port of a PC.

- Likewise, the I/O pins (RA0-5, RB0-3, 5, RC0-2, 6-7, RD0-7, RE0-2) are connected to interface to other **input / output devices**.

2.7 A simple application – "zebra crossing light control"

- Take a look at the more **intelligent** version of the Zebra Crossing Lights below. If no body presses the buttons, the lights will blink at a certain frequency (e.g. 1 Hz). If a pedestrian presses a button, the lights will blink at a higher frequency and the Buzzer will beep for the next few seconds (e.g. 10 seconds). Of course, the buttons can be replaced by suitable **sensors**.

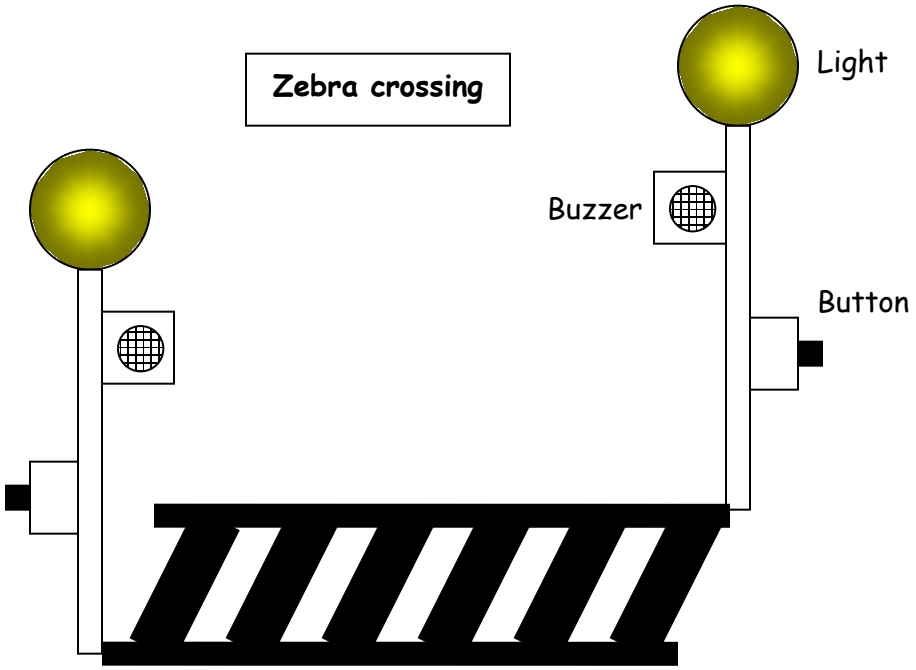


Figure 2.4 – "Intelligent Zebra Crossing Lights"

- The **block diagram** is shown below:

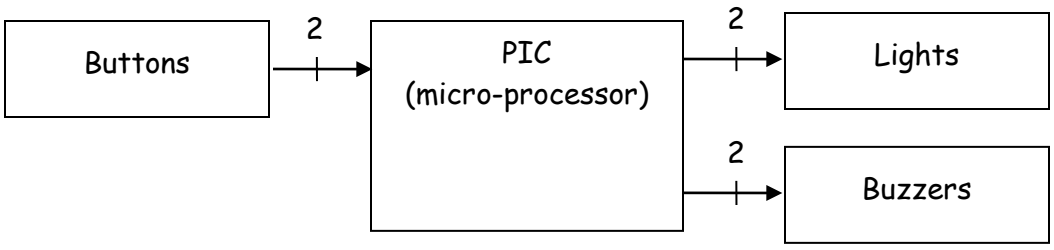


Figure 2.5 – Block diagram of "Intelligent Zebra Crossing Lights"

☐ The **flowchart** follows:

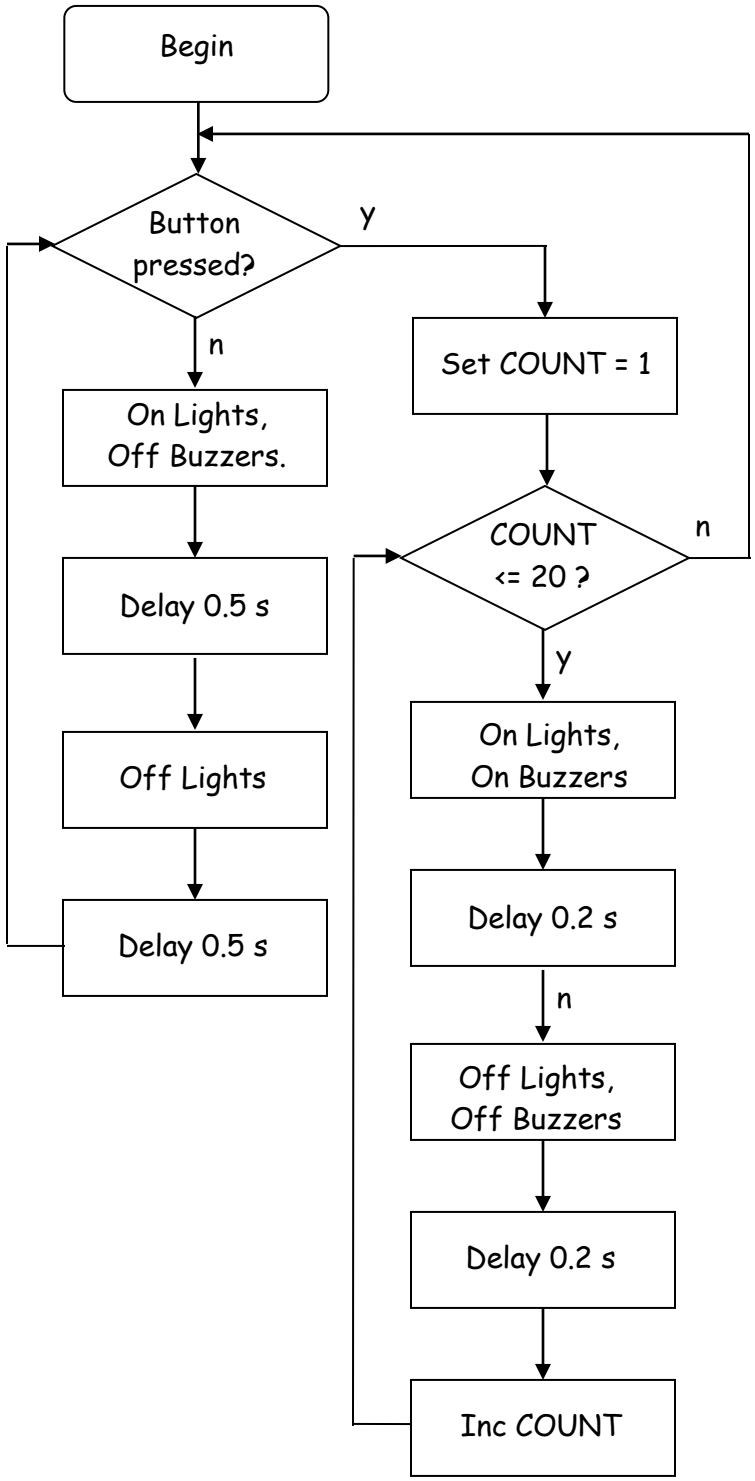


Figure 2.6 – Flowchart of "Intelligent Zebra Crossing Lights"

☐ You have learnt how to draw block diagram and flowchart in Chapter 1. Here, you will learn to draw the **circuit diagrams** for simple **input/output devices** as well, e.g. a push button as input and an LED as output.

☐ The **push button** below gives logic '0' when pressed. Such button is said to be "**active low**".

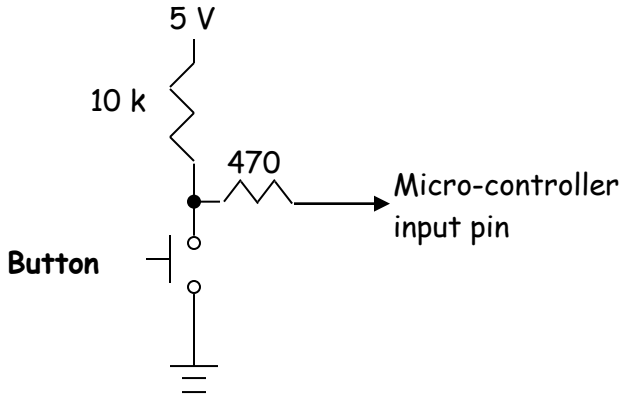


Figure 2.7 – Circuit diagram for push button (active low)

☐ The micro-controller pin connected to the button must be **configured** as a **digital input** pin to read the status of the button i.e. "pressed" or "not pressed". We will learn how to do this in the next chapter.

☐ How is the push button below different?

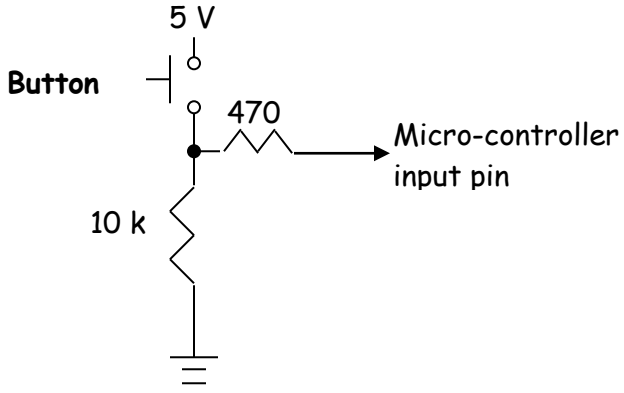


Figure 2.8 – Circuit diagram for push button (**active high**)

Your answer: _____

- In the lab, LED's are sometimes used to "emulate" traffic lights.

- The **LED** below lights up when logic '1' is applied. Such LED is said to be "**active high**".

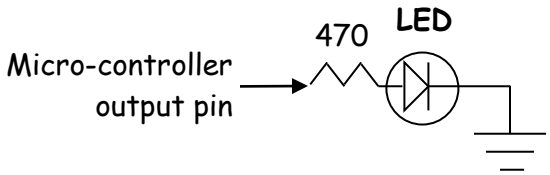


Figure 2.9 – Circuit diagram for LED (active high)

- The micro-controller pin connected to the LED must be **configured** as a **digital output** pin to control the status of the LED i.e. "on" or "off". Again, we will wait until the next chapter.

- How is the LED below different?

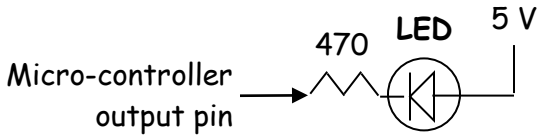


Figure 2.10 – Circuit diagram for LED (**active low**)

Your answer: _____

☐ The **buzzer** below is turned on when logic '1' is applied i.e. **active high**.

☐ The buzzer can also be connected as active low. Again, the micro-controller pin connected to this must be **configured** as a **digital output** pin.

2.8 <u>The steps to create a project, to write & compile a C program and to download a hex file into the PIC18F4550.</u>

☐ Suppose you have **fabricated** on a single PCB:

1. the "basic PIC18F4550-based **circuit**" – Figure 2.3
2. the "push button circuit" x 2 – Figure 2.7/2.8
3. the "LED circuit" x 2 – Figure 2.9/2.10
4. the "buzzer circuit" - Figure 2.11,

☐ How do you go about **programming** the PIC18F4550 to get the project working?

☐ The **key steps** are: 1. create a **project**, 2. write a **C-program**, 3. **compile** the C-program and 4. **download** the hex file into the PIC.

☐ You will go through these steps in the lab **experiments**.

☐ In the lab, a "**boot-loader**" is used in the PIC18F4550. (This program downloads a user program from a PC via the USB port.)

☐ The C program to be written for the "zebra crossing light control" follows:

You may come back to this C-program in the future…

```
#include …
#include …

unsigned char COUNT;

main (void)
{
  TRISB = 0b00000001; // use Bit 0 of Port B for input BUTTON… – active low
  TRISC = 0b00000000; // use Bit 0 of Port C for output Buzzer…
  TRISD = 0b00000000; // use Bit 0 of Port D for output LED…

  while(1) // repeat forever
  {
    if (PORTBbits.RB0 == 0) // checking if active low BUTTON pressed...
    {
      for (COUNT = 1; COUNT <= 20; COUNT++) // pressed => UP freq for next 10 blinks
      {
        PORTDbits.RD0 = 1; // On LED
        PORTCbits.RC0 = 1; // On BUZZER
        Delay… // a short while
        PORTDbits.RD0 = 0; // Off LED
        PORTCbits.RC0 = 0; // Off BUZZER
        Delay… // a short while
      } // for
    } // if
    else
    {
        PORTDbits.RD0 = 1; // On LED
        PORTCbits.RC0 = 0; // Off BUZZER
        Delay… // a short while
        Delay… // a short while
        Delay… // a short while
        PORTDbits.RD0 = 0; // Off LED
        Delay… // a short while
        Delay… // a short while
        Delay… // a short while
    } // else
  } // while
} // main
```

Chapter 3 – PIC18F4550's I/O Ports and Device Interfacing (Part 1)

Chapter Overview

3.1 PIC18F4550 I/O ports

- PIC18F4550 has **5 I/O ports**: Ports A to E.

- Why are I/O ports required? The micro-controller is an **intelligent device**. For it to be really useful, it must be able to monitor the **status** of some **input** devices e.g. sensors and to **control** some **output** devices e.g. actuators.

- In the figure below, the micro-controller monitors the temperature of a laptop. If the temperature exceeds a preset threshold e.g. 30 deg Celsius, the fan will be switched on.

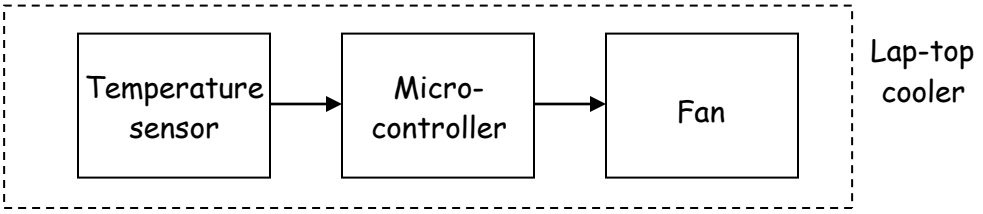


Figure 3.1 – Micro-controller + input + output

☐ Examine the PIC18F4550 **pin diagram** below. Can you see where the Port A pins are? How many are there? How are they numbered?

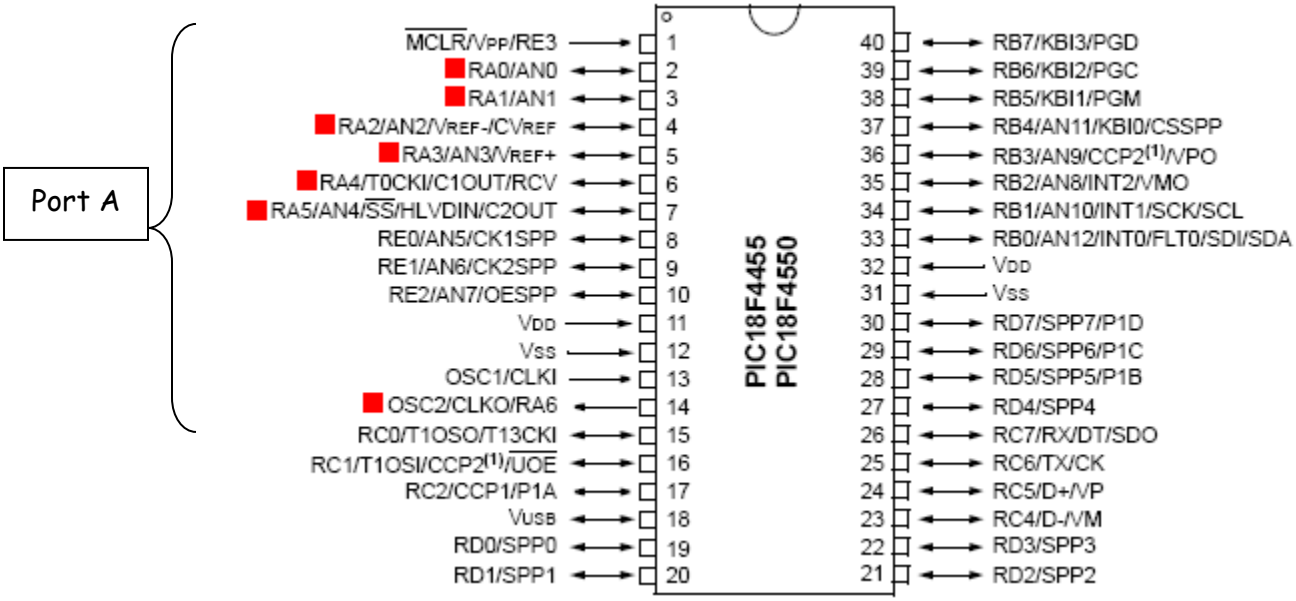


Figure 3.2 – PIC18F4550 I/O ports

☐ List the I/O pins available for each port.

Port A	Port B	Port C	Port D	Port E
RA6 - 0				
total 7 pins				

☐ Many pins are multiplexed with **alternate functions** from the peripheral features, e.g. RA0 is also AN0 i.e. an analogue input. Likewise, RA1, RA2, RA3 & RA5 can be AN1, AN2, AN3 & AN4 respectively.

☐ In general, when a **peripheral** is enabled, that pin may not be used as a **general purpose I/O** pin.

3.2 Port A

☐ On the **micro-controller board** you are using in the lab, the available Port A pins are RA5-0. Pin 14 (RA6) is connected to the oscillator, so RA6 is not available.

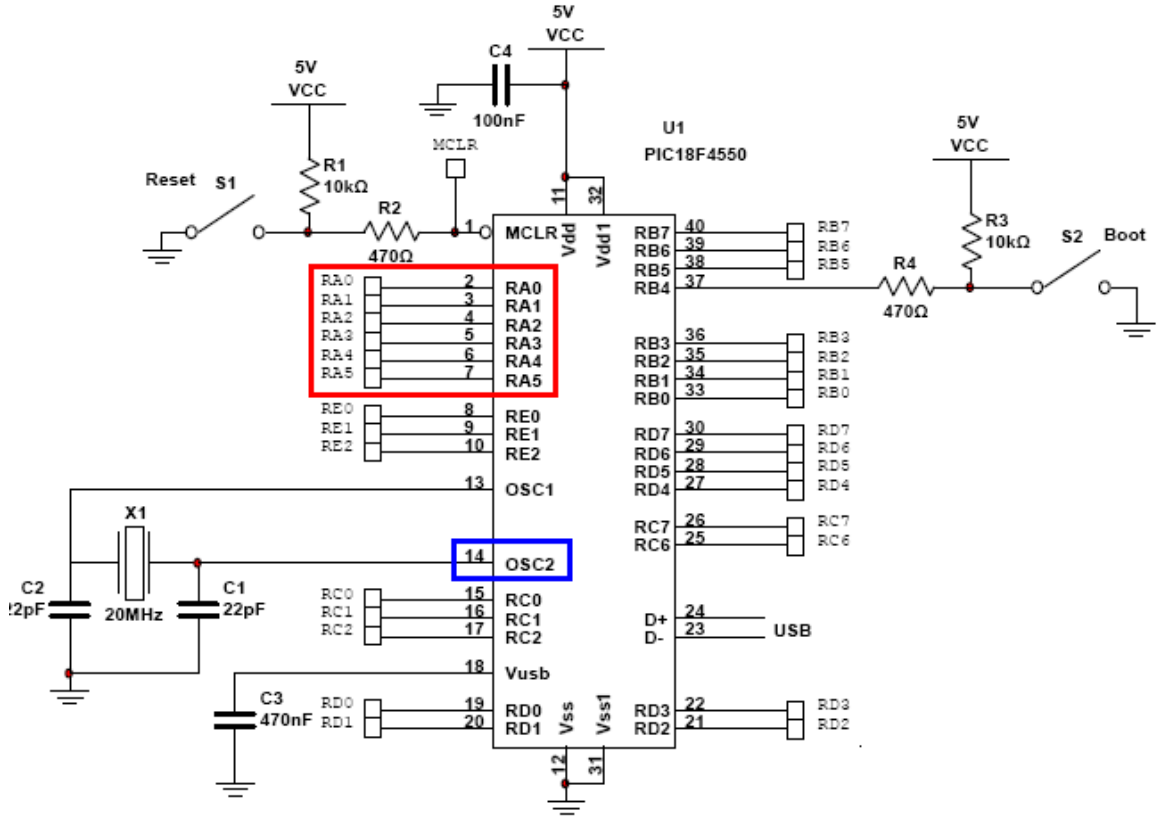


Figure 3.3 – General purpose I/O pins available on the micro-controller board

☐ Upon power on **reset**, RA5 and RA3-0 are configured as analogue inputs and RA4 is configured as a digital input. To turn Port A into a **digital I/O** port, use the C-language command below. How this works is explained on the next page (no need to know the explanation in details).

ADCON1 = 0x0F; or **ADCON1 = 0b00001111;**

Optional

REGISTER 21-2: ADCON1: A/D CONTROL REGISTER 1

U-0	U-0	R/W-0	R/W-0	R/W-0(1)	R/W(1)	R/W(1)	R/W(1)
—	—	VCFG0	VCFG0	PCFG3	PCFG2	PCFG1	PCFG0
bit 7							bit 0

PCFG3:PCFG0: A/D Port Configuration Control bits:

PCFG3: PCFG0	AN12	AN11	AN10	AN9	AN8	AN7(2)	AN6(2)	AN5(2)	AN4	AN3	AN2	AN1	AN0
0000(1)	A	A	A	A	A	A	A	A	A	A	A	A	A
0001	A	A	A	A	A	A	A	A	A	A	A	A	A
0010	A	A	A	A	A	A	A	A	A	A	A	A	A
0011	D	A	A	A	A	A	A	A	A	A	A	A	A
0100	D	D	A	A	A	A	A	A	A	A	A	A	A
0101	D	D	D	A	A	A	A	A	A	A	A	A	A
1100	D	D	D	D	D	D	D	D	D	D	A	A	A
1101	D	D	D	D	D	D	D	D	D	D	D	A	A
1110	D	D	D	D	D	D	D	D	D	D	D	D	A
1111	D	D	D	D	D	D	D	D	D	D	D	D	D

A = Analog input D = Digital I/O

Figure 3.4 – Making Port A a digital I/O port by ADCON1 = 0x0F

☐ We will discuss **ADC** (analogue to digital conversion) in details in a subsequent chapter.

3.3 Configuring a pin to be an input or output pin

☐ Port A is a **7-bit** wide **bidirectional** port.

☐ The **registers** associated with Port A are TRISA and PORTA. **TRISA** is the "**data direction register**" for Port A.

☐ **Setting** a TRISA bit (= 1) will make the corresponding PORTA pin an **input**. **Clearing** a TRISA bit (= 0) will make the corresponding PORTA pin an **output**. For instance:

	7	6	5	4	3	2	1	0
TRISA		0	0	0	1	1	1	1

		RA6	RA5	RA4	RA3	RA2	RA1	RA0
PORTA								
		Outputs			Inputs			

Figure 3.5 – Configuring Port A pins as inputs / outputs using TRISA

☐ In the figure above, the C-language command TRISA = 0x0F or TRISA = 0b00001111 is used to make RA6-4 output pins and RA3-0 input pins.

[TRISA bit 7 is unimplemented, it doesn't matter what is written to that bit.]

☐ An easy way to remember this is '**0**' (zero) looks like '*O*' (in ***O*utput**) while '**1**' (one) looks like 'I' (in ***I*nput**).

☐ What is the command to achieve this?

		RA6	RA5	RA4	RA3	RA2	RA1	RA0
PORTA								
		Input	Input	Output	Output	Input	Input	Input

	7	6	5	4	3	2	1	0
TRISA								

C-language command : TRISA = _____

☐ Upon **reset**, TRISA = 0b - 1 1 1 1 1 1 1, i.e. PORTA is an **input** port by default.

3.4 Connecting an LED as output and a switch as input

☐ The circuit diagram below shows how an **LED** can be connected as an **output** and a **switch** as an **input**. Find out why the two 470-ohm resistors are required.

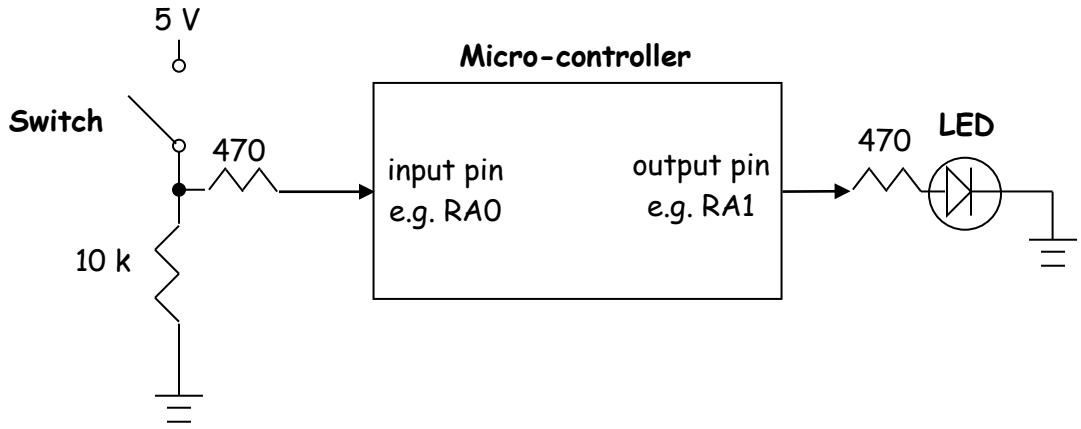


Figure 3.6 – LED as output and switch as input

- If you have connected RA1 to the LED circuit, RA1 must be configured as an output pin. Likewise, if you have connected RA0 to the switch circuit, RA0 must be configured as an input pin.

- The correct configuration command is TRISA = 0b00000001 i.e. the last 2 bits must be 0 and 1.

3.5 Reading an input pin

- How can the micro-controller know if the switch is closed or open?

- It can be seen from the diagram that the switch is "**active high**" i.e. closed switch = logic '1', while opened switch = logic '0'.

- The C-language command to read the switch status is

```
if ( PORTAbits.RA0 == 1 )  // if switch closed
        …. // do something
```

- The status of the I/O pins can thus be checked **bit by bit,** as shown above.

- An alternative is shown below.

Optional

We only want to know RA0, so **mask off** the other bits

```
if ( ( PORTA & 0b00000001 ) == ( 0b00000001 ) )   // if switch closed
        …. // do something
```

If, after masking, there is a '1' at the LSBit i.e. RA0, then do something

	7	6	5	4	3	2	1	0
PORTA	-	?	?	?	?	?	?	???
MASK	0	0	0	0	0	0	0	1
PORTA & 0b00000001	0	0	0	0	0	0	0	???
Is the result this?	0	0	0	0	0	0	0	1
If so, do something.								

3.6 Writing a '1' or '0' to an output pin

☐ How can the micro-controller turn the LED on or off?

☐ It can be seen from the diagram that the LED is "**active high**" i.e. logic '1' turns it on.

☐ The C-language command to turn the LED on is

```
PORTAbits.RA1 = 1;
```

☐ Give the two alternative C-language commands to turn the LED off. 

PORTA_____;

PORTA_____;

3.7 Other I/O ports B to E

☐ As Port A has already been discussed in details, the description below for the other ports will be brief.

Port B

☐ Port B is a **8-bit** wide **bidirectional** port. TRISB is the "data directional register" for PORTB. Remember '1' = **I**nput, '0' = **O**utput.

☐ Examine the PIC18F4550 **pin diagram** (in Figure 3.2) again. Where are the 8 Port B pins RB7-0? What **other peripherals** use these pins?

☐ Upon power on **reset**, RB7-0 are configured as digital inputs.

☐ On the **micro-controller board** (see Figure 3.3), the available Port B pins are RB7-5 and RB3-0. Pin 37 (RB4) is connected to an active low switch, so RB4 is not available.

Port C

- Port C is a **7-bit** wide **bidirectional** port. RC3 pin is not implemented. TRISC is the "data directional register" for Port C. However, RC4 and RC5 do not have TRISC bits associated with them – they can only function as digital inputs.

- Port C is primarily multiplexed with the **serial communication modules** – e.g. the USB module.

- Upon power on **reset**, RC7-4, RC2-0 are configured as digital inputs.

- On the **micro-controller board** (see Figure 3.3), the available Port C pins are RC7-6 and RC2-0. Pin 24 (RC5) and Pin 23 (RC4) are used for USB downloading of hex file into the micro-controller, so RC5-4 are not available.

Port D

- Port D is a **8-bit** wide **bidirectional** port. TRISD is the "data directional register" for Port D.

- Port D is multiplexed with the **Enhanced CCP module** and the **Streaming Parallel Port** (SPP).

- Upon power on **reset**, RD7-0 are configured as digital inputs.

- On the **micro-controller board** (see Figure 3.3), the Port D pins RD7-0 are all available.

Port E

- Port E is a **4-bit bidirectional** wide port. TRISE is the "data directional register" for Port E. However, RE3 does not have TRISE bit associated with it – it can only function as input.

- Similar to other ports, Port E is multiplexed with other functions.

- Upon power on **reset**, RE2-0 are configured as analogue inputs, while RE3 is configured as a digital input (RE3 is only available if Master Clear (/MCLR) functionality is disabled).

☐ To turn Port E into a **digital I/O** port, use the C-language command below.

ADCON1 = 0x0F; // there are other possibilities

☐ On the **micro-controller board** (see Figure 3.3), the available Port E pins are RE2-0. Pin 1 (RE3) is used as the active low Master Clear (/MCLR) input, so RE3 is not available.

3.8 An exercise on I/O port

☐ Study the figure below and fill in the blanks to describe how the I/O ports are used as digital inputs / outputs.

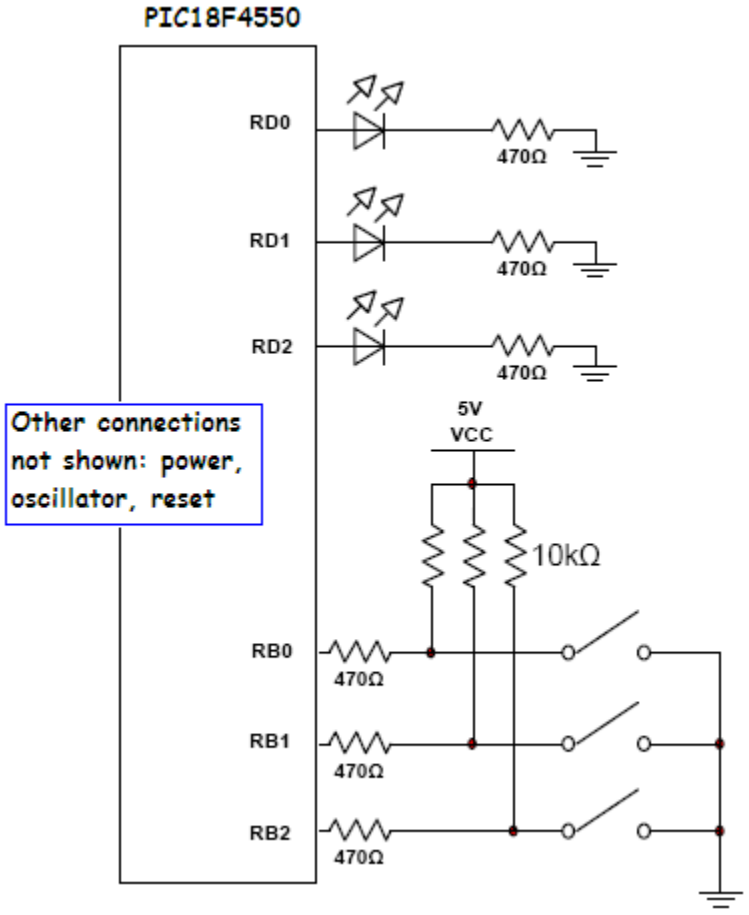


Figure 3.7 – Monitoring status of switches, using LEDs

- ☐ The Port B pins _____ are used as digital _____. They are connected to the _____, _____ and _____.

- ☐ To configure the pins as digital inputs, the commands to be used are

  _____

  _____

- ☐ If a switch is left open, the corresponding pin will read a logic _____. If the switch is closed, the pin will read a logic _____.

- ☐ The Port D pins _____ are used as digital _____. They are connected to the _____ and _____.

- ☐ To configure the pins as digital outputs, the command to be used is

  _____

- ☐ If logic '1' is written to a pin, the LED will _____. If logic '0' is written to a pin, the LED will _____.

- ☐ The short program below shows how the LED's can be used to indicate the status of the switches, i.e. a closed switch lights up the corresponding LED.

  Fill in the blanks and describes what the lines marked X do.

```
#include …
// other lines of code before main……..

main()
{
// configure Port B as digital input and Port D as digital output
// TRISB = 0xFF; -- is not necessary as, upon reset, Port B is input
TRISD = _______; // upon reset RD7-0 are digital inputs – so make them outputs

while (1) // continuously monitor
  {
  // alternative #1:

   if (PORTBbits.RB0 == 0)  // _____________________________________
     PORTDbits.RD0 = 1;     // _____________________________________
   else                     // _____________________________________
     PORTDbits.RD0 = 0;     // _____________________________________

  if (PORTBbits.RB1 == 0)
    PORTDbits.RD1 = 1;
  else
    PORTDbits.RD1 = 0;

  if (PORTBbits.RB2 == 0)
    PORTDbits.RD2 = 1;
  else
    PORTDbits.RD2 = 0;

  // alternative #2:
  // PORTD = ~PORTB; --- copy "complemented" PORTB to PORTD

  Delay… // introduce some delay before checking switch status again
  } // while
} // main
```

X

Chapter 3 – PIC18F4550's I/O Ports and Device Interfacing (Part 2)

Chapter Overview

3.9 Why interfacing?

☐ Many I/O (input/output) devices, beside LED's and switches, can be connected to a micro-controller.

☐ Various kinds of **sensors** can be connected to sense the **environment**. State the purpose of the following sensors:

Sensor	Purpose
LDR	
PIR	
Tilt switch	
Ultrasonic ranger	

☐ Likewise, various kinds of "**actuators**" can be connected to bring about some **response** to changes in the environment. For instance, it is hot, so switch on the fan.

- So, a micro-controller is an intelligent, programmable device. But it requires other electronic components to be useful.

- The PIC18F4550 micro-controller is a **low-power** device. It can be interfaced (or connected) directly to **TTL** and **CMOS** digital devices. Most other devices cannot be connected directly to its I/O pins due to the **mismatch of some electrical properties**: (Reason #1 for "interfacing")

    - A 15V motor cannot be driven directly by the PIC 's 5V output.
    - An input device producing 12V can damage the PIC 's input pin, if connected directly.

- Suitable components/circuits must be placed between the micro-controller pin and the high power devices.

- For PIC18F4550, the maximum **current** that can be **sunk** by an I/O pin is 25mA and the maximum current sunk by all ports is 200mA.

- The maximum **current** that can be **sourced** by an I/O pin is also 25mA and the maximum current sourced by all ports is also 200mA.

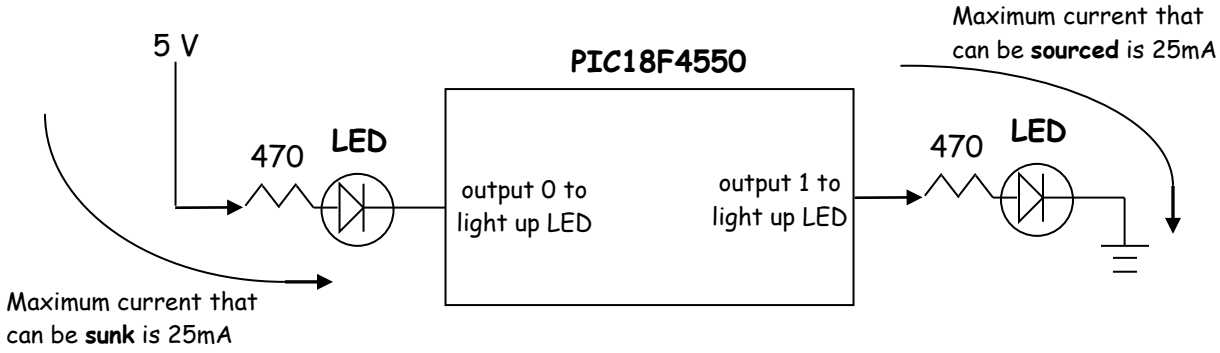


Figure 3.7 – Current sunk vs. current sourced

- The micro-controller also has a **limited number of I/O pins**. So sometimes, **encoder / decoder** are used, to reduce the number of pins required to interface to a device. (Reason #2 for "interfacing")

- The sections below show how common I/O devices can be interfaced to PIC18F4550.

3.10 Interfacing to LED bar

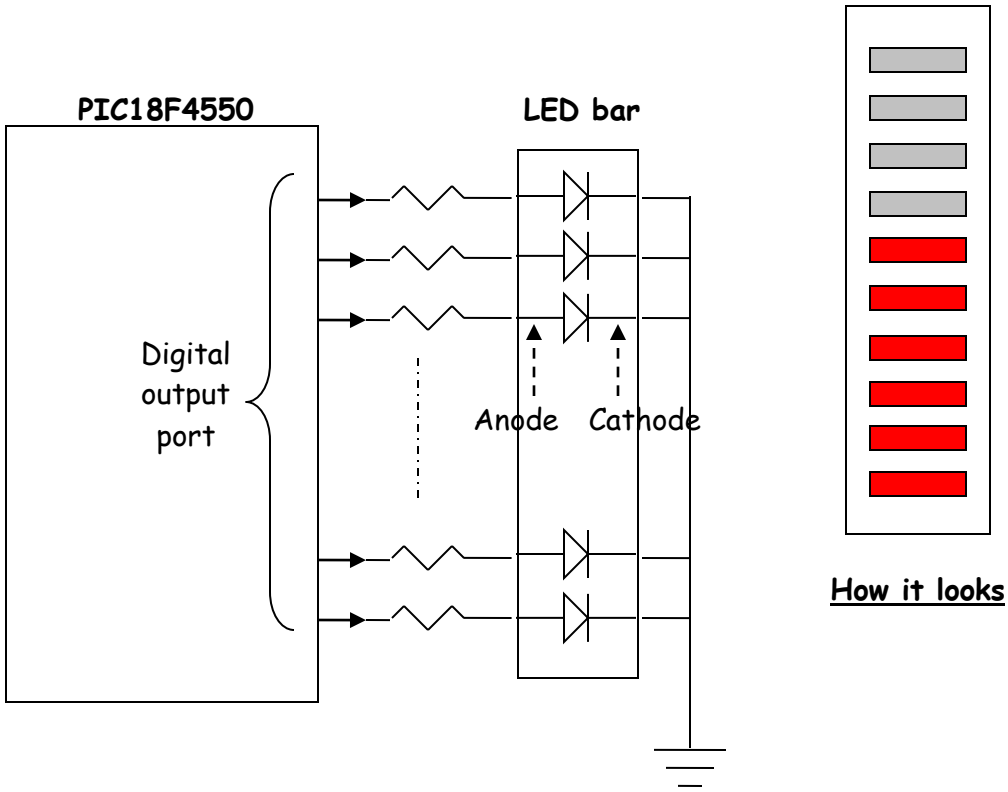


Figure 3.8 – Bar LED

- An LED bar consists of **several** (e.g.10) **LED's** in a single package.
- It is useful as a "**level indicator**".
- Above, the **anodes** are driven (via **current limiting resistors**) by the PIC **output pins** while the **cathodes** are **grounded**.
- The PIC outputs a **logic '1'** to light up an LED.
- **Alternative** connection: the cathodes are driven by the PIC output pins while the anodes are connected to **Vcc**.
- The resistors can be individual resistors or an SIL (single in line) package.
- If the micro-controller outputs 5V at a pin and the LED drops 1.8V and the current-limiting resistor used is 470 ohms, calculate the current in the LED.

Your answer: _____

3.11 Interfacing to 7-segment display (using a 7-segment decoder)

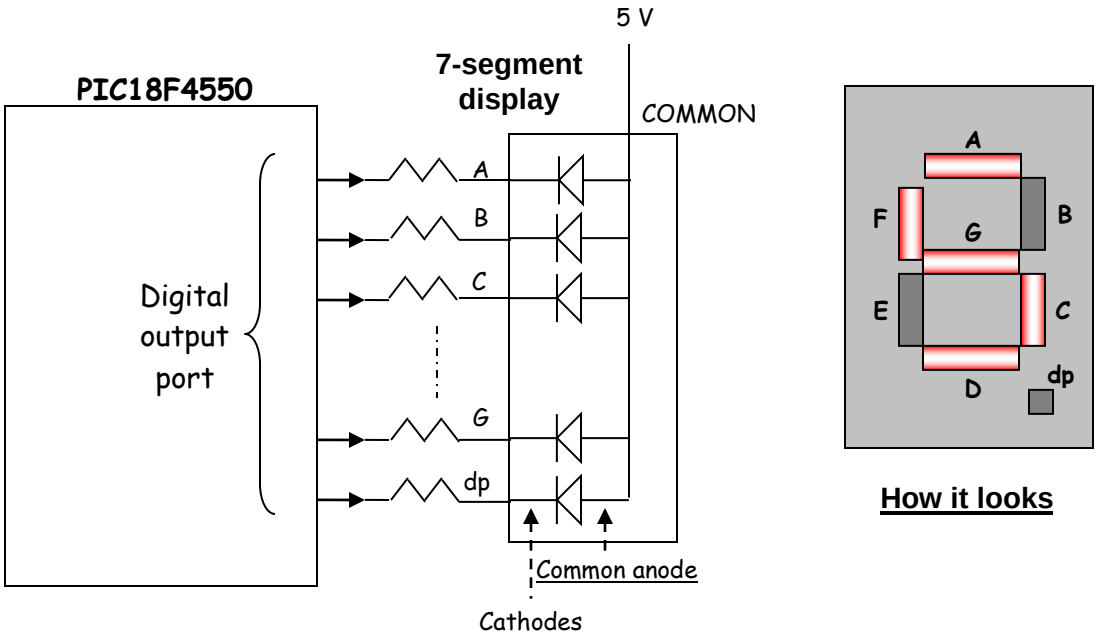


Figure 3.9 – 7 segment display

- A 7-segment display also consists of **several** (usually 8) **LED's** in a single package.
- The segments are arranged into a **figure of 8**, and can display **one decimal/hex digit**. (See "multi-digit display" regarding the decimal point.)
- Above, the **cathodes** are driven (via **resistors**) by the PIC **output pins** while the **anodes** are tied together internally (- a **common anode** device) and connected to **Vcc** (externally).
- The PIC outputs a **logic '0'** to light up a segment.
- If A-G are connected to RD0-6 while dp is connected to RD7, fill in the table below to show how "5" can be displayed, with the dp off.

Segment	dp	G	F	E	D	C	B	A
On / Off	Off							
PIC output pin	RD7	RD6	RD5	RD4	RD3	RD2	RD1	RD0
'0' / '1'	'1'							

☐ The C-code is

```
TRISD = 0b 0 0 0 0 0 0 0 0 ;      // configure port D as output
PORTD = 0b 1 0 0 1 0 0 1 0 ;      // display "5" on 7-segment
```

☐ Below, a **binary to 7-segment decoder** is used to reduce the number of PIC pins required.

☐ The PIC only needs to output 0 1 0 1 (binary for 5) at the decoder inputs A3-A0. The decoder will produce the correct 7-segment pattern (0 0 1 0 0 1 0) to display 5.

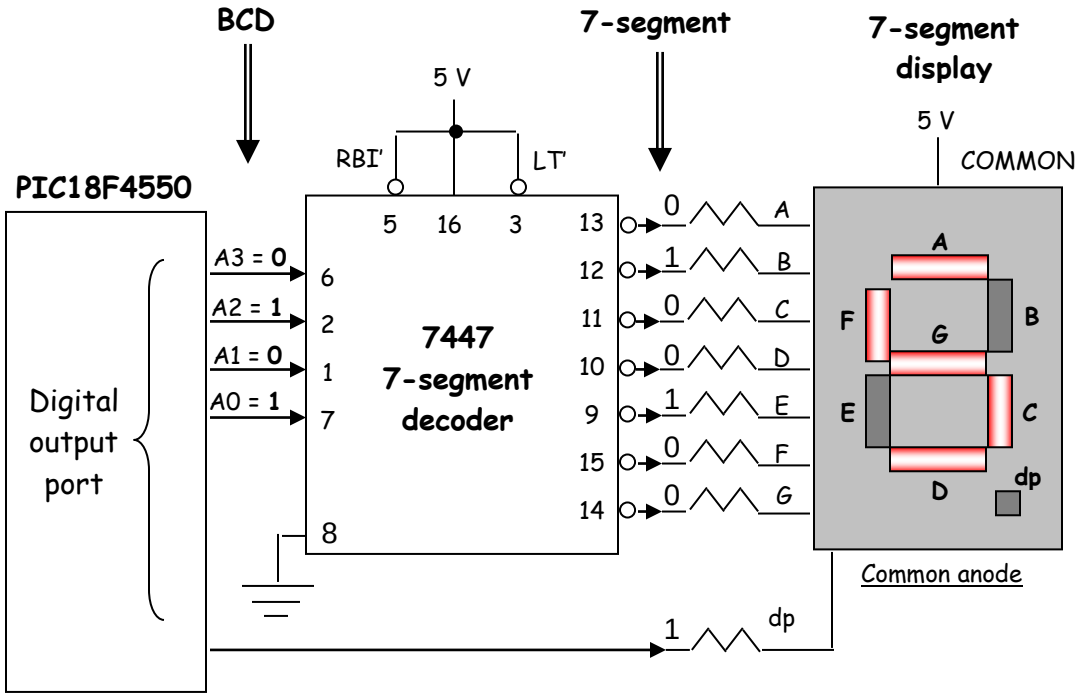


Figure 3.10 – 7 segment display with decoder

3.12 Interfacing to multi-digit 7-segment display

☐ To display multiple digits e.g. 1.23, we need several 7-segment displays.

☐ To **reduce** the number of I/O pins needed, use the following method, based on "**persistence of vision**", "multiplexing" & "transistors as switches".

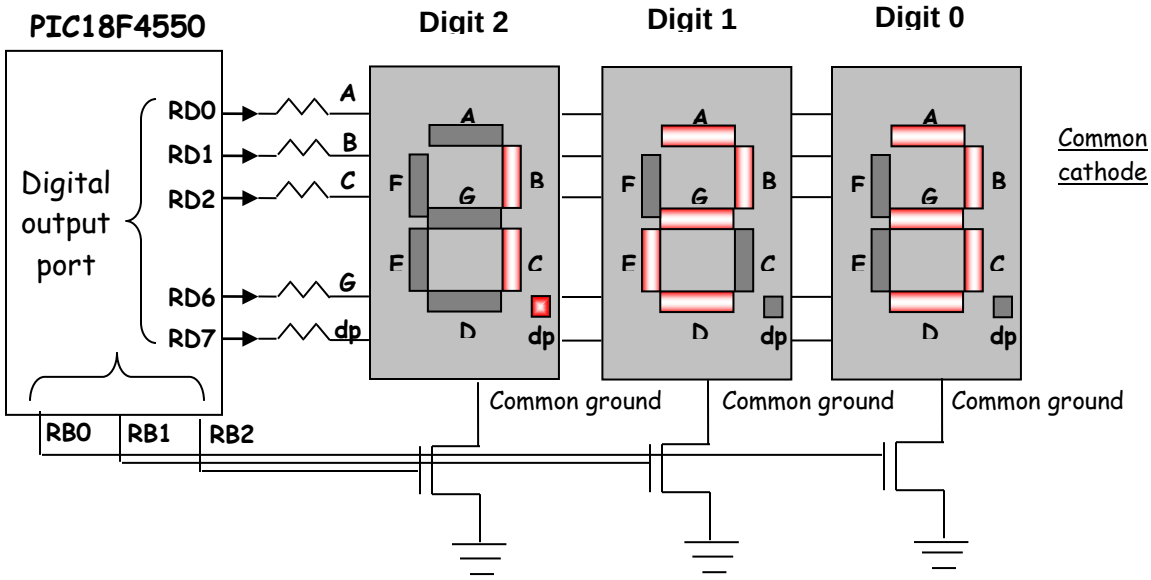


Figure 3.11 – multi-digit 7 segment display

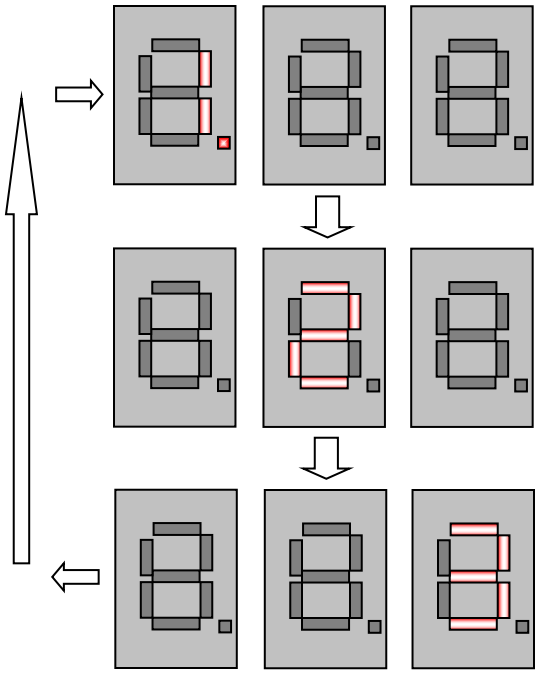
☐ Above, "**common cathode**" displays are used (i.e. a **logic '1'** to light up a segment). RB2=1 enables Digit2 (*), RB1=1 enables Digit1, RB0=1 enables Digit0.

☐ **(*) Transistors as switches** – if a logic '1' written to the control input of a transistor, that transistor will be "on" and the common cathode of the digit will be connected to ground, i.e. enabled.

☐ To show 1.23, the code below can be used. Fill in the blanks for the Digit 1.

```
while (1) // a "forever loop" to show each digit in turn, repeatedly
  {
        // enable Digit 2 and display "1 with decimal point on" on it:
        PORTB = 0 b 0 0 0 0 0 1 0 0; // enable Digit 2
        PORTD = 0 b 1 0 0 0 0 1 1 0; // show 1. on it
        Delay… // introduce a brief delay

        // enable Digit 1 and display "2" on it
        PORTB = 0 b _____________ ; // enable Digit 1
        PORTD = 0 b _____________; // show 2 on it
        Delay… // introduce a brief delay

        // enable Digit 0 and display "3" on it
        PORTB = 0 b 0 0 0 0 0 0 0 1; // enable Digit 0
        PORTD = 0 b 0 1 0 0 1 1 1 1; // show 3 on it
        Delay… // introduce a brief delay

        // RD7-0 are multiplexed to output the digits
        // one after another.
  }
```

☐ What actually happens: Draw what you see here



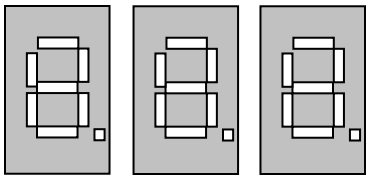


Figure 3.12 – Persistence of vision

D. I. Y.

☐ The "7-segment / switch board" used in the **lab** is shown below.

☐ **Common anode** 7-segments are used together with **inverting drivers**. The end result is that a logic '1' lights up a corresponding segment. RD7-0 multiplexes out the data to be displayed while RB3-0 enables one digit at a time.

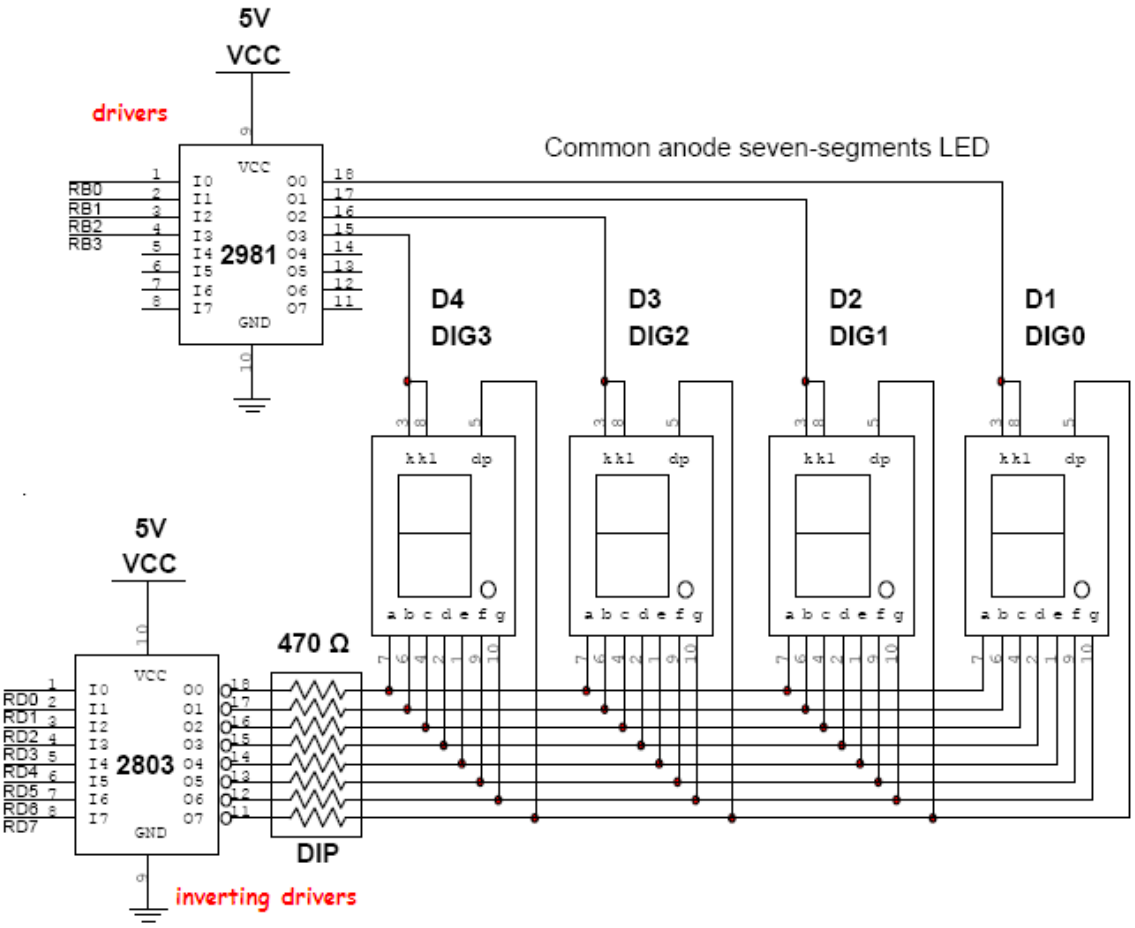


Figure 3.13 – Lab's multi-digit 7 segment display

3.13 Interfacing to matrix keypad (using a keypad encoder)

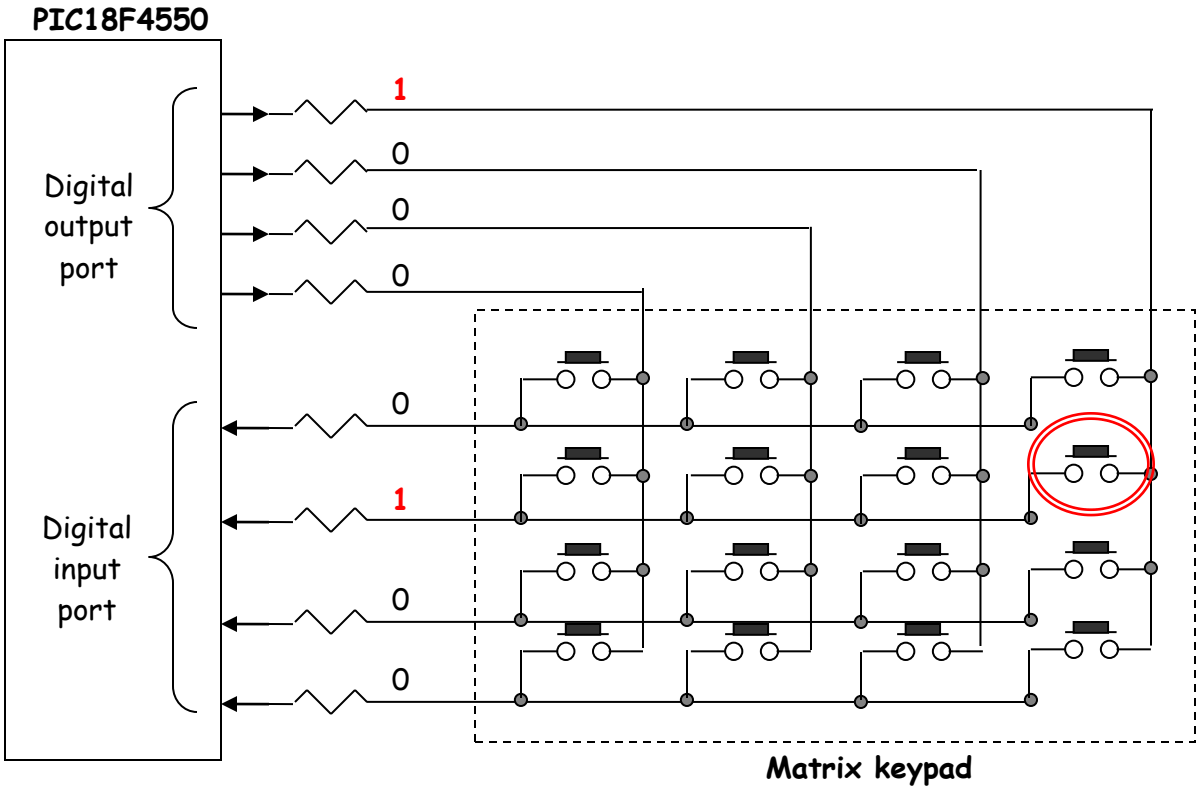


Figure 3.14 – Matrix keypad

- A 4x4 matrix keypad consists of 16 (push button) **switches** arranged in a matrix.
- Only 8 PIC pins (4 out, 4 in) need to be connected, to know which of the 16 keys has been pressed.
- The **scanning** mechanism goes like this, a logic '1' is written to only one column (e.g. right most or 4th column) and a logic '0' to the rest.
- The rows are then read. The row that reads logic '1' uniquely identifies the closed switch.
- If no '1' is read, try the next column e.g. 3rd column, etc.
- We assume that opened switches read '0' (– this is true if there is a weak **pull-down** inside the PIC).

☐ In the lab, the **keypad encoder 74C922** is used:

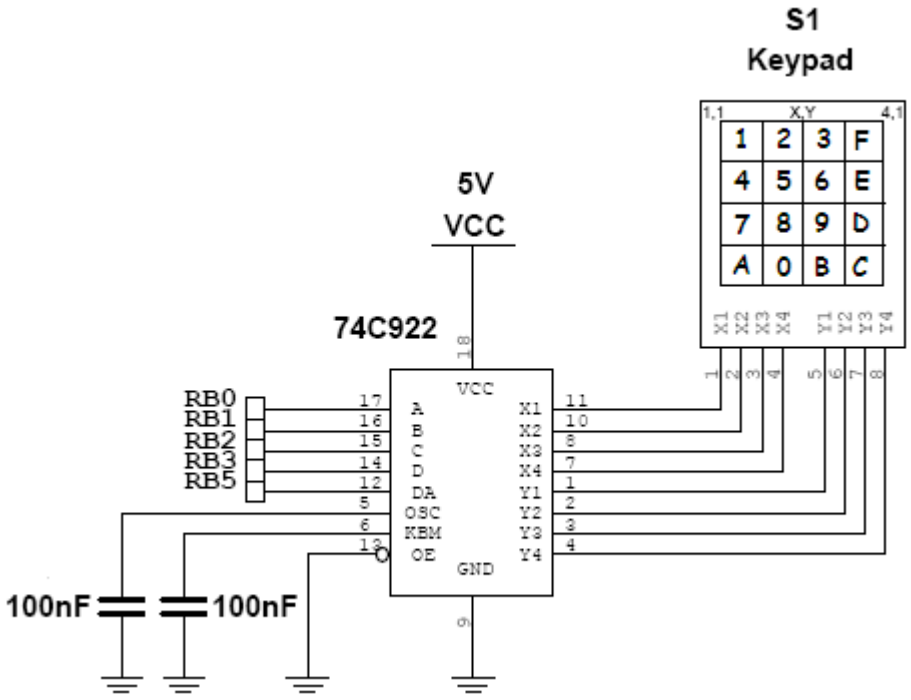


Figure 3.15 – Lab's matrix keypad with encoder

☐ It has the following **truth table**.

	Keys															
	X1 Y1	**X2 Y1**	**X3 Y1**	**X4 Y1**	**X1 Y2**	**X2 Y2**	**X3 Y2**	**X4 Y2**	**X1 Y3**	**X2 Y3**	**X3 Y3**	**X4 Y3**	**X1 Y4**	**X2 Y4**	**X3 Y4**	**X4 Y4**
D	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
C	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
B	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
A	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
DA	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

☐ D (msb), C, B, A identify the key pressed. For instance, if key '2' is pressed, X2, Y1 cause DCBA = 0001. The DA (**data available**) signal will be set to logic '1', whenever a key is pressed.

☐ The code to read the key is as follows

```
const unsigned char lookup[] = "123F456E789DA0BC ";
while (PORTBbits.RB5 == 0);  //wait for a key to be pressed: DA==1
temp = PORTB & 0x0F;  //read from encoder, mask off top 4 bits
KEY = lookup [temp];  // look up table to find key pressed
```

3.14 Interfacing to LCD

- An **alphanumeric** LCD is commonly used with a micro-controller to prompt the user for inputs e.g. password or preset temperature.

- The numbers and alphabets can be displayed in 2 rows of say, 16 characters.

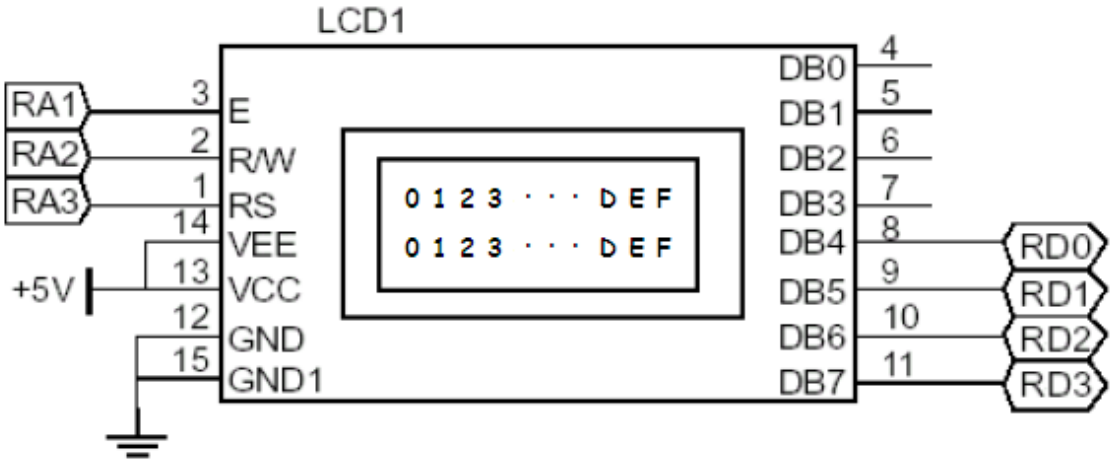


Figure 3.16 – Alphanumeric LCD display – 4-bit mode

- In the "**4-bit mode**" above, a byte (8 bits) can be written as **two nibbles** (one nibble = 4 bits), one after another, via RD3-0.

- The "**8-bit mode**" connection is shown below. Name one advantage and one disadvantage of the 8-bit mode": 

  Advantage: _____ Disadvantage: _____

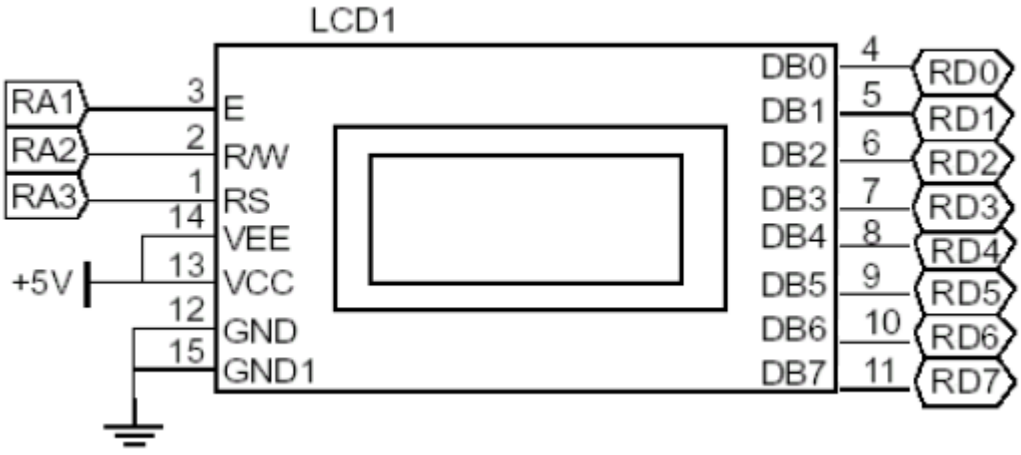


Figure 3.17 – Alphanumeric LCD display – 8-bit mode

3.15 Using a transistor as a switch

- For this module, the details of how transistors work is not important, as we will be using transistors as switches only.
- Transistors have been used as switches in the multi-digit 7-segment display earlier and in the buzzer, motor, solenoid to be covered next.
- Suitable transistors include **2N2222**.

3.16 Interfacing to buzzer

- When the PIC outputs a logic '0', the (BJT) transistor is OFF. This turns the buzzer OFF (as it is "**reverse-biased**": '0' -> '+', Vcc -> '-').
- When the PIC outputs a logic '1', the transistor is ON (Vce[sat] ~0.2 volt). This turns the buzzer ON.

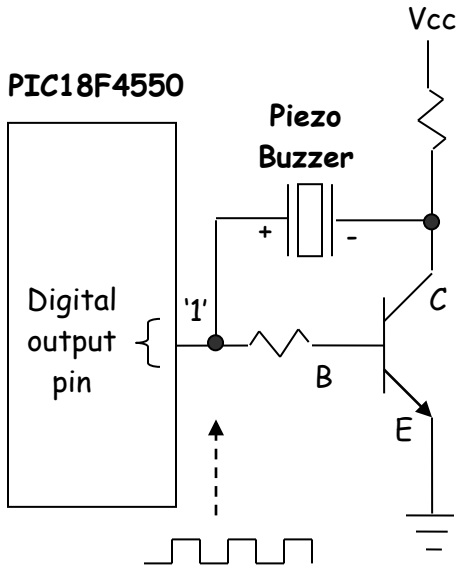


Figure 3.18 – Buzzer

- A **rectangular pulse train** / wave can be produced at the PIC output pin to produce a "**tone**". By varying the **frequency** (& duty cycle) of the wave, the **pitch** (& loudness) of the tone can be controlled. – We will do this in the **Timer** topic.

3.17 Interfacing to DC motor / solenoid

☐ When the PIC outputs a logic '0', the motor is OFF.

☐ When the PIC outputs a logic '1', the transistor (usually a "Darlington" – 2 transistors in cascade, for increased current) is ON. This turns the motor ON, as there is now a **path** for the current to flow from Vcc through the motor and the transistor, to the ground.

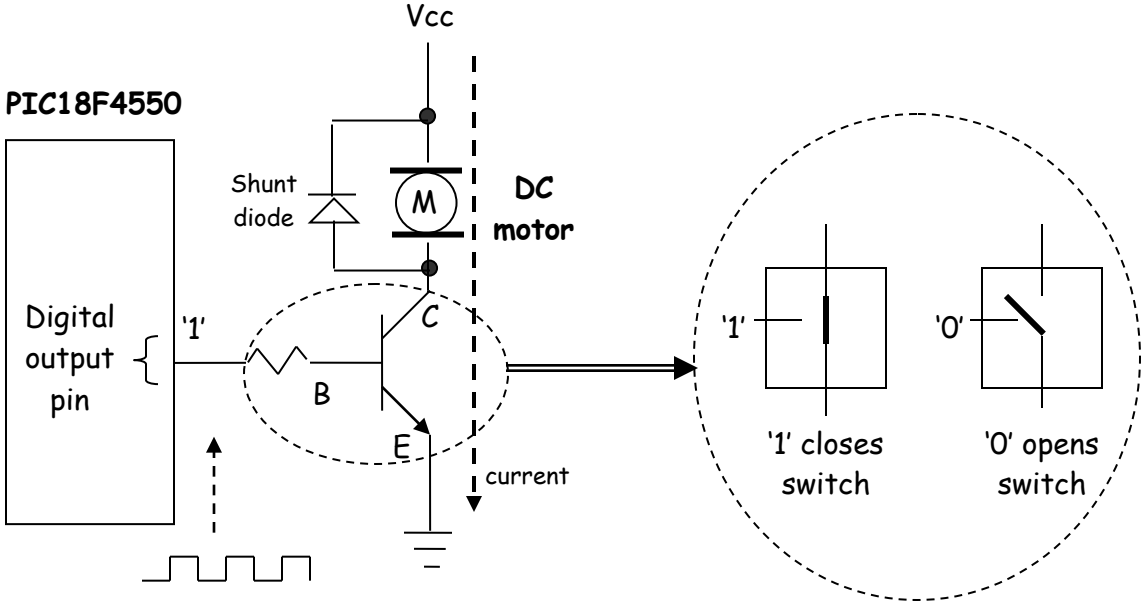
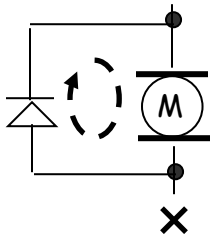


Figure 3.19 – DC Motor

☐ When the transistor is turned from ON to OFF, the **shunt diode** allows current to continue to flow for a short while. Otherwise, the motor circuit would be damaged.



☐ A rectangular wave can be produced at the PIC output pin to control the motor speed. When the **duty cycle** is high (*), the motor moves faster. Again, we will do this (so called **Pulse Width Modulation** or PWM) in the Timer topic.

(*) "Duty cycle" means what % of the rectangular wave is "high".

☐ It is also possible to control the motor **direction**, by using **4 transistors** arranged as a **bridge**.

- The motor circuit can also be used for a **solenoid**. A solenoid is an **electromagnet** which when energised, can suck in (or push out) an iron bar. This is useful in electronic lock, for instance.

3.18 Driving high-power device via mechanical relay – the need for power isolation

- All the devices discussed so far are "**low power**" devices that can share the same Vcc (5volt) with the micro-controller circuit.

- Sometimes, a motor / solenoid / valve etc that requires **higher voltage** (e.g. 12 / 24 V, or even 230 V ac) must be used in a micro-controller application.

- In such cases, a **separate power source** can be used for the high power device and a **mechanical relay** used to **isolate** it from the low power circuitry, as follows.

- When the transistor is on and the relay energised, the switch for the high power device will be closed.

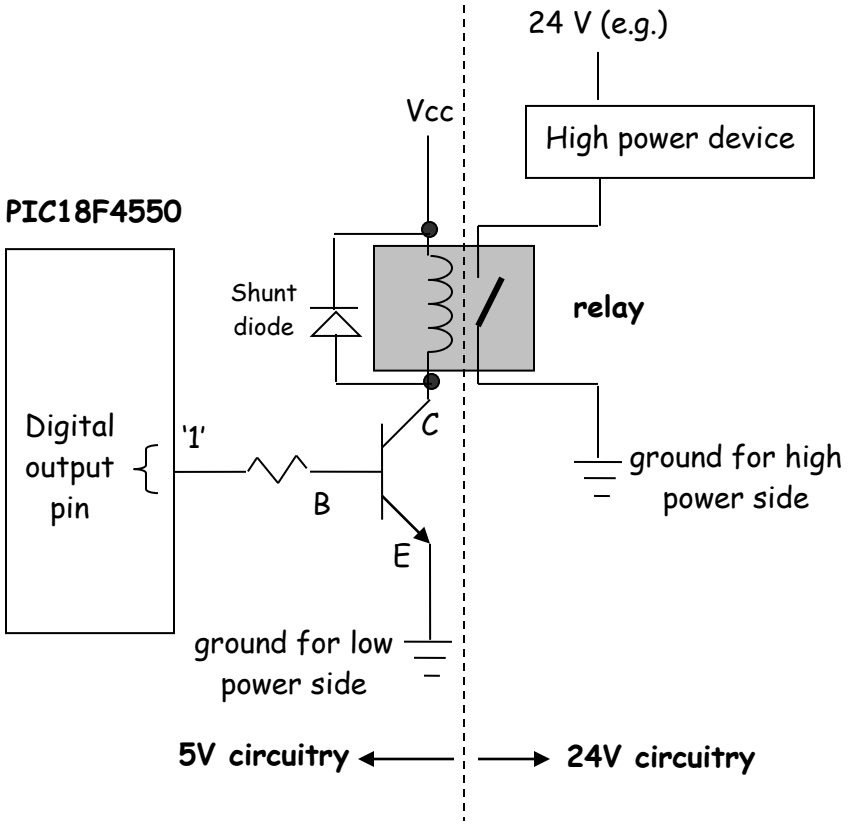


Figure 3.20 – Mechanical relay for power isolation

Chapter 4 – PIC18F4550's Analogue to Digital Converter

Chapter Overview

4.1 Introduction to A/D Converter

- Why are A/D & D/A converters required in a micro-controller application?

- Although a micro-controller can process data at high speed & accuracy (repeatedly without complaining!), it can only work with **digital** data.

- But, the outside world is **analogue**. All the physical quantities that matters – temperature, pressure, humidity, light intensity etc are analogue.

- A **transducer** is a device that converts a physical quantity e.g. length, weight, brightness, loudness into an electrical quantity – voltage or current – which is still analogue in nature.

- An **A/D** (analogue to digital) converter converts an analogue signal (e.g. 3.25V) to a corresponding digital number (e.g. $666_{10} = 1010011010_2$), a **D/A** (digital to analogue) converter does the reverse.

- As can be seen in the dashed box below, the PIC18F4550 has a built-in A/D converter module. If analogue output is to be produced, a D/A converter is still required.

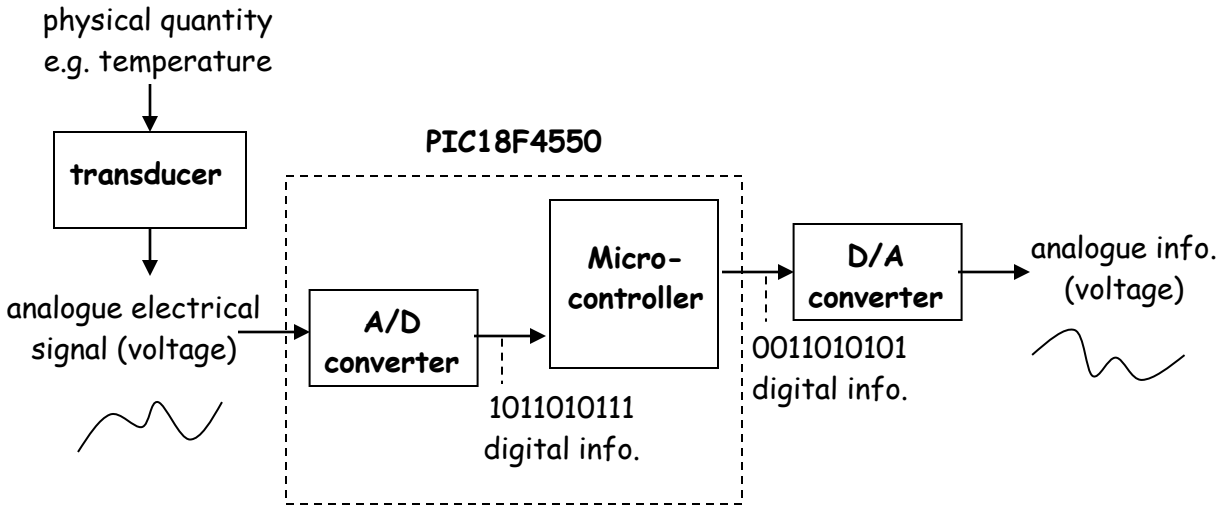


Figure 4.1 – Analogue world, digital controller

4.2 Examples of application

4.2.1 Measuring temperature using a digital thermometer

☐ In a **digital thermometer**, a pair of wires (made of dissimilar alloys) produces a milli-volt output corresponding to the temperature being measured. This output is amplified and then A/D converted for display on the LCD.

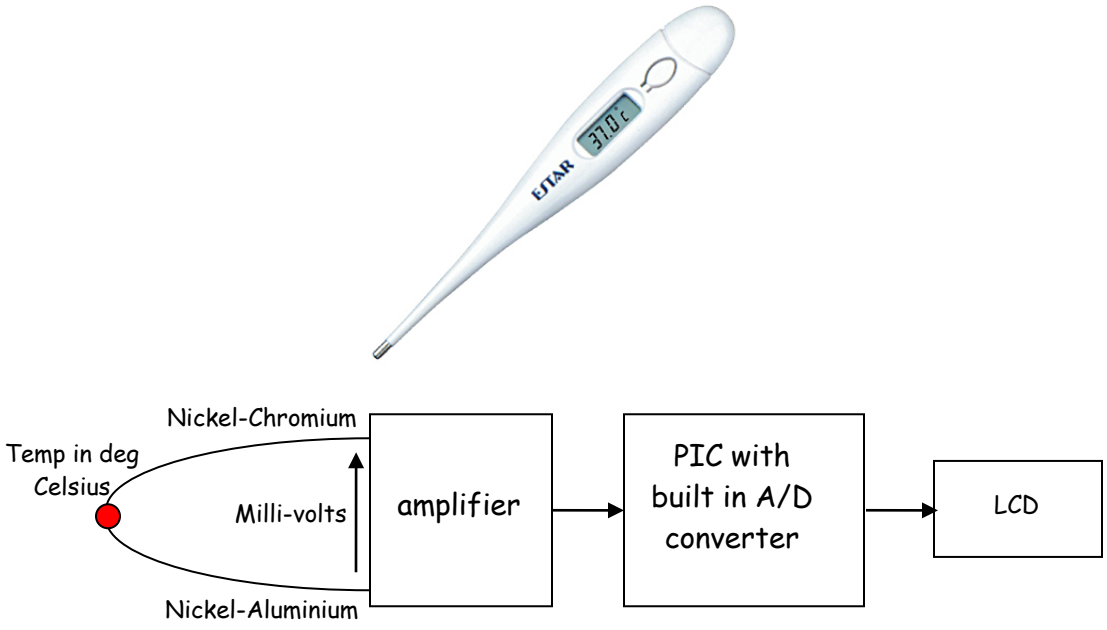


Figure 4.2 – Digital thermometer

4.2.2 Measuring brightness using a light dependent resistor

- The resistance of an LDR (**Light Dependent Resistor**) depends on the ambience brightness.
- By using it in a "**potential divider** circuit", a voltage output corresponding to the brightness can be produced.
- This can be A/D converted by the micro-controller and then used to make intelligent decision.
- For instance, if it is dark, switch on the lights (on the overhead bridge) automatically. The darker it is, the more lights will be switched on.

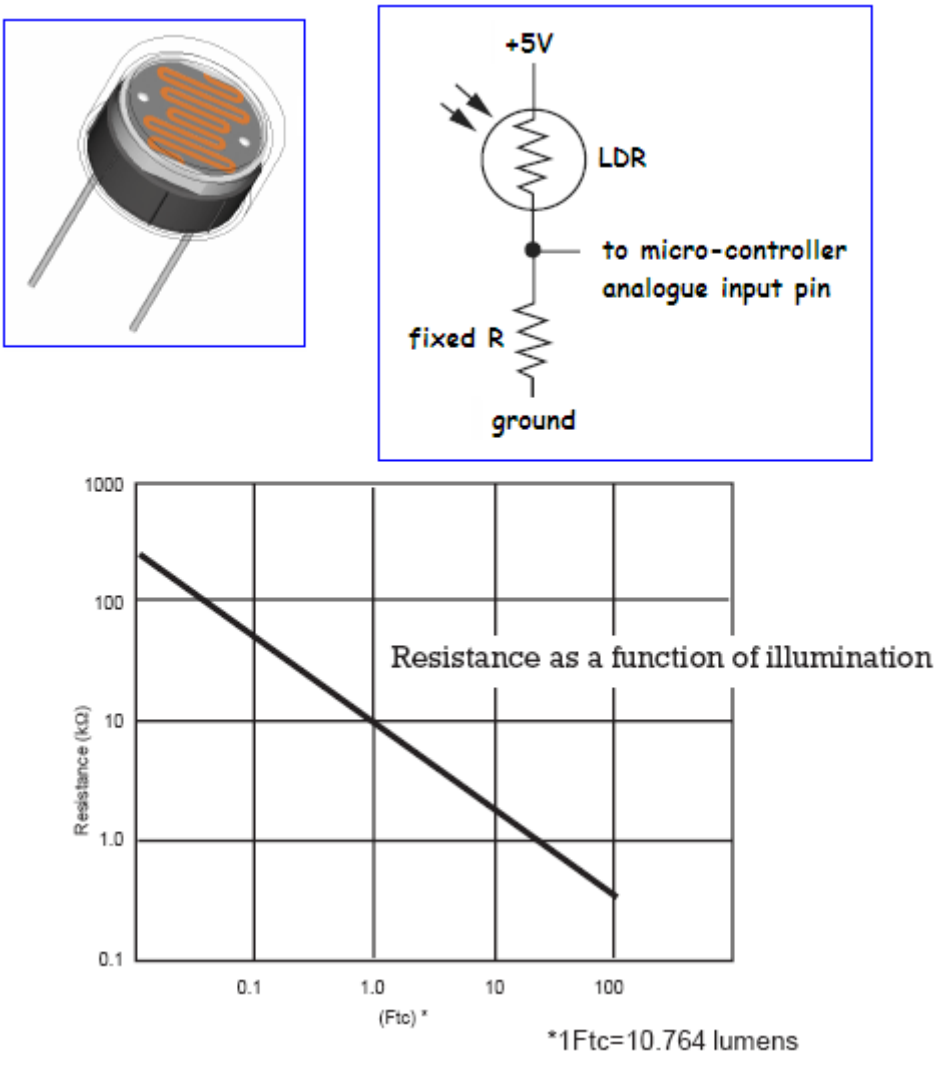
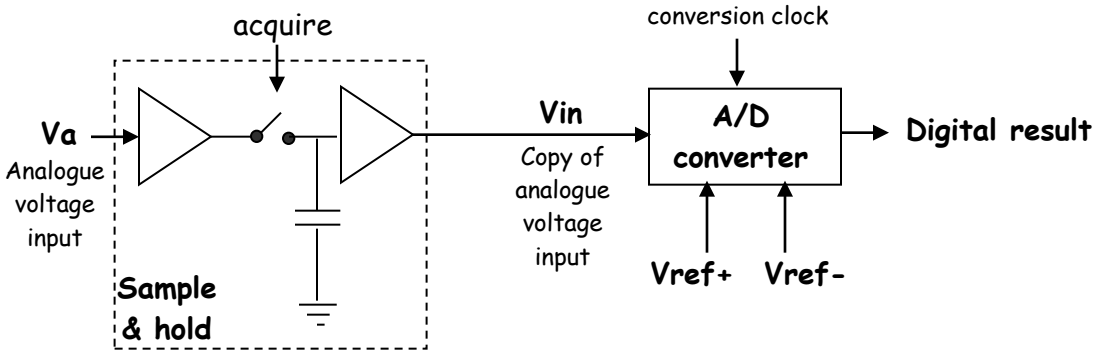


Figure 4.3 – Measuring brightness using LDR

4.3 A/D conversion in PIC18F4550 – sample calculations

☐ There are 2 main steps in the A/D conversion.

☐ First, during "**acquisition** time", the analogue voltage input (Va) is "**sampled & held**" – imagine the switch is closed (and later opened) to make (& freeze) an exact copy of Va (as Vin). After acquisition, Vin will remain unchanged even though Va continues to vary.



$$\text{Digital result} = \frac{V_{in} - V_{ref-}}{V_{ref+} - V_{ref-}} \times [\,2^{10} - 1\,]$$

Figure 4.4 – Sample and hold and A/D conversion formula

☐ Next, during "A/D **conversion**", the sampled input Vin is converted to its **10-bit** digital equivalent, using the **formula** given.

☐ V_{ref+} and V_{ref-} are the **reference voltages**. For simplicity, assume that V_{ref+} = Vcc i.e. 5V and V_{ref-} = Vss i.e. ground or 0V.

☐ If Vin = 3.25V, the digital result =

$$\frac{\text{Vin} - 0\text{V}}{5\text{V} - 0\text{V}} \times [\,2^{10} - 1\,] = \frac{3.25\text{V}}{5\text{V}} \times 1023 = 665_{10} = 1010011001_2$$

☐ If Vin < V_{ref-} , the conversion result would be V_{ref-} . Likewise, if Vin > V_{ref+} , the conversion result would be V_{ref+} .

☐ Determine the digital result if Vin = 2.5V and 5V and ground are used as voltage references. 

Working :

Answer : _____

☐ Determine Vin if the digital result is 0111001110 and 5V and ground are used as voltage references. 

Working :

Answer : _____

Quantization error (DIY)

- In a 10-bit A/D converter, there are only 2^{10} = 1024 different digital levels (0 – 1023).
- When an analogue input is digitized, because of the finite number of digital levels, the difference between the actual analogue value and the digital representation is called the quantization error.
- If the reference voltages used are 5V and ground i.e. an input range of 5V, the **maximum** quantization error is 5V / 1023 / 2 = 2.44 mV.
- To see why this is the case, consider a **simpler case** – a 4-bit A/D converter with an input voltage range of 0-3V. The truth table follows:

Analogue Input	Digital Representation			
Volts	D_3	D_2	D_1	D_0
0	0	0	0	0
0.2	0	0	0	1
0.4	0	0	1	0
0.6	0	0	1	1
0.8	0	1	0	0
1.0	0	1	0	1
1.2	0	1	1	0
1.4	0	1	1	1
1.6	1	0	0	0
1.8	1	0	0	1
2.0	1	0	1	0
2.2	1	0	1	1
2.4	1	1	0	0
2.6	1	1	0	1
2.8	1	1	1	0
3.0	1	1	1	1

Figure 4.5 – A/D converter with 0-3V input and 4-bit result

- The "**step size**" is 3V / (2^4-1) = 0.2V i.e. every 0.2 volt increase in the input voltage results in the binary result increasing by 1.
- So, 0V is represented as 0000_2 while 0.2V is represented as 0001_2. However, 0.0999V is still represented as 0000_2. Hence the biggest quantization error is **half** the "step size". And an A/D with more bits will be more accurate.

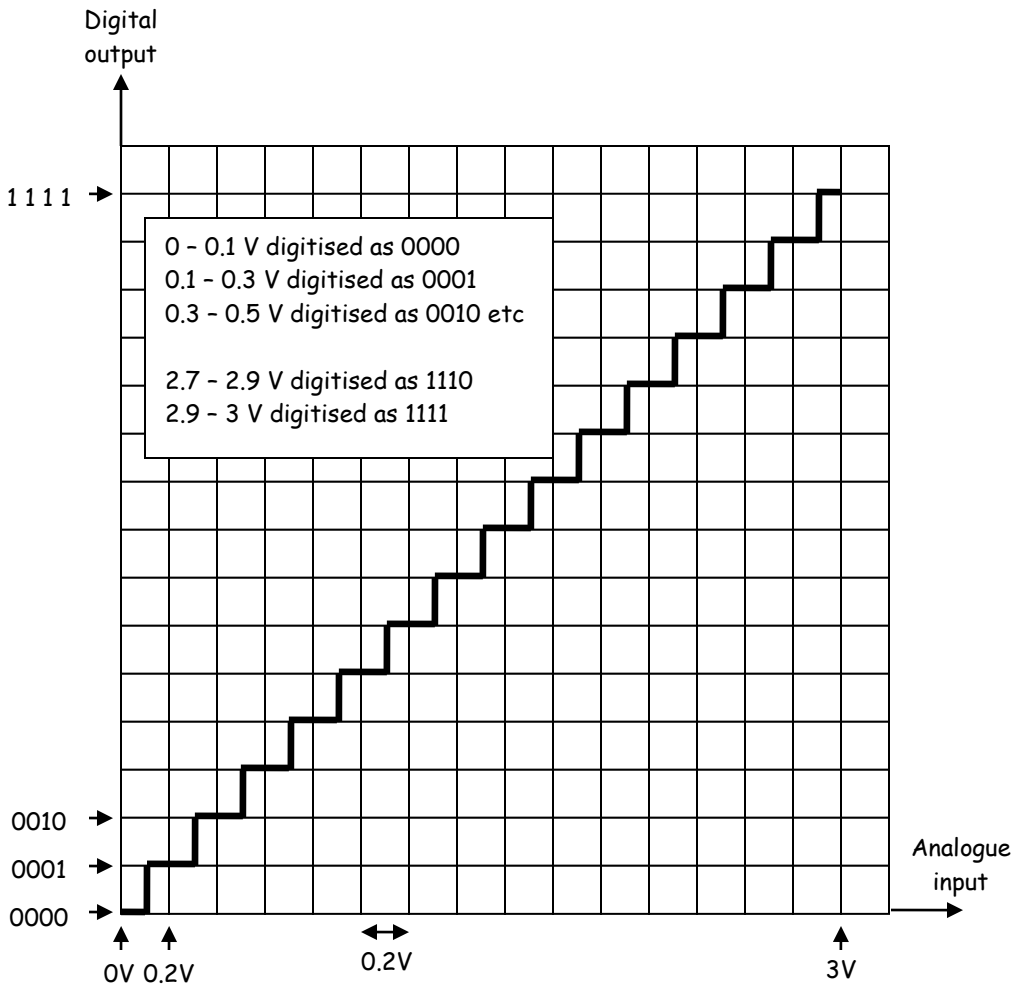


Figure 4.6 – Quantization in A/D converter

4.4 The structure of PIC18F4550's A/D converter module

- The PIC18F4550's A/D converter module has **13 input channels** (for the 40-pin devices). At any one time, only one channel can be selected for conversion. And a micro-controller application seldom requires all 13 inputs to be used – the unused pins can be configured as digital I/O pins.

- This module allows conversion of an analogue input to a corresponding **10-bit digital** number.

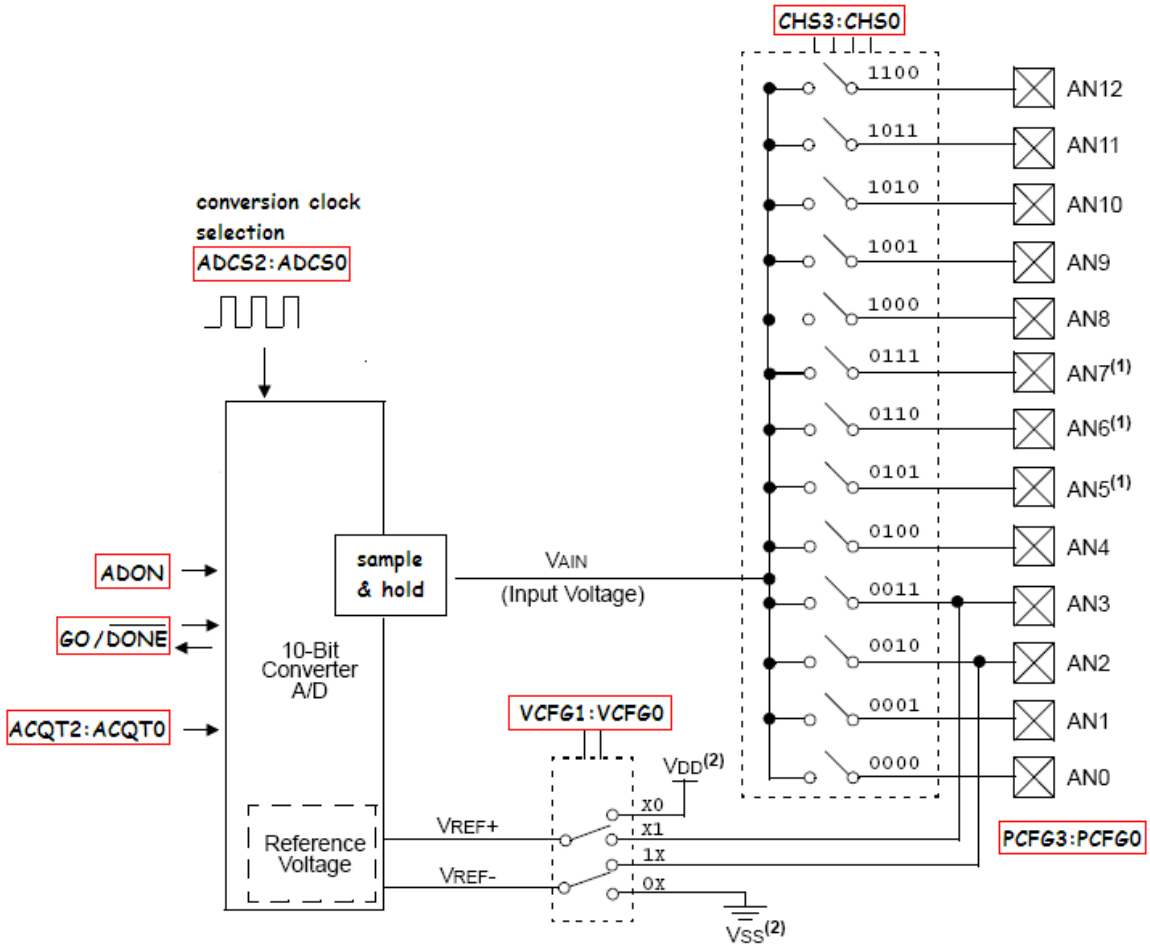


Figure 4.7 – Structure of PIC18F4550's A/D converter

- The user is allowed **many choices** e.g. which analogue input pin to use, what reference voltages (V_{REF+} & V_{REF-}) to use, how much time to sample the analogue input, which clock source to use for the A/D conversion etc.

- How these choices are made will be described in the next section. You may need to refer to the above diagram as you read the next section.

4.5 Registers associated with A/D operation

☐ The PIC18F4550's A/D converter module has **five registers**:
- o A/D Result High Register (ADRESH)
- o A/D Result Low Register (ADRESL)
- o A/D Control Register 0 (ADCON0)
- o A/D Control Register 1 (ADCON1)
- o A/D Control Register 2 (ADCON2)

☐ How these registers are used in the A/D conversion will be described below.

4.5.1 ADCON0

☐ The ADCON0 register controls the **operation of the A/D module**.

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	CHS3	CHS2	CHS1	CHS0	GO/$\overline{\text{DONE}}$	ADON
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 7-6 **Unimplemented:** Read as 'o'

bit 5-2 **CHS3:CHS0:** Analog Channel Select bits
- 0000 = Channel 0 (AN0)
- 0001 = Channel 1 (AN1)
- 0010 = Channel 2 (AN2)
- 0011 = Channel 3 (AN3)
- 0100 = Channel 4 (AN4)
- 0101 = Channel 5 (AN5)(1,2)
- 0110 = Channel 6 (AN6)(1,2)
- 0111 = Channel 7 (AN7)(1,2)
- 1000 = Channel 8 (AN8)
- 1001 = Channel 9 (AN9)
- 1010 = Channel 10 (AN10)
- 1011 = Channel 11 (AN11)
- 1100 = Channel 12 (AN12)
- 1101 = Unimplemented(2)
- 1110 = Unimplemented(2)
- 1111 = Unimplemented(2)

bit 1 **GO/$\overline{\text{DONE}}$:** A/D Conversion Status bit

When ADON = 1:
- 1 = A/D conversion in progress
- 0 = A/D Idle

bit 0 **ADON:** A/D On bit
- 1 = A/D converter module is enabled
- 0 = A/D converter module is disabled

Figure 4.8 – ADCON0

☐ **CHS3:CHS0** select the **analogue channel**.

☐ Complete the C code below to select channel 3 for conversion:

ADCON0bits.CHS3 = _____;
ADCON0bits.CHS2 = _____;
ADCON0bits.CHS1 = _____;
ADCON0bits.CHS0 = _____;

☐ For instance, on the "General I/O Board" used in the lab, a variable resistor is connected to RA0 (Port A Pin 0 – also pin 2 or AN0). To use AN0 to read this analogue input (which can vary continuously from 0 to 5 volts), set CHS3:CHS0 = 0000.

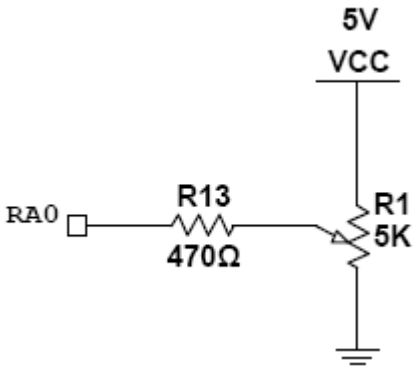


Figure 4.9 – Variable resistor connected to RA0/AN0

☐ Setting **ADON** = 1 **enables** the A/D converter module.

☐ C code: ADCON0bits.ADON = 1;

☐ Setting **GO** = 1 (with ADON set) **starts** the A/D conversion.

☐ C code: ADCON0bits.GO = 1;

☐ The same bit (GO/ $\overline{DONE}$) becoming 0 indicates that the A/D conversion is **complete**.

4.5.2 ADCON1

☐ The ADCON1 register configures the **voltage references** and the **functions of the port pins**.

U-0	U-0	R/W-0	R/W-0	R/W-0(1)	R/W(1)	R/W(1)	R/W(1)
—	—	VCFG1	VCFG0	PCFG3	PCFG2	PCFG1	PCFG0
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 7-6 **Unimplemented:** Read as '0'

bit 5 **VCFG1:** Voltage Reference Configuration bit (VREF- source)
1 = VREF- (AN2)
0 = VSS

bit 4 **VCFG0:** Voltage Reference Configuration bit (VREF+ source)
1 = VREF+ (AN3)
0 = VDD

bit 3-0 **PCFG3:PCFG0:** A/D Port Configuration Control bits:

PCFG3: PCFG0	AN12	AN11	AN10	AN9	AN8	AN7(2)	AN6(2)	AN5(2)	AN4	AN3	AN2	AN1	AN0
0000(1)	A	A	A	A	A	A	A	A	A	A	A	A	A
0001	A	A	A	A	A	A	A	A	A	A	A	A	A
0010	A	A	A	A	A	A	A	A	A	A	A	A	A
0011	D	A	A	A	A	A	A	A	A	A	A	A	A
0100	D	D	A	A	A	A	A	A	A	A	A	A	A
0101	D	D	D	A	A	A	A	A	A	A	A	A	A
0110	D	D	D	D	A	A	A	A	A	A	A	A	A
0111(1)	D	D	D	D	D	A	A	A	A	A	A	A	A
1000	D	D	D	D	D	D	A	A	A	A	A	A	A
1001	D	D	D	D	D	D	D	A	A	A	A	A	A
1010	D	D	D	D	D	D	D	D	A	A	A	A	A
1011	D	D	D	D	D	D	D	D	D	A	A	A	A
1100	D	D	D	D	D	D	D	D	D	D	A	A	A
1101	D	D	D	D	D	D	D	D	D	D	D	A	A
1110	D	D	D	D	D	D	D	D	D	D	D	D	A
1111	D	D	D	D	D	D	D	D	D	D	D	D	D

A = Analog input D = Digital I/O

Figure 4.10 – ADCON1

- **VCFG1:VCFG0** configures the **voltage reference**.
- If both bits are cleared, the VREF+ = VDD i.e. 5 V while VREF- = VSS i.e. 0 V.
- *C* code:
```
ADCON1bits.VCFG1 = 0;
ADCON1bits.VCFG0 = 0;
```
- As can be seen above, these reference voltages (VREF+ & VREF-) could also come from the pins AN3 and AN2.

- **PCFG3:PCGF0** divide the I/O pins between **analogue and digital usage**.
- The table below (or the PIC18F4550's pin diagram in Chapter 2) shows 13 (possible) analogue input pins.

Pin	33	37	34	36	35	10	9	8	7	5	4	3	2
	AN12	AN11	AN10	AN9	AN8	AN7	AN6	AN5	AN4	AN3	AN2	AN1	AN0
	RB0	RB4	RB1	RB3	RB2	RE2	RE1	RE0	RA5	RA3	RA2	RA1	RA0
A? D?	D	D	D	D	D	D	D	D	D	D	D	D	A

Figure 4.11 – Analogue or digital I/O pins

- Setting PCFG3:PCGF0 = 1110 configures pin 2 (AN0) as an analogue input and the rest as digital inputs.
- *C* code:
```
ADCON1bits.PCFG3 = 1;
ADCON1bits.PCFG2 = 1;
ADCON1bits.PCFG1 = 1;
ADCON1bits.PCFG0 = 0;
```
- As can be seen in the diagram on the previous page, if only two analogue inputs are required, AN0 and AN1 can be selected. If only three analogue inputs are required, AN0, AN1 and AN2 can be selected etc.

4.5.3 ADCON2

☐ The ADCON2 register configures the **A/D clock source**, **programmed acquisition time** and **justification**.

R/W-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
ADFM	—	ACQT2	ACQT1	ACQT0	ADCS2	ADCS1	ADCS0
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 7 **ADFM:** A/D Result Format Select bit
1 = Right justified
0 = Left justified

bit 6 **Unimplemented:** Read as '0'

bit 5-3 **ACQT2:ACQT0:** A/D Acquisition Time Select bits
111 = 20 TAD
110 = 16 TAD
101 = 12 TAD
100 = 8 TAD
011 = 6 TAD
010 = 4 TAD
001 = 2 TAD
000 = 0 TAD(1)

bit 2-0 **ADCS2:ADCS0:** A/D Conversion Clock Select bits
111 = FRC (clock derived from A/D RC oscillator)(1)
110 = FOSC/64
101 = FOSC/16
100 = FOSC/4
011 = FRC (clock derived from A/D RC oscillator)(1)
010 = FOSC/32
001 = FOSC/8
000 = FOSC/2

Figure 4.12 – ADCON2

☐ Complete the C code below to select left justification, acquisition time of 4 T_{AD}, F_{OSC}/64 as the clock source:

```
ADCON2bits.ADFM = _____;
ADCON2bits.ACQT2 = _____;
ADCON2bits.ACQT1 = _____;
ADCON2bits.ACQT0 = _____;
ADCON2bits.ADCS2 = _____;
ADCON2bits.ADCS1 = _____;
ADCON2bits.ADCS0 = _____;
```

☐ You can also write ADCON2 = 0 b 0 0 0 1 0 1 1 0; to achieve the same results.

Justification

☐ When an A/D conversion is completed, the 10-bit result is stored in the registers ADRESH and ADRESL, either **left-justified** or **right-justified**, depending on the value of the **ADFM** bit, as shown below. Note that the other bits are filled with 0's.

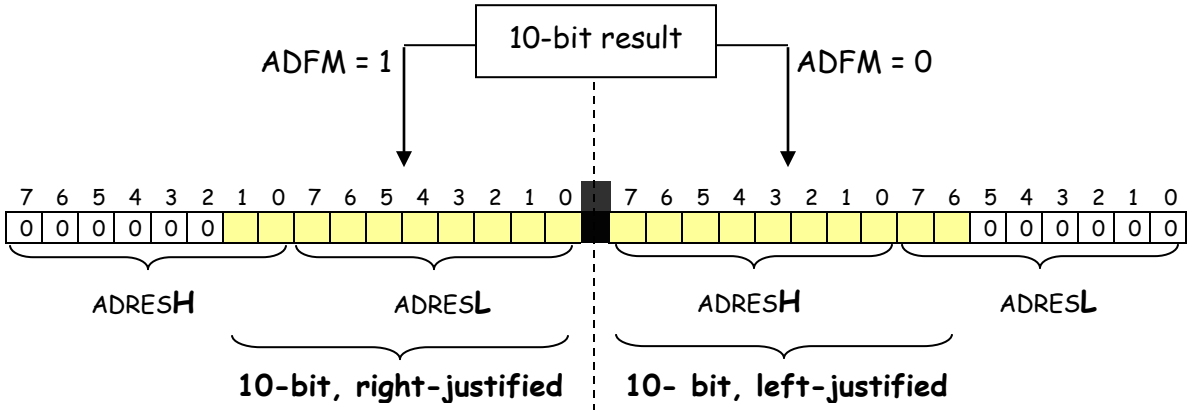


Figure 4.13 – Right or left justification

☐ Determine the 10-bit result of an A/D conversion, given that ADFM = 0, ADRESH = 0xA3 and ADRESL = 0x80.

Your answer: ADFM = 0 => _____ - justified

ADRESH = 0xA3 = 0b _____ ADRESL = 0x80 = 0b _____

ADRESH : ADRESL = 0b _____ _____

10- bit result, left-justified

The 10-bit result is _____

☐ If the above 10-bit result is to be displayed on 8 LED's (of an LED bar) connected to PORT D, which of the following is a better choice? Why?

(a) PORTD = ADRESH; or (b) PORTD = ADRESL;

Better choice: _____ Reason: _____

A/D clock source

- The A/D **conversion time per bit** is defined as T_{AD}. (T_{AD} is actually the period of the conversion clock.) The A/D conversion requires 11 T_{AD} per 10-bit conversion.
- The source of the A/D conversion clock is software selectable using **ADCS2:ADCS0**.
- For correct A/D conversions, the A/D conversion clock (T_{AD}) must be as **short** as possible but greater than the **minimum** T_{AD} i.e. ~0.7 us.

A/D CONVERSION REQUIREMENTS

Param No.	Symbol	Characteristic		Min	Max	Units	Conditions
130	TAD	A/D Clock Period	PIC18FXXXX	0.7	25.0(1)	µs	Tosc based, VREF ≥ 3.0V

Figure 4.14 – Minimum T_{AD}

- The table below shows the resultant T_{AD} times derived from the device operating frequencies and the A/D clock source selected.

TAD vs. DEVICE OPERATING FREQUENCIES

AD Clock Source (TAD)		Maximum Device Frequency	
Operation	ADCS2:ADCS0	PIC18FXXXX	PIC18LFXXXX(4)
2 Tosc	000	2.86 MHz	1.43 MHz
4 Tosc	100	5.71 MHz	2.86 MHz
8 Tosc	001	11.43 MHz	5.72 MHz
16 Tosc	101	22.86 MHz	11.43 MHz
32 Tosc	010	45.71 MHz	22.86 MHz
64 Tosc	110	48.0 MHz	45.71 MHz
RC(3)	x11	1.00 MHz(1)	1.00 MHz(2)

Note 1: The RC source has a typical TAD time of 4 ms.
2: The RC source has a typical TAD time of 6 ms.
3: For device frequencies above 1 MHz, the device must be in Sleep for the entire conversion or the A/D accuracy may be out of specification.
4: Low-power devices only.

Figure 4.15 – T_{AD} vs device operating frequencies

- For instance, if Fosc = 48 MHz and ADCS2:ADCS0 = 110, T_{AD} = 64 T_{OSC} = 64 / 48 MHz = 1.33 us (> the 0.7 us minimum specified).

Acquisition time

- For an accurate A/D conversion, the charge holding capacitor (refer to Figure 4.4) must be allowed to **fully charge** to the selected analogue input channel's voltage level i.e. the input channel must be sampled for at least the "**minimum acquisition time**" before a conversion is started.

- You may refer to the PIC18F4550 datasheet for the calculation of the "minimum acquisition time". (When it is difficult to do the calculation, large acquisition time is a safe choice.)

- **ACQT2:ACQT0** selects the **acquisition time**.

- Setting ACQT2:ACQT0 = 010 cause acquisition time to be equal to 4 T_{AD}. If T_{AD} = 1.33 us, acquisition time = 4 x 1.33 us = 5.32 us.

DIY

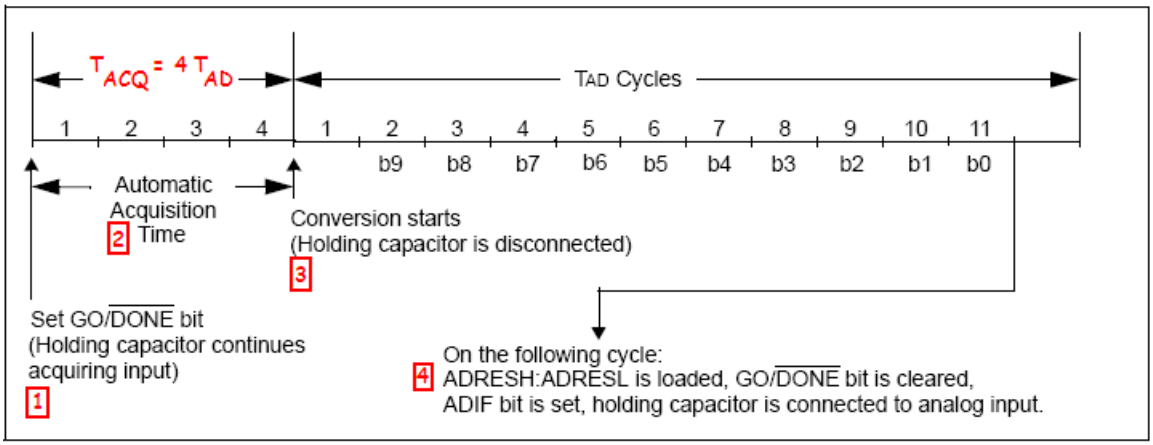


Figure 4.16 – Acquisition & A/D conversion

- As can be seen in the diagram above, after the GO bit is set, the holding capacitor continues to acquire the input for 4 T_{AD} i.e. "automatic acquisition time". ["Automatic acquisition time" does not happen when ACQT2:ACQT0 = 000. In this case, the user has to ensure enough acquisition time elapses before conversion is started with GO = 1 <- we will not use this.]

- After the acquisition time, conversion starts and it takes 11 T_{AD} before conversion is completed. Then the result loaded is loaded into ADRESH:ADRESL and the GO/ $\overline{DONE}$ bit cleared to indicated end of conversion. The holding capacitor is connected to analogue input for the next cycle.

4.6 The complete A/D operation sequence

- Note that it is possible for the completion of an A/D conversion to cause an **interrupt**. We will not use interrupt with A/D conversion in this chapter.

- Note also that the port pins needed as analogue inputs must have their corresponding **TRIS bits** set (input).

- The following steps should be followed to perform an A/D conversion:

  1. Configure the A/D module
     - configure voltage reference, and analogue/digital I/O (ADCON1)
     - select A/D input channel (ADCON0)
     - select A/D acquisition time (ADCON2)
     - select A/D conversion clock (ADCON2)
     - turn on A/D module (ADCON0)
  2. Wait the required acquisition time (if required).
  3. Start conversion
     - Set GO/$\overline{DONE}$) (ADCON0)
  4. Wait for A/D conversion to complete, by polling for the GO/$\overline{DONE}$) bit to be cleared.
  5. After a minimum wait of 3 T_{AD}, go to step 1 (or 2, if no changes to the configuration are required *,) for the next conversion.

- (*) If more than 1 analogue input channel is used, there is a need to go back to step 1 to "**switch channel**" i.e. the channels are sampled & converted one by one, in a predetermined order.

4.7 A programming example

☐ In the example below, a variable resistor is connected to AN0 and a program is written to **configure** the A/D converter and then to repeatedly **sample** the analogue input, **convert** to digital equivalent, and output the most significant 8-bits to the LED bar for **display**.

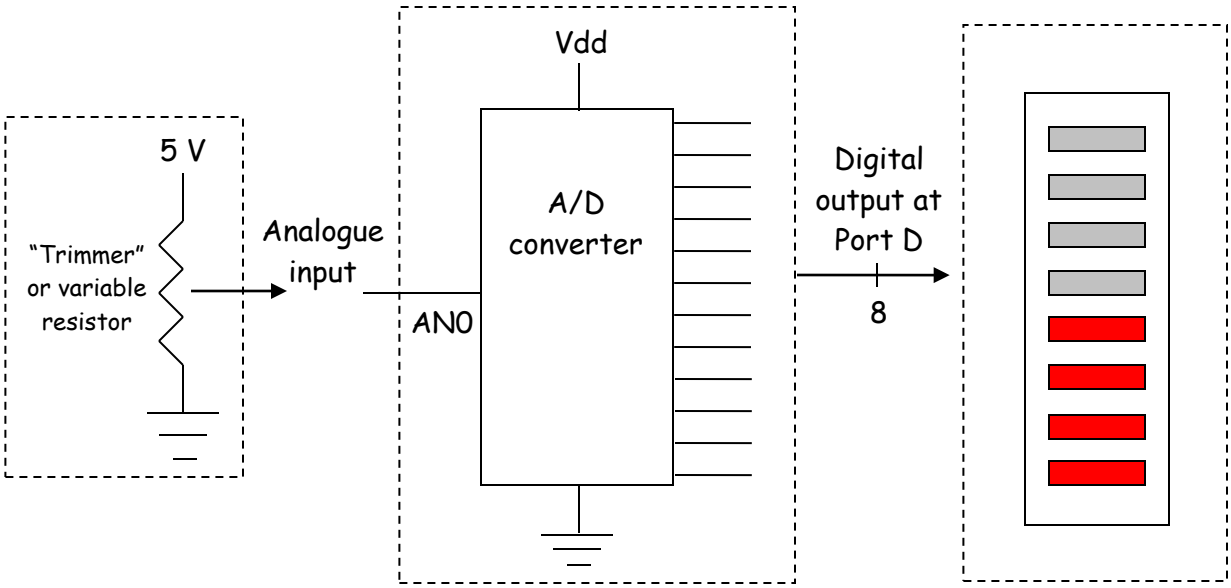


Figure 4.17 – Sampling the trimmer for display on the LED bar

```
main (void)
{
  TRISD = 0x00;          // configure Port D as output - for LED bar
  PORTD = 0x00;          // initialise Port D – all LED's off

  // configure A/D converter module & switch it on
  ADCON0 = 0 b 0 0 0 0 0 0 0 1;  // bits <5:2> = 0000, channel AN0 selected
                                 // bit <0> = 1, A/D activated (powered up)

 ADCON1 = 0 b 0 0 0 0 1 1 1 0;   // bit <5> = 0, Vref- = Vss (0V)
                                 // bit <4> = 0, Vref+ = Vdd (5V)
                                 // bits <3:0> = 1110, pin AN0 as analogue input

  ADCON2 = 0 b 0 0 0 1 0 1 1 0;  // bit <7> = 0, result left justified
                                 // bits <5:3> = 010, acquisition time = 4 TAD
                                 // bits <2:0> =  110, conversion clock = FOSC/64
  while (1)
    {
    ADCON0bits.GO = 1;           // start A/D conversion
    while (ADCON0bits.GO == 1); // wait here for /DONE to becomes 0
    PORTD = ADRESH;              // output upper 8-bit of result to LED bar
    }
}
```

Chapter 5 – A brief revision on C programming language

Chapter Overview

5.1 C language fundamentals – #define & #include directives, comments, constants, variables, operators

5.2 C language fundamentals – functions & program control statements

5.1 C language fundamentals – #define & #include directives, comments, constants, variables, operators

☐ You have studied C programming language in module such as "structured programming". This short chapter serves as a quick revision on the important concepts.

5.1.1 #define directive

☐ The **#define** directive is used to define a **string constant** that is **substituted** into the source code before the code is compiled.

☐ If you include the directive *#define switch_1 PORTBbits.RB5*, writing *if (switch_1 == 0)* is the same as writing *if (PORTBbits.RB5 == 0)*.

☐ The advantages are improved source code i. **readability** (switch_1 is more meaningful than PORTBbits.RB5) and ii. **maintainability** (one day, you may decide to move the switch to PORTBbits.RB6 and all you need to do is to change the definition).

☐ This is especially so when the string constants are used in **many places** and when they provide **short cuts** to complex expressions.

5.1.2 #include directive

☐ The **#include** directive **inserts** the full text from **another file**, at the point where the #include appears in the source code.

☐ The inserted file may contain any number of valid C statements.

☐ You have used #include <delays.h> in the lab.

5.1.3 Comments

- Comments are used to describe and explain the source code. The compiler **ignores** all comments.

- Examples:

```
// This is a single line comment

/* This is a multiple line comment. It is good to include lots of comments in
your source code, so as to remind yourself why you write the code in a
certain way. Others in the project team will also be able to follow the code.
The comment can stretch over many lines and is terminated by a */
```

5.1.4 Constants

- The value of a constant **never changes**. In C language, a constant can be a **number**, a **single character** or a **character string**.

- Examples:

12	Decimal
0x0C	Hexadecimal for 12
0b1100	Binary for 12
"Hello World !"	a string constant
'y'	a char constant

- Example:

```
const float PI = 3.14; // this declares a floating point constant
// it is a common practice to capitalize constants
```

5.1.5 Variables

- The value of a variable **changes** as the program executes. The variable can be of the following **data types**:

Data type	Remarks	How many bits?	Range
void	no type	not applicable	not applicable
char	single character	8	-128 to 127
unsigned char	unsigned single character	8	0 to 255
int	integer	16	-32768 to 32767
unsigned int	unsigned integer	16	0 to 65535
long	long integer	32	-2,147,483,648 to 2,147,483,647
unsigned long	unsigned long integer	32	0 to 4,294,967,295
Float / double	floating point	32	1.7549435 E-38 to 6.80564693 E+38

Figure 5.2 – Basic data types

- **Negative numbers** are represented in the **two's complement** format.

- All variables must be **declared** before they are used. When you declare a variable, choose an appropriate data type.

- Examples:

```
char reply; // a char to store 'y' (yes) or 'n' (no) as reply

unsigned char count; // this is a byte (or 8 bit) that stores
                    // non-negative count value

int num; // this is a 16-bit variable that stores both +ve & -ve
         // values, from -32768 to 32767
```

- A variable *num_student* is to be declared to keep track of the number of students in a lecture theatre. What is the appropriate data type? Why?

Answer : Data type : _____

Reason : _____

- A **local variable** – declared inside a function (including the main function) - can only be used by statements within that function. A local variable must be declared before the executable statements in the function.

- A **global variable** – declared outside the functions – can be used by all the functions in the source code. However, the global variable must be declared before the function that uses it.

- Example:

```
unsigned int ticks; // global variable can be used by any function that
                    // appears after this
main (void)
{
        unsigned char count; // local variable can be used by main only
        ….
}
```

5.1.6 Operators

☐ **Arithmetic operators** may be used with any basic data type:

+	addition
-	subtraction
*	multiplication
/	division

☐ Examples:

-b	negative b
count – 163	variable count minus 163
a * b	a multiplied by b
c / b	c divided by b

☐ **Relational operators** compare two values and return **1** (for **true**) or **0** (for **false**) based on the comparison

>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	equal to
!=	not equal to

☐ Examples:

count > 0	is count greater than 0 ?
value <= MAX	is value less than or equal to MAX ?
input != BADVAL	is input not equal to BADVAL ?

☐ **Logical operators** return **1** (for **true**) or **0** (for **false**) depending on the result of the operations.

&&	Logical AND
||	Logical OR
!	Logical NOT

☐ Example:

```
Lecture_Open && (Students > MIN)

- checking that the  lecture theatre is open and there are
        enough students turning up
```

☐ **Bitwise operators** – operation is performed bit by bit on the number(s).

&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
>>	Right shift
<<	Left shift

☐ Examples

Flags & MASK	if MASK = 0b00000111, the upper 5 bits of Flags will be filtered or masked off, while the lower 3 bits will be unchanged.
Flags ^ 0x07	invert bits 0, 1 and 2
Val << 2	shift left by 2 bits i.e. multiply by 4

☐ What is achieved by the statement status = status | 0b00000010; ?

Answer : _____

☐ **Assignment operator** – the most common operation in any source code is to assign a value to a variable.

☐ Example:

```
a = 0x05;
```

5.2 C language fundamentals – functions & program control statements

5.2.1 Functions

- Functions are the basic **building blocks** of a C program. All **executable statements** must reside within a function.

- ***main*** is also a function – there must be a *main* function in any C program.

- A **function prototype** should be declared before the function is called. A function prototype declares the **return type**, **function name**, and **types of parameters** for a function, but no other statements.

- A function must be written before it can be used.

- Example:

```
unsigned char AddOne (unsigned char x);  // prototype terminated by ;

// actual function below
unsigned char AddOne (unsigned char x) // x is a local variable
{
        return (x + 1);
}
```

☐ What is wrong with the following C program?

```
unsigned char K;

unsigned char add_one (unsigned char L);  // function prototype

unsigned char add_one (unsigned char L);
{
        unsigned char M;
        M = L + 1;
        return (M);
}

main (void)
{
        unsigned char N, S;
        N = 168;
        S = add_one (N);
        …
}
```

Answer : _____

5.2.2 Program control statements

☐ Program control statements control the **flow of execution** in a source code. They use **relational** and **logical operator** to decide **which path** to take. **Loops** are also used to **repeat**edly execute an instruction.

☐ **if statement** is a "**conditional** statement". The statement associated with the *if* is executed based on the outcome of a condition – if the condition evaluates to **non-zero**, the statement is **executed**. Otherwise, it is skipped.

☐ Example:

```
if (second > 59)
{
        second = 0;
        minute += 1;
}
```

- **if-else statement**: The most general form of "if-else" is

```
if (expression) statement1 else statement2
```

- Example #1:

```
if (a < b)
        smaller = a;  // single statement
else
        smaller = b;
```

- Example #2:

```
if (num < 0)
{
        num = 0;  // multiple statement
        Valid = 0;
}
else
        Valid = 1;
```

- Example #3:

```
if (num == 1)
        DoCase1 (); // DoCase1 is a function that handles case 1
else if (num == 2)
        DoCase2 ();
else if (num == 3)
        DoCase3 ();
else
        DoInvalid ();
```

- For multiple if – else if – else if – else, you may consider changing it to a **switch statement**:

```
switch (expression)
{
        case const1 : statement1(s) ;
                      break;
        case const2 : statement2(s) ;
                      break;
        :
        default :     statement3(s) ;
                      break;
}
```

- **for loop** is used to repeat a statement (or a set of statements) a number of times.

- The general form is

```
for (initialization; expression; increment)
        statement
```

- Example:

```
unsigned char i;

for (i = 0; i < 10; i++)
        do_something ();
```

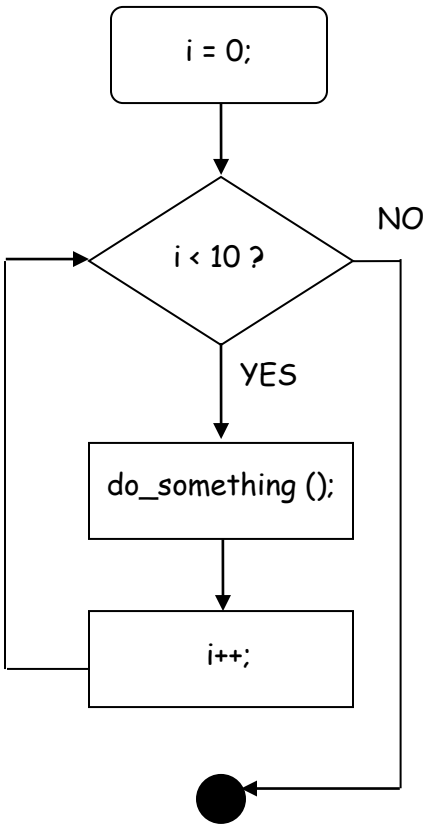


Figure 5.3 – Flowchart for "for loop"

- **while loop** is used to repeat a statement (or a set of statements) until a condition becomes true (or false).

- The general form is

```
while (expression) statement
```

- Example #1:

```
i = 0;

while (i < 10)
{
        do_something ();
        i = i + 1;
}
```

☐ Complete the **flowchart** below with YES NO i = 0; i < 10 ? i++; do_something (); 

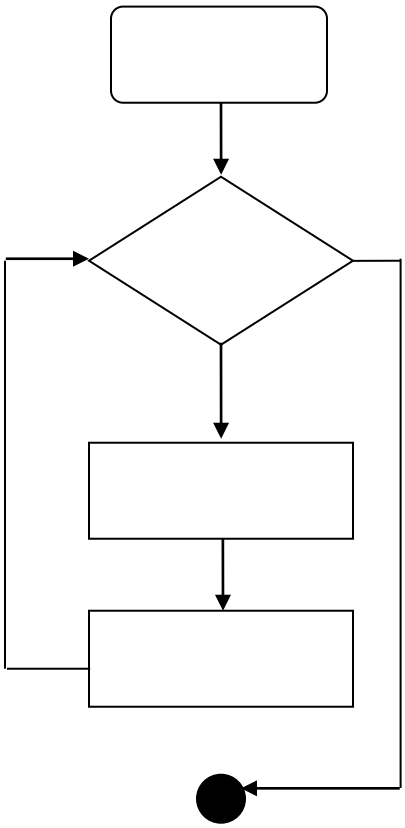


Figure 5.4 – Flowchart for "while loop"

☐ Example #2:

```
count = 0;
read_switch ();

while (switch_not_pressed)
{
        delay_1_sec (); // delay 1 second
        count = count + 1;
        read_switch ();
}
```

Chapter 6 – PIC18F4550's Programmer Timer / Counter

Chapter Overview

6.1 Why are timers / counters needed in micro-controller applications?

☐ Many micro-controller applications have **timing requirements**.

☐ For instance, you may want to switch off the light 30 seconds after the last person leaves a room:

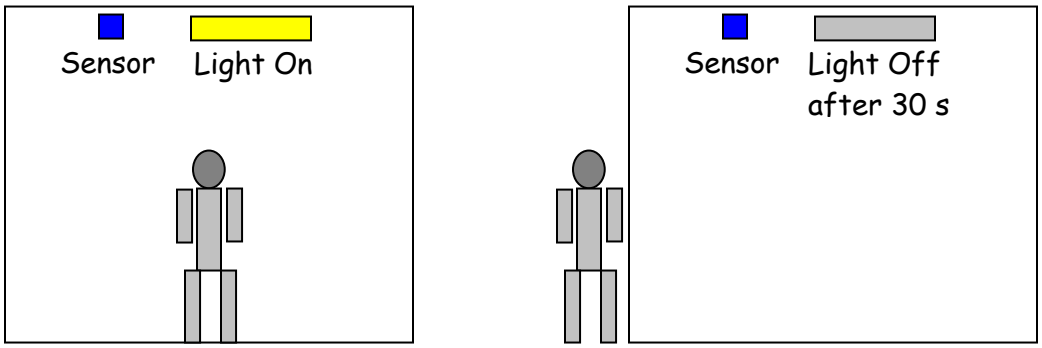


Figure 6.1 – Automatic room light

☐ Sometimes you may want to measure a **time duration**, for instance, how long has a signal been set to high?

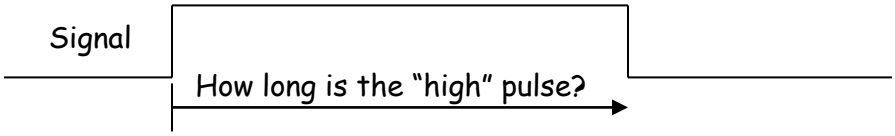


Figure 6.2 – Measuring pulse width

☐ You will learn how to **schedule an event** (e.g. light off 30 s after person leaves) and how to measure the elapsed time etc in the next section.

- Note that the two examples are "**opposite**": In scheduling an event, you wait for certain duration to pass before you do something. In measuring the elapsed time, you wait for an event (pulse goes from high to low) before you stop the "stop watch" and see how long it has been.

- So what is the difference between a **timer** and a **counter**?

- A counter counts up every time a **clock pulse** arrives:

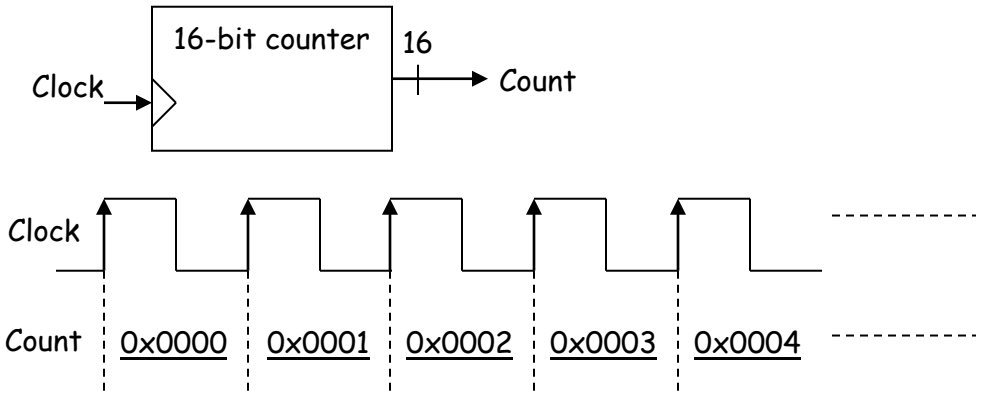


Figure 6.3 – 16-bit counting

- If the clock period is **fixed** and **known**, the counter becomes a "timer" i.e. a device that lets you know how much time has passed.

- For instance, clock period = 10 us, a count of 1,234 is equivalent to 12,340 us or 12.34 ms.

- The clock could be generated by **events** – for instance, boxes (on a conveyor belt) passing an infra-red sensor which produces pulses that can be used as a clock signal:

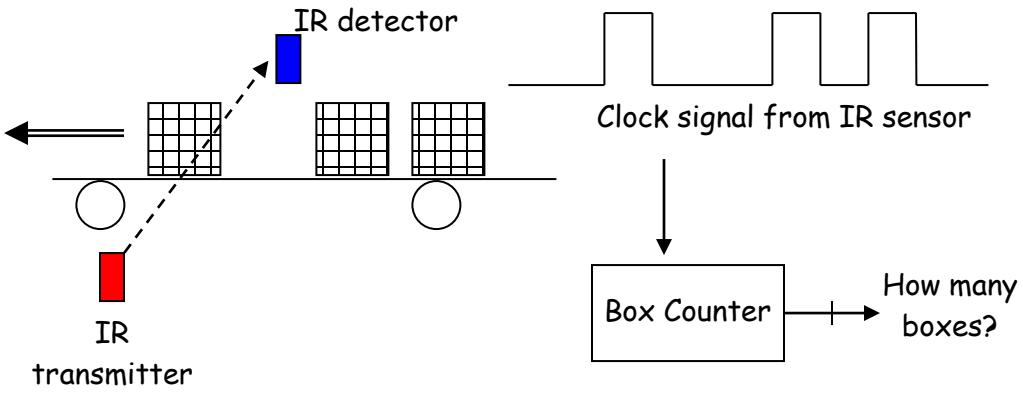


Figure 6.4 – Box counting

- In this case, the clock period is neither fixed nor known. So, the counter tells you how many boxes have moved pass the IR sensor but not how much time has passed. It is just a "counter".

6.2 Programmable timers / counters in PIC18F4550

- PIC18F4550 has **4 timers**: Timer0 to Timer3.

- The **key features** of each is highlighted in the table below:

Feature\Timer	Timer0	Timer1	Timer2	Timer3
Counting range	8-/16-bit timer or counter	16-bit timer or counter	8-bit timer (TMR2) and period (PR2) registers	16-bit timer or counter
Pre-scaler	1, 2, 4…256	1, 2, 4, 8	1, 4, 16 Also post-scaler 1, 2, 3,…6	1, 2, 4, 8
Clock source	internal/ external	internal/ external	internal	internal/ external
Interrupt	on overflow	on overflow	on TMR2 & PR2 match	on overflow
Other special features	edge select for external clock	has its own internal oscillator	can compare register value	can use TMR1 internal oscillator

Figure 6.5 – Key features of Timer0 to Timer3

- As the timers are very **similar**, we will not study everything. We will learn how to use **Timer0** well, and then learn the special feature of **Timer2**.

6.3 Using Timer0 to schedule an event

- Before we go into the details of Timer0, let's look at a **simple scenario** – you want an LED to be turned on, 0.1 second **after** a button is pressed.

- You can write a C program to achieve this:

```
while (PORTBbits.RB0 == 1); // wait for "active low" button at RB0 to
                            //    be pressed
Delay…                      // 0.1 s delay
PORTDbits.RD0 = 1;          // turn on "active high" LED at RD0
```

- In the above example, the turning ON of the LED is **scheduled** to take place 0.1 second after the trigger i.e. "button pressed".

- Let's see how the 0.1 s delay can be achieved using **Timer0**.

6.3.1 Registers associated with Timer0 operation

☐ Timer0 operation uses the following registers: TMR0H, TMR0L, T0CON and the TMR0IF bit of the INTCON register. The key points are highlighted below.

TMR0H and **TMR0L** (Timer0 High and Low Registers)

TMR0H								TMR0L							
D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

These two 8-bit registers together form a **16-bit timer/counter**.

INTCON (Interrupt Control Register) bit 2
-**TMR0IF** (Timer0 Interrupt "overflow" Flag)

					TMR0IF		

TMR0IF is set to 1 whenever the timer **overflows** i.e. count from FFFF to 0000 (in 16-bit mode) or from FF to 00 (in 8-bit mode).

Other bits are associated with the operations of **other peripherals** e.g. ADC etc.

Figure 6.6 – Timer0 registers TMR0H, TMR0L, INTCON

T0CON (Timer0 Control Register)

TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0

TMR0ON D7 Timer0 **ON** and **OFF control** bit
1 = Enable (start) Timer0
0 = Stop Timer0

T08BIT D6 Timer0 **8-bit / 16-bit selector** bit
1 = Timer0 is configured as an 8-bit timer/counter
0 = Timer0 is configured as a 16-bit timer/counter

T0CS D5 Timer0 **clock source select** bit
1 = External clock from RA4/T0CK1 pin
0 = Internal clock (Fosc/4 from XTAL oscillator)

T0SE D4 Timer0 **source edge select** bit
1 = Increment on H-to-L transition on T0CK1 pin
0 = Increment on L-to-H transition on T0CK1 pin

PSA D3 Timer0 **pre-scaler assignment** bit
1 = Timer0 clock input bypasses pre-scaler
0 = Timer0 clock input comes from pre-scaler output

T0PS2:T0PS0

D2	D1	D0	Timer0 **pre-scaler selector**
0	0	0 = 1:2	Pre-scale value (Fosc/4/2)
0	0	1 = 1:4	Pre-scale value (Fosc/4/4)
0	1	0 = 1:8	Pre-scale value (Fosc/4/8)
0	1	1 = 1:16	Pre-scale value (Fosc/4/16)
1	0	0 = 1:32	Pre-scale value (Fosc/4/32)
1	0	1 = 1:64	Pre-scale value (Fosc/4/64)
1	1	0 = 1:128	Pre-scale value (Fosc/4/128)
1	1	1 = 1:256	Pre-scale value (Fosc/4/256)

Note that the use of the pre-scaler "steps down" the clock that reaches the timer/counter. For instance, an oscillator frequency of 48MHz used with a pre-scale value of 16 gives the timer/counter a clock with period = 1/(48MHz/4/16).

Figure 6.7 – Timer0 register T0CON

☐ Give the binary pattern to be written to the T0CON register to turn on Timer0 as a 16-bit timer using internal clock (Fosc/4) and a pre-scale value of 32.

Answer:

T0CON (Timer0 Control Register)

TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0

T0CON = 0b_____;

☐ The following block diagram shows in a "**schematic** way", how the various registers affect the Timer0 operations. (optional)

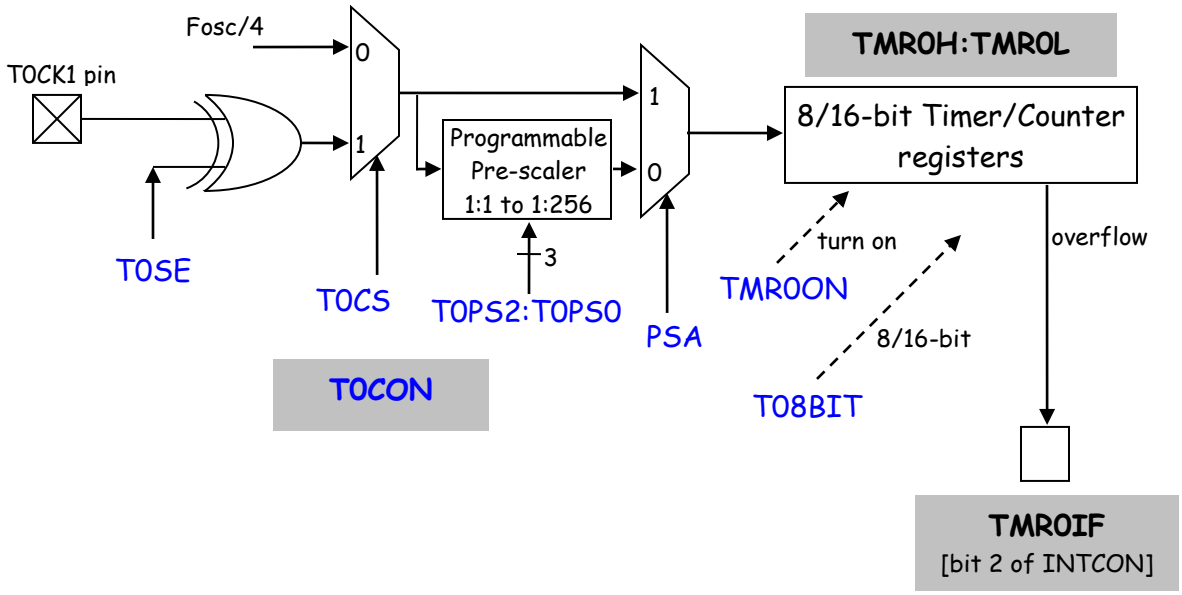


Figure 6.8 – Timer0 block diagram

6.3.2 Timer0 calculations

Example 1

☐ Assuming the crystal oscillator (with Fosc = 48 MHz) is used as the clock source and the **pre-scaler is not used**, how long does it take for Timer0 (used as 16-bit timer) to count from 0x0000 to 0xFFFF and roll over?

Answer:

Period = 1 / (Fosc/4) = 1 / (12MHz) = 1/12 us

Counting from 0x0000 to 0xFFFF and then rolling over is counting up 2^{16} = 65536 times.

So, total time = 65536 x (1/12 us) = 5.4613 ms

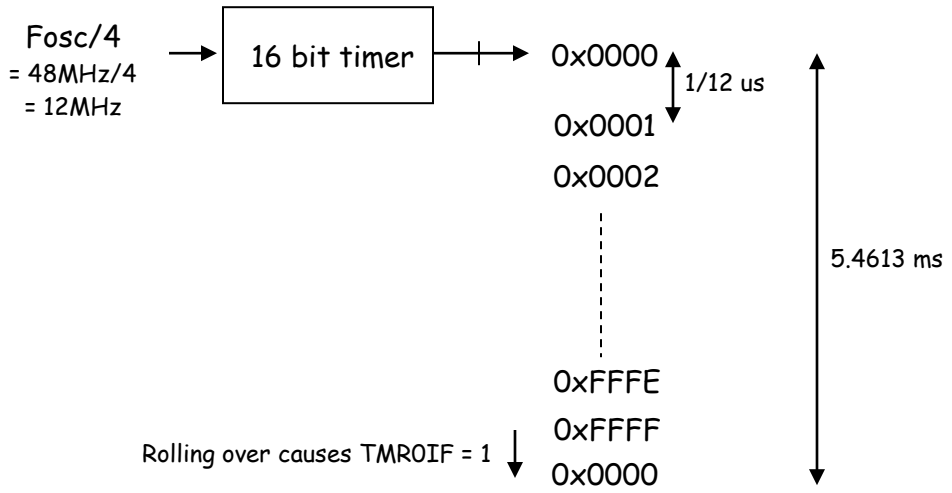


Figure 6.9 – Timer0 calculation – example 1

Example 2

☐ Assuming the crystal oscillator (with Fosc = 48 MHz) is used as the clock source and the **pre-scale value of 256 is used**, how long does it take for Timer0 (used as 16-bit timer) to count from 0x0000 to 0xFFFF and roll over?

Answer:

The pre-scaler "slows down" the clock (i.e. clock frequency is "stepped down" by 256 times. Delay will become 256 times as long, as shown below:

Period = 1 / (Fosc/4/**256**)

Total time = 65536 x (**256** x 1/12 us) = 5.4613 ms x **256** = 1.3981 second

[This is the **longest delay** using Fosc = 48 MHz.]

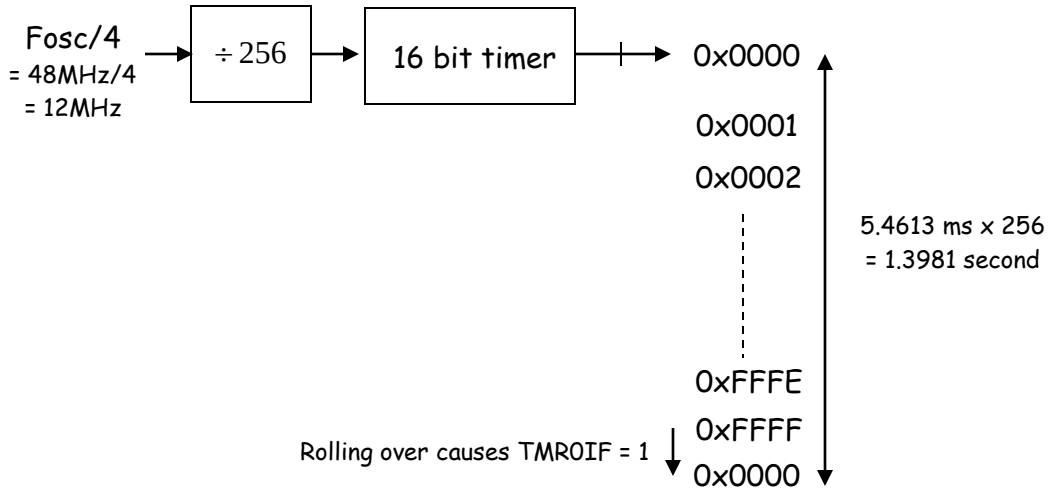


Figure 6.10 – Timer0 calculation – example 2

Example 3

☐ Assuming the crystal oscillator (with Fosc = 48 MHz) is used as the clock source and the **pre-scale value of 32 is used**, what is the **starting count** value in order that when the timer overflows (i.e. counts from 0xFFFF to 0x0000), exactly 0.1 second has elapsed?

Answer:

Step 1: *Calculate the period of the clock signal that reaches the timer.*

With pre-scale value of 32, Period = 1 / (Fosc/4/32) = 2.667 us

Step 2: *Calculate the number of counts equivalent to a given duration.*

A duration of 0.1 second is equivalent to counting up (0.1s / 2.667 us =) 37500 times.

Step 3: *Determine the count value to start from, by subtracting the answer from Step 2 from 65536.*

Since the overflow occurs when 65535 → 0, we should start counting from (65536-37500 =) 28036 or 0x6D84

[You can use the Microsoft window's calculator or your own calculator to do the conversion.]

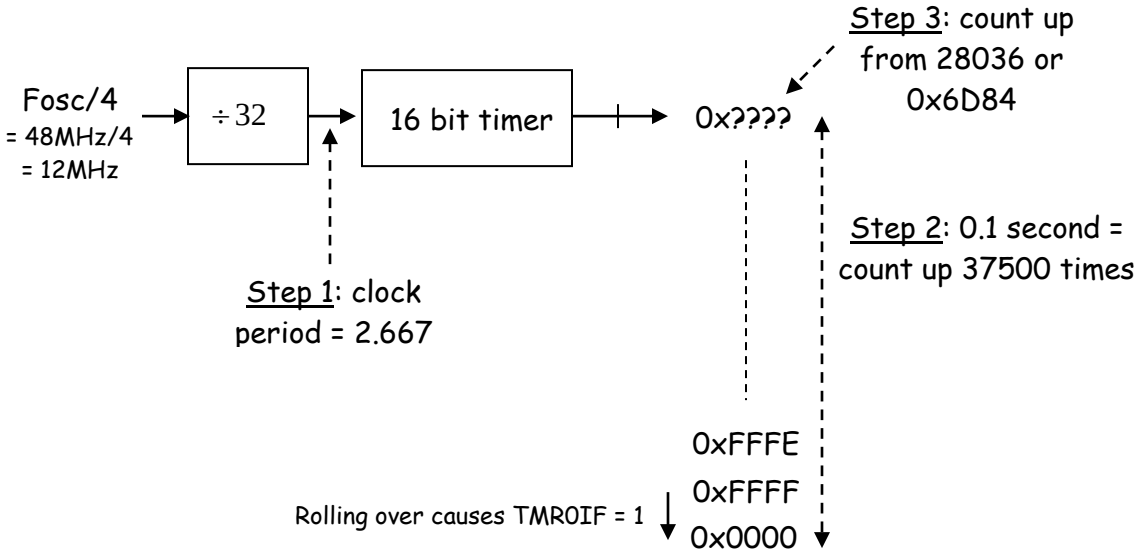


Figure 6.11 – Timer0 calculation – example 3

☐ Using the pre-scale value of 16, determine the starting count value so that when the timer overflows, exactly 0.025 second has elapsed. [DIY]

Step 1: *Calculate the period of the clock signal that reaches the timer.*

With pre-scale value of 16, Period = _____

Step 2: *Calculate the number of counts equivalent to a given duration.*

A duration of 0.025 second is equivalent to counting up _____ times.

Step 3: *Determine the count value to start from, by subtracting the answer from Step 2 from 65536.*

Since the overflow occurs when 65535 → 0, we should start counting from

_____

6.3.3 Programming Timer0 to schedule an event

☐ Let's go back to the original problem: turning on an LED 0.1 second after a button is pressed.

☐ As computed in Example 3 in the previous section, a 0.1 second delay can be achieved using Fosc = 48 MHz, a pre-scale value of 32 and a starting count value of 0x6D84.

☐ The complete ***C* program** looks like this:

```
T0CON = 0b10000100; // Timer on, 16-bit, Fosc/4, pre-scaler 32

while (PORTBbits.RB0 == 1); // wait for button to be pressed

INTCONbits.TMR0IF = 0; // clear the flag

TMR0H = 0x6D; // always write to TMR0H before TMR0L
TMR0L = 0x84;
while (INTCONbits.TMR0IF == 0); // wait here for Timer0 overflow
// the previous 3 lines gives a delay of 0.1 second

PORTDbits.RD0 = 1; // turn on LED
```

Figure 6.12 – Code to add a delay using Timer0

Steps to add a delay using Timer0

1. T0CON register is used to **configure** the timer (8 or 16 bit, clock source, clock edge if external clock is used, pre-scale value) and to **turn on** the timer. The timer will then start counting up, at the rate of the selected clock source.
2. An appropriate "**starting count**" value is written to the TMR0H:TMR0L registers. TMR0H should be written to first, before TMR0L.
3. Wait for the timer to "**overflow**", by checking the TMR0IF bit of the INTCON register.

Lengthening the delay [DIY]

☐ There are several ways to lengthen the **delay** that Timer0 can produce in this manner –

1. Use Timer0 as 16-bit, instead of 8-bit.
2. Use a lower frequency crystal (not easy for an assembled board).
3. Use external clock and apply a slower clock at T0CK1.
4. Use a bigger pre-scale value.
5. Write a smaller number to TMR0H:TMR0L.

6.3.4 Scheduling a periodic event using Timer0 [DIY]

☐ How do you schedule a **periodic event** e.g. for the alarm to sound at 6 am every morning? Or for an LED to be toggled every 0.1 second?

☐ The complete ***C* program** to toggle an LED every 0.1 second follows:

```
T0CON = 0b10000100; // Timer on, 16-bit, Fosc/4, pre-scaler 32

while (1) // loop forever
{
INTCONbits.TMR0IF = 0; // clear the flag

TMR0H = 0x6D; // always write to TMR0H before TMR0L
TMR0L = 0x84;
while (INTCONbits.TMR0IF == 0); // wait here for Timer0 overflow
// the previous 3 lines gives a delay of 0.1 second

PORTDbits.RD0 = ~ PORTDbits.RD0; // toggle RD0
}
```

Figure 6.13 – Code to schedule a periodic event using Timer0

Steps to schedule a periodic event using Timer0

1. Use T0CON to **configure** & **turned on** the timer, which will then start counting.
2. Write a "**starting count**" to TMR0H:TMR0L.
3. Check TMR0IF bit of INTCON register for timer **overflow**.
4. Do "something" e.g. toggle LED, then go back to 1. to repeat.

Delay using Timer

- Since **Delay** can also be introduced easily with a loop that does nothing, why bother with Timer?
- Answer 1. Timer can be used with **interrupt** (which will be covered in the next chapter), so the PIC can do other things while waiting for the time to pass.
- Answer 2.Timer that uses external clock becomes an "**event counter**" (i.e. how many times an event happens, but we will not discuss this usage).

6.4 Using Timer0 to measure the elapsed time [DIY]

- As mentioned, measuring elapsed time is "opposite" to scheduling an event.
- Let's try a simple application, measuring the pulse width of a signal:

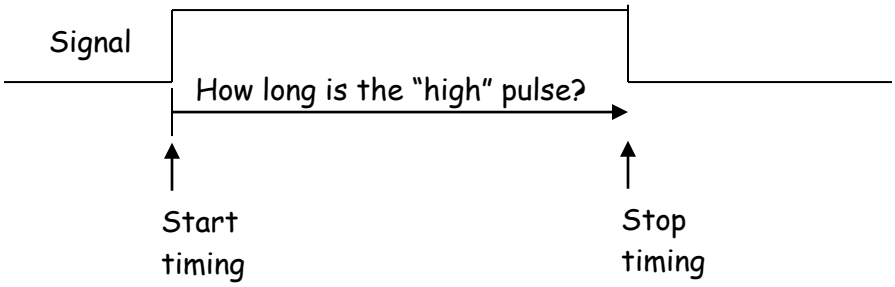


Figure 6.14 – Measuring pulse width

Steps to measure a time duration using Timer0

1. The timer is **configured** & **turned on**.
2. When the "low to high" transition occurs, the timer is "**reset**" to start counting from **0x0000** (this is similar to resetting the stop watch).
3. When the "high to low" transition occurs, the timer is **stopped** from counting further.
4. The final **count** (multiplied by the **clock period**) gives the **elapsed time** i.e. the pulse width.

☐ We assume that the counter has **not overflowed**. You can modify the code below if the pulse width is long and overflow is likely.

```
T0CON = 0b10000100; // Timer on, 16-bit, Fosc/4, pre-scaler 32
                    // other pre-scale value can be chosen

while (PORTBbits.RB0 == 0); // wait for signal at RB0 to go high

TMR0H = 0x00; // reset timer -- always write to TMR0H first
TMR0L = 0x00;

while (PORTBbits.RB0 == 1); // wait for signal at RB0 to go low

T0CONbits.TMR0ON = 0; // stop timer

// the time elapsed is now in TMR0H:TMR0L. To extract the 16-bit
// value, use unsigned integer time_elapsed as follows:

TempLow = TMR0L; // read TMR0L first
TempHigh = TMR0H;
Time_elapsed = TempHigh * 256 + TempLow;

// Time_elapsed x 2.667 us then gives the "real time" elapsed
```

Figure 6.15 – Code to measure a time duration using Timer0

6.5 Using Timer2 for PWM control of DC motor speed

6.5.1 Timer 2 – a brief intro

☐ As mentioned earlier, the four PIC18F4550 timers are very similar. We will not study everything – after learning Timer0 in details, we will now learn the **special features** of **Timer2** (-- refer to Figure 6.5 to note the **key differences** between Timer2 and Timer0).

☐ The diagram below shows how Timer2 works.

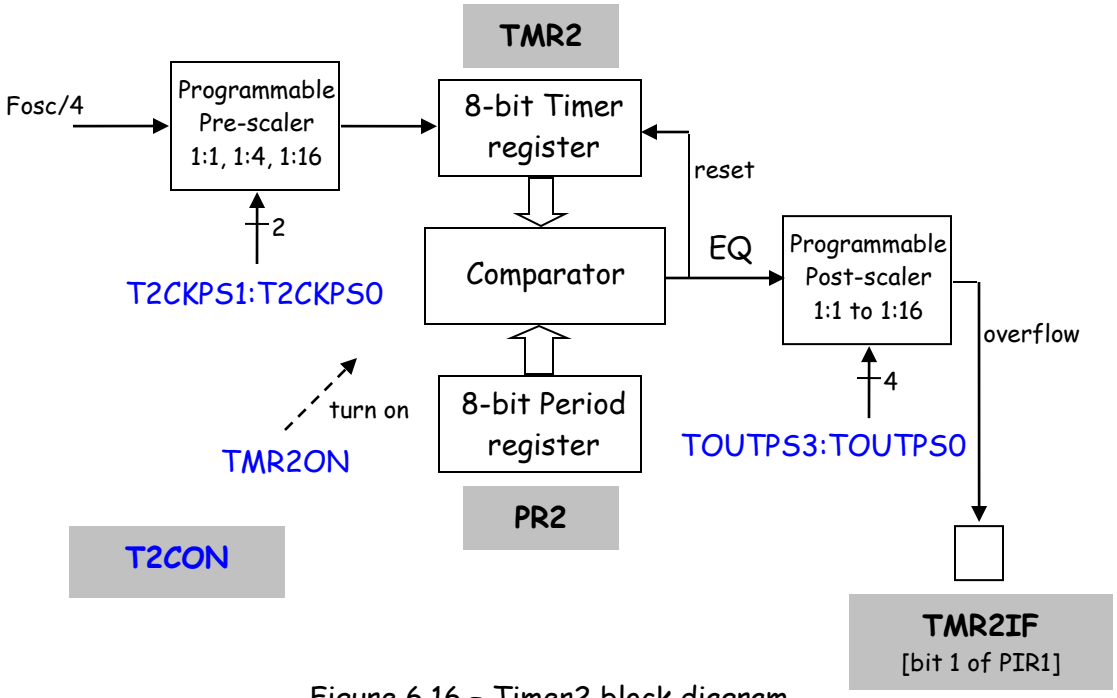


Figure 6.16 – Timer2 block diagram

☐ Timer2 is an **8-bit timer**. The 8-bit register of Timer2 is called **TMR2**.

☐ Timer2 also has an 8-bit register called the **period register** (**PR2**).

☐ We can set the PR2 register to a **fixed** value and Timer2 will **increment** from 00 until it **matches** the value in PR2. Then, the equal (**EQ**) signal will **reset** TMR2 to 00, and depending on the post-scaler, may raise the **TMR2IF** flag.

☐ The clock source for Timer2 is **Fosc/4** with the options of both **pre-scaler** and **post-scaler**, as shown in the figure above.

☐ There is no external clock source for Timer2. In other words, it cannot be used as a counter.

☐ The **registers** associated with Timer2 operations are the following:

TMR2 (Timer2 Register)

D7	D6	D5	D4	D3	D2	D1	D0

PR2 (Period Register)

D7	D6	D5	D4	D3	D2	D1	D0

PIR1 (Peripheral Interrupt Flag Register 1) bit 1
–**TMR2IF** (Timer2 Interrupt "overflow" Flag)

						TMR2IF	

TMR2 is reset to 0 whenever TMR2 matches PR2.
If no post scaler is used, TMR2IF is set to 1 whenever the match occurs.
With post scaler, many matches have to occur before TMR2IF is set.

T2CON (Timer2 Control Register)

D7	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0

D7 – not used

TOUTPS3:TOUTPS0 D6 D5 D4 D3 — Timer2 output **post-scaler selector**
0 0 0 0 = 1:1 Post-scale value
0 0 0 1 = 1:2 Post-scale value
0 0 1 0 = 1:3 Post-scale value
…
1 1 1 1 = 1:16 Post-scale value

TMR2ON D2 — Timer2 **ON** and **OFF control** bit
1 = Enable (start) Timer2
0 = Stop Timer2

T2CKPS1:T2CKPS0 D1 D0 — Timer2 clock **pre-scaler selector**
0 0 = 1:1 Pre-scale value
0 1 = 1:4 Pre-scale value
1 X = 1:16 Pre-scale value

Figure 6.17 – Timer2 registers TMR2, PR2, PIR1 and T2CON

☐ Let's see how we can write a C-program to turn on the LED at pin RD0 of PORTD when TMR2 reaches value 100:

```
#include …

void main (void)
{
  TRISDbits.TRISD0 = 0;          // configure RD0 as output
  PORTDbits.RD0 = 0;             // turn OFF LED at RD0

  T2CON = 0x00;                  // Timer2, no pre-/post- scaler
  TMR2 = 0x00;                   // TMR2 = 0
  PR2 = 100;                     // load period register 2
  T2CONbits.TMR2ON = 1;          // turn ON Timer2

  while (PIR1bits.TMR2IF == 0);  // wait for TMR2IF to be raised

  PORTDbits.RD0 = 1;             // turn ON LED at RD0

  T2CONbits.TMR2ON = 0;          // turn OFF Timer2
  PIR1bits.TMR2IF = 0;           // clear flag

  while (1);                     // stay here
}
```

Figure 6.18 – Code to add a **delay** using Timer2

Longest Delay [DIY]

☐ What is the **longest delay** that can be introduced using Timer2?

☐ To introduce the longest delay, use the maximum pre-scale value (16) and the maximum post-scale value (16), and load the period register with the biggest 8-bit number (255).

☐ Effective, Timer2 is clocked by Fosc / 4 / 16 / 16 (= 46875 Hz). This gives a clock period of 1/46875 s. TMR2 counting from 0 to 255 and back to 0 takes 256/46875 s = 0.00546 s or 5.46 ms (max delay).

6.5.2 PWM (Pulse Width Modulation)

- ☐ PWM (**Pulse Width Modulation**) is frequently used to control the **speed** of a **DC motor**.
- ☐ For instance, a small DC motor can be made to turn by applying a d.c. voltage (e.g. 5V) to it.
- ☐ When 5V is applied, it turns at a certain speed.
- ☐ When the 75% duty cycle wave is applied, the motor **slows down** – this is because effectively, it is getting 5V x 75% or 3.75V d.c.
- ☐ When the 50% duty cycle wave is applied, the motor slows down further, as it is effectively getting 2.5V d.c.

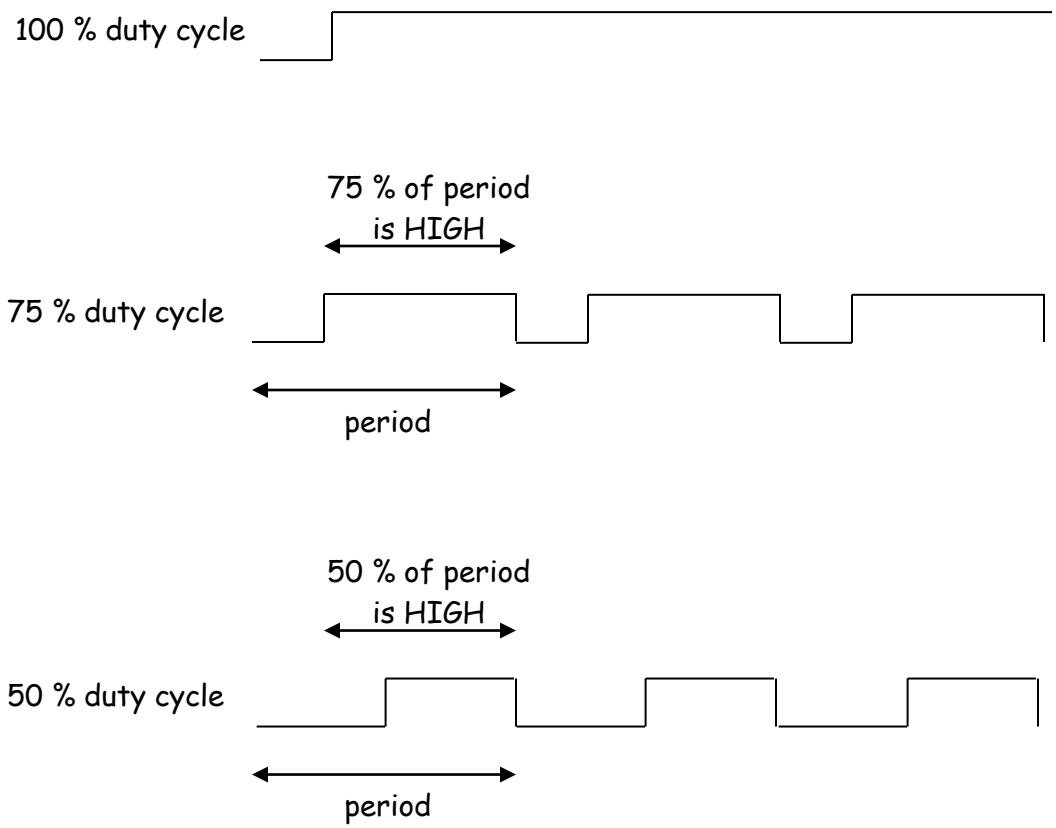


Figure 6.19 – 100%, 75% and 50% duty cycle waves

☐ Sketch a 5 kHz, 25% duty cycle wave: 

6.5.3 PIC18F4550's PWM capability

☐ PIC18F4550 has a ***CCP*** (Capture Compare) module.

☐ **Compare** means this: when a timer counts up to a certain value, an event is triggered (e.g. a certain pin set to high or toggled or…) i.e. we can program an event to occur at a certain time.

☐ **Capture** means the opposite: when an event (e.g. low to high transition) occurs at a certain pin, the content of a timer is copied i.e. we know when the event occurs.

☐ We will not study Capture-Compare in this module. However, the CCP module comes with PWM capability and we will focus on just this.

Point ◆ 1

☐ The PWM output will appear at RC2 i.e. Port C, Pin 2. So RC2 must be made an output pin. TRISCbits.TRISC2 = 0; [This pin is also called CCP1.]

Point ◆ 2

☐ The bottom 4 bits of the CCP1 Control Register (CCP1CON) must be set to 1100 for PWM operation: CCP1CON = 0b00001100;

CCP1CON (CCP1 Control Register)

D7	D6	DC1B1	DC1B0	CCP1M3	CCP1M2	CCP1M1	CCP1M0

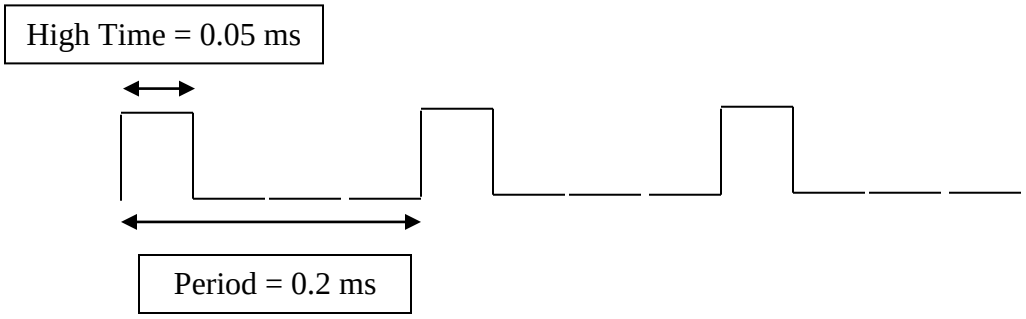
1 1 0 0 for PWM mode

Other combinations for Capture / Compare modes not studied in this module

Figure 6.20 – CCP register CCP1CON

Sample calculations

- Assuming a 5 kHz, 25% duty cycle wave is required:
- Period = 1 / 5k = 0.2 ms. High Time = Period x 25% = 0.2 ms x 25% = 0.05 ms.



- The "**Period**" is specified using PR2 register using the formula below:

PWM period = (PR2 + 1) x 4 x N x Tosc

where PR2 is the 8-bit Period register
N = Timer2 prescaler value of 1, 4 or 16, as set in T2CON register
Tosc = 1 / Fosc, where Fosc = 48 MHz in our case

- Assuming pre-scaler N = 16, and Tosc = 1 / 48M.

☐ Substitute into formula:

PWM period = (PR2 + 1) x 4 x N x Tosc

0.2m = (PR2 + 1) x 4 x 16 x (1 / 48M)

=> PR2 = 149 Point ◆ 3

☐ The "**High Time**" is specified using another register called CCPR1L, as follows:

High Time = 25% of Period

25% x 149 = 37.25 = 37 (truncate the decimal portion)

=> CCPR1L = 37 Point ◆ 4

☐ Determine the appropriate PR2 & CCPR1L values to generate a 4 kHz, 50% duty cycle waveform. You can use any suitable pre-scaler value. Fosc = 48 Mhz. [DIY]

☐ *Here, the "High Time" is specified using the 8-bit CCPR1L register. The accuracy can be extended to 10 bits (i.e. 2 decimal places) if necessary. The details can be found in PIC18F4550 datasheet.*

PWM programming

☐ We will now put everything together to program the PIC18F4550 to produce a 5 kHz, 25% duty cycle PWM wave:

```
TRISCbits.TRISC2 = 0; // RC2 as output          Point ◆ 1
CCP1CON = 0b00 00 1100; // PWM mode             Point ◆ 2

PR2 = 149;  // Period                           Point ◆ 3
CCPR1L = 37; // High Time                       Point ◆ 4

TMR2 = 0; // start from 0

T2CON = 0b0 0000 1 10; // no post-scale, Timer2 on, pre-scale 16

while (1)
{
// PWM wave generated continuously at RC2, without further code
// PIC free to do other things……
}
```

Output waveforms (optional)

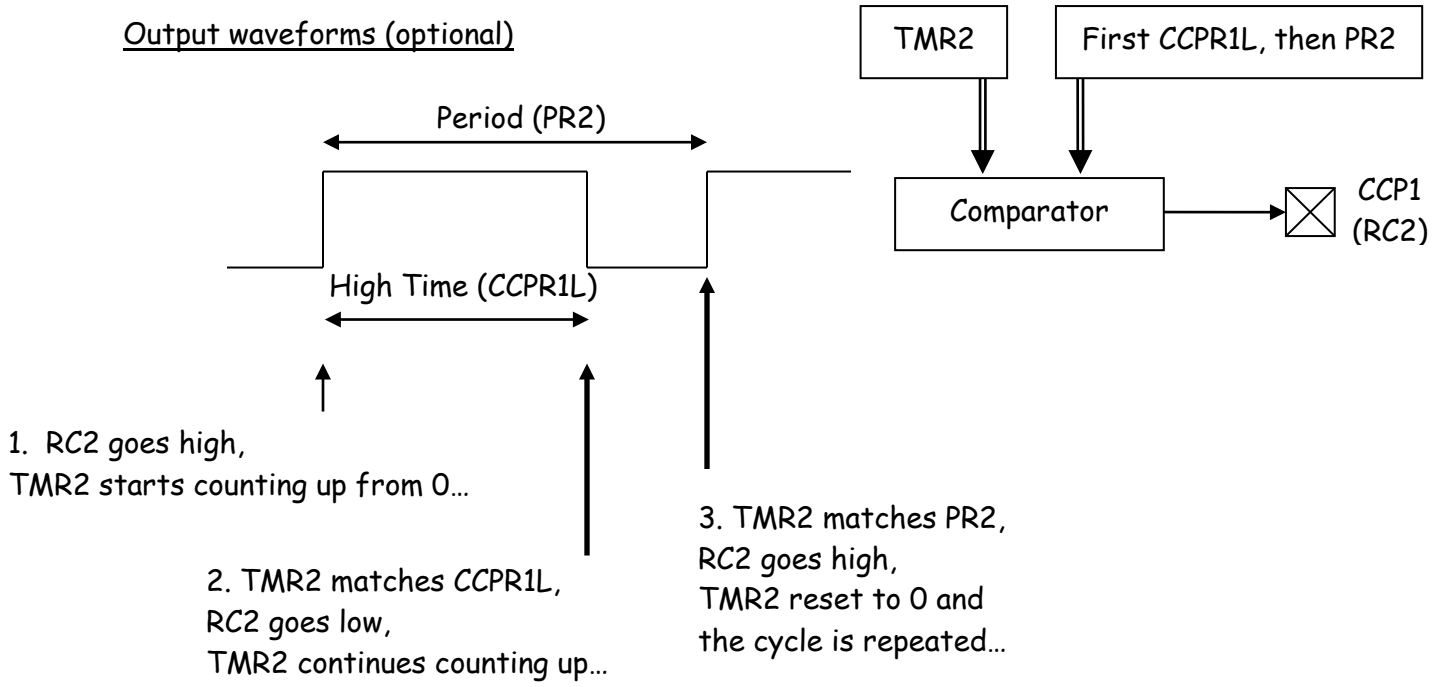


Figure 6.21 – Sequence of events to produce PWM output

Chapter 7 – PIC18F4550's Interrupt

Chapter Overview

7.1 Introduction to interrupt
7.2 PIC18F4550's Timer0 interrupt
7.3 PIC18F4550's INT0 external hardware interrupt
7.4 PIC18F4550's other interrupt features

7.1 Introduction to interrupt

- According to "wikipedia", an interrupt is a signal indicating the need for attention. This signal halts a micro-controller from its present task, so that something else can happen.

- Many **peripherals** e.g. I/O port, ADC, timer, serial port can be **active** simultaneously in a micro-controller application.

- The micro-controller needs to **check** for various **events** (e.g. a change in the voltage at an input pin, the completion of an A to D conversion, timer overflow, a serial byte received) and **responds** appropriately.

- For instance, if a user presses a button (causing a change in the voltage at an input pin), the micro-controller may need to turn on a light. Or, when the A to D conversion (of a temperature input, say) is completed, the micro-controller may need to display the result on an LCD display. Or, when the timer overflows, it is time to switch off the heater.

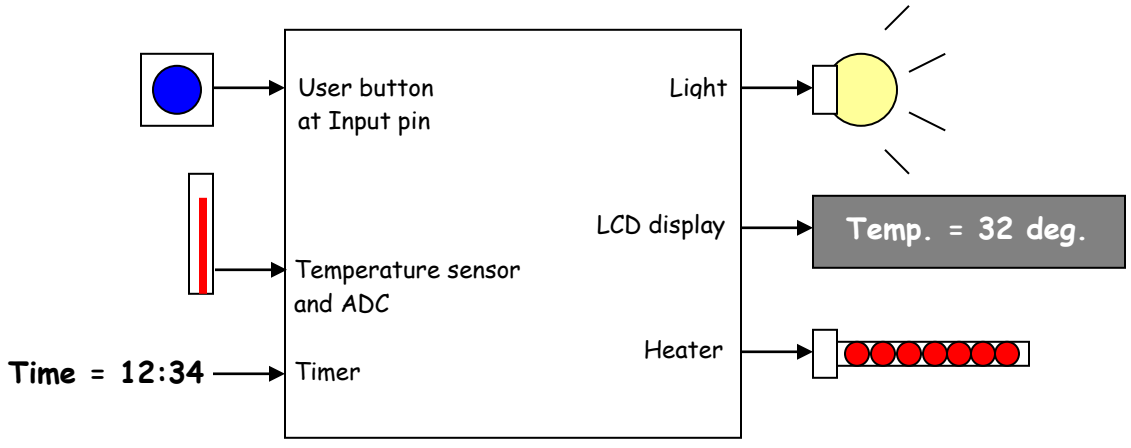


Figure 7.1 – Many active peripherals, a busy "automation" micro-controller

- The micro-controller can respond to "events" by **2 methods**: polling or interrupt.

Polling

- The code below shows how **polling** is done, for 2 events: 1.) RB0 changes from 0 (low) to 1 (high) and 2.) Timer0 overflows:

```
while (1)
{
        if (PORTBbits.RB0 == 1) …. // do something
        if (INTCONbits.TMR0IF == 1) … // do something else
        … // do other stuff, as micro-controller usually have a handful…
}
```

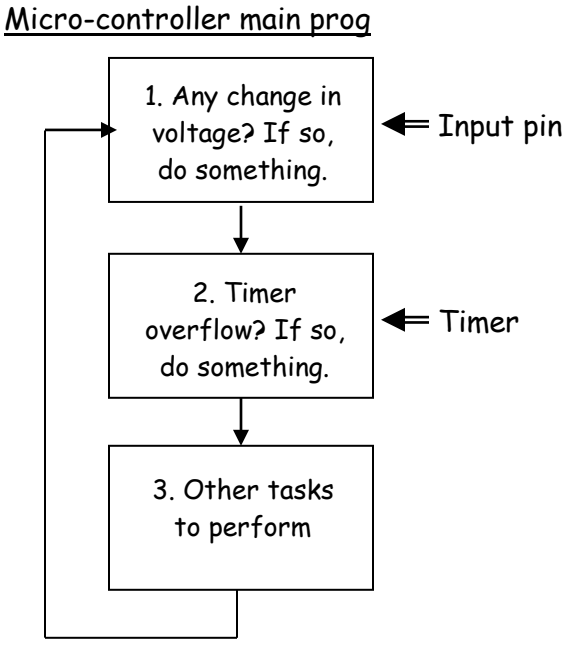


Figure 7.2 – Polling method to respond to event

- As you can see, polling means "checking" in a **round robin** fashion and **no priority** can be assigned to the more important event i.e. if the micro-controller is busy responding to a change in voltage at a port pin, it cannot do anything else even if the "more urgent" Timer0 overflow occurs during this time.

- Sometimes, an event hardly occurs and the micro-controller **wastes time** polling for the event.

Interrupt

- In the **interrupt** method, whenever a peripheral needs something to be done, it **notifies** the micro-controller, which stops whatever it is doing and "serves" the peripheral. (The program associated with the interrupt is called the **interrupt service routine** or ISR). After that, the micro-controller goes back to continue what it was doing.

- The interrupt method does not tie down the micro-controller (as you will see later) and allows priority to be assigned to the various events.

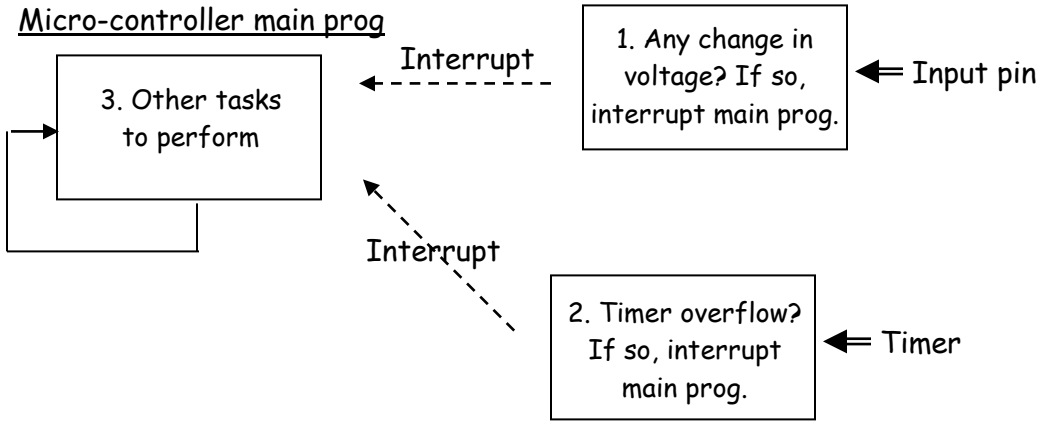


Figure 7.3 – Interrupt method to respond to event

- As an analogy, you could be reading newspaper the whole day (3 in above figure). But when the door bell rings (1) or there is a buzz tone on your hand phone (2), you are interrupted – you stop reading your newspaper and go to open the door or reply to an SMS. After that, you continue reading your newspaper where you left off.

- Fill in the table below to show the differences between the 2 methods by which a micro-controller can respond to events.

	Polling	Interrupt
Can priority be assigned to events?		
Does micro-controller have to monitor continuously?		

☐ We will learn only the basics of interrupt in this chapter, focusing on **Timer0 interrupt** and **INT0 external hardware interrupt**.

☐ We will also learn how an ISR (interrupt service routine) can be written. An "interrupt service routine" is a program / function used by a micro-controller to "serve" a peripheral or an event that needs attention.

7.2 PIC18F4550's Timer0 interrupt

☐ In Chapter 6, we learn how to use Timer0 and Timer2 (briefly). Let's see how we can use **Timer0 with Interrupt** in this section.

☐ Recall how a **periodic event** can be scheduled (e.g. toggling an LED every 0.1 second)? The complete C program to toggle an LED every 0.1 second follows. Take a few minutes to understand this (again).

T0CON (Timer0 Control Register)

TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0

```
TRISDbits.TRISD0 = 0; // RD0 as output
T0CON = 0b10000100; // Timer0 on, 16-bit, Fosc/4, pre-scaler 32
while (1) // loop forever
{
  TMR0H = 0x6D; // load Timer0 for a 0.1 second delay
  TMR0L = 0x84;
  while (INTCONbits.TMR0IF == 0); // wait for Timer0 overflow
  // the previous 3 lines gives a 0.1 second delay
  INTCONbits.TMR0IF = 0; // clear the flag

  PORTDbits.RD0 = ~ PORTDbits.RD0; // toggle RD0
}
```

Figure 7.4 – C code to toggle LED every 0.1 second

☐ The line highlighted in bold is checking for "Timer0 overflow" i.e. **polling**.

☐ Let's see what changes we need to make, if we decide to use **interrupt** instead of polling.

- To **enable** the Timer0 interrupt, we must first set both the GIE (**Global** Interrupt Enable) and the TMR0IE (**Timer0** Interrupt Enable) bits in the INTCON register.

INTCON (Interrupt Control Register)

GIE		TMR0IE			TMR0IF		
1		1					

- You can consider GIE as the "main switch", and TMR0IE as the "switch for a particular light". Both must be switched on, before the light is turned on.

- The C-code to do this follows:

```
INTCONbits.GIE = 1; // Global Interrupt Enable
INTCONbits.TMR0IE = 1; // Timer0 Interrupt Enable
```

- That is the easy part. The difficult part is, we must **re-organise** the C-code for polling (on the previous page) to change it into the C-code for interrupt (on the next page). The explanation follows:

Explanatory Notes

1. The lines that need to be executed **once before** the while (1) loop still remain before the while (1) loop: set RD0 as output, turn on Timer0: 16 bits, Fosc/4, pre-scaler 32.

2. Timer0 interrupt is **enable**d by setting both GIE (Global Interrupt Enable) and TMR0IE (Timer0 Interrupt Enable).

3. The lines that need to be executed **when Timer0 overflows** have been moved from the while (1) loop to the "**interrupt service routine**" (called my_Timer0_isr or any other suitable name). Note however that the line that **polls** for "Timer0 overflow" i.e. *while (INTCONbits.TMR0IF == 0);* is **NO LONGER required**, as interrupt is used, instead of polling. When TMR0IF becomes 1, the micro-controller will be "notified".

NEW "interrupt" code for Timer0

```
// The compiler will place this at a predefined location: 0x0008 for "high
// priority interrupt" service routine. (Similarly, 0x0018 for "low priority
// interrupt" service routine and 0x0000 for "reset" service routine.)
void interrupt my_Timer0_isr (void)
{
  TMR0H = 0x6D;
  TMR0L = 0x84;
  INTCONbits.TMR0IF = 0;
  PORTDbits.RD0 = ~PORTDbits.RD0;
}

void main (void)
{
  TRISDbits.TRISD0 = 0;
  T0CON = 0b10000100;
  INTCONbits.GIE = 1;
  INTCONbits.TMR0IE = 1;
  while (1)
  {
    // micro-controller freed to do other things here
  }
}
```

5 2 4

ORIGINAL "polling" code for Timer0

```
..main…
TRISDbits.bTRISD0 = 0;
T0CON = 0b10000100;
while (1)
{
  TMR0H = 0x6D;
  TMR0L = 0x84;
  while (INTCONbits.TMR0IF == 0);
  INTCONbits.TMR0IF = 0;
  PORTDbits.RD0 = ~PORTDbits.RD0;
}
```

1 3

Figure 7.5 – Changing code from "polling" to "interrupt"

4. As a result of 3 above, the micro-controller is **not kept busy** (**checking** for TMR0IF to become 1 *). It is now free to do other things. Of course, what it does with the "free time" depends on the application.

   (*) Checking "Timer0 overflow" all the time is like checking if there is a visitor at the door constantly – you might as well just read your newspaper!

   Note that when Timer0 overflows, whatever need to be done will still be done – the micro-controller will not be spared this effort!

5. The codes that handle Reset, High Priority Interrupt, Low Priority Interrupt must start at specific memory locations (called "**interrupt vectors**"), as follows:

Interrupt	Location (hex)
Power-on Reset	0000
High Priority Interrupt	0008 ★
Low Priority Interrupt	0018 ★

If priority feature is not used, all interrupts are treated as **high priority** interrupts. This is what is happening here i.e. our Timer0 interrupt is "high priority", to be "served" at location 0x0008.

A low priority interrupt service routine will appear like this:

```
void interrupt low_priority name_of_ISR (void) {
 // …
}
```

(#) Note that from 0x0008 to 0x0018, there are only a **few bytes** (16 bytes to be exact). If the code to service the high priority interrupt is longer than 16 bytes, it may be necessary to write the code elsewhere and a "GOTO" is placed at 0x0008 to "branch" there. We will not discuss this further.

(★) In the lab, a "boot-loader" is used in the PIC18F4550. This program downloads a user program from a PC via the USB port. The use of the boot-loader program affects the placement of the ISR. Again, we will not discuss this further.

☐ Which of the following conditions is/are necessary in order for Timer0 overflow to be serviced by an interrupt service routine?

☐ Global Interrupt Enable bit (in the INTCON register) must be set.

☐ Timer0 (Overflow) Interrupt Enable bit (in the INTCON register) must be set.

☐ Timer0 overflow has occurred i.e. Timer0 Overflow (Interrupt) Flag bit (in the INTCON register) is set.

- To recap, so far we have re-examined the C-code that toggles an LED every 0.1 second and re-organised the code so that we don't have to poll for "Timer0 overflow". Instead interrupt is used and when Timer0 overflows, interrupt occurs and whatever needs to be done will be carried out by the interrupt service routine.

- Changing from polling to interrupt in effect, pass the role of event checking from the C-code to the hardware.

- If you wish to know which lines of the C-code the micro-controller will execute, before and after Timer0 overflows, read the DIY material below:

Timer0 interrupt – execution sequence (DIY)

```
// At location 0x0008
void interrupt my_Timer0_isr (void)
{
  ……  ↓ E
}

void main (void)
{
  ……  // set up, including interrupt ↓ A
  while (1)
  {
    // assume the micro-controller does these lines all the time,
    //     when not interrupted by Timer0 overflow:
     // statement_1;        B      C      D      G
     // statement_2;                      F
     // statement_3;
     // statement_4;
  }
}
```

Figure 7.6 – Which lines get executed, before and after an interrupt?

- After setting up (A), including interrupt, the statements 1-4 get executed, over and over again (B, C…).
- If Timer0 overflows (i.e. TMR0IF=1 and interrupt occurs) when the micro-controller is executing statement_2 (D), it will **finish the line** of code before serving the Timer0 interrupt.
- To handle the interrupt, it will go to the interrupt service routine (my_Timer0_isr) at location 0x0008 and execute the code there (E).
- After executing the routine, the micro-controller will return to where it was i.e. to continue running from statement_3 (F, G…), until another interrupt.

7.3 PIC18F4550's INT0 external hardware interrupt

☐ The PIC18F4550 has 3 **external hardware interrupts**: INT0, INT1 and INT2 which use pins **RB0**, RB1 and RB2 respectively. We will discuss **INT0** (and INT1 and INT2 are similar in terms of operation).

☐ The INT0 interrupt responds to a change of voltage (from low to high – default) i.e. a transition at RB0.

☐ To **enable** the INT0 interrupt, we must first set both the GIE (**Global** Interrupt Enable) and the INT0IE (**INT0** Interrupt Enable) bits in the INTCON register.

INTCON (Interrupt Control Register)

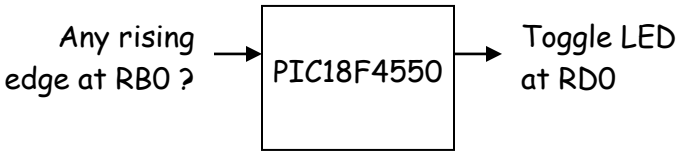
GIE			INT0IE			INT0IF	
1			1				

☐ Give the C-code to enable the INT0 interrupt. 

INTCONbits._____ = _____;
_____;

☐ When there is a rising edge at RB0, the INT0IF bit of the INTCON register will be set.

☐ The follow C-code has been written to **toggle** the LED at RD0 whenever there is a **rising edge** (i.e. a low to high transition) at RB0:




```
TRISBbits.TRISB0 = 1; // RB0 as input
TRISDbits.TRISD0 = 0; // RD0 as output

while (1) // loop forever
{
  while (PORTBbits.RB0 == 0); // wait for RB0 to become 1
  PORTDbits.RD0 = ~ PORTDbits.RD0; // toggle RD0
  while (PORTBbits.RB0 == 1); // wait for RB0 to become 0
}
```

Figure 7.7 – C code to toggle LED every time there is a rising edge at RB0

☐ Plot the waveform at RD0, giving the waveform at RB0. (DIY)

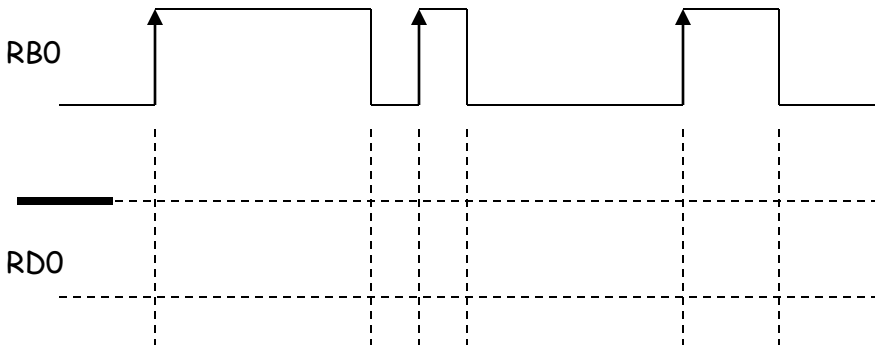


Figure 7.8 – Toggle RD0 every time there is a rising edge at RB0

☐ As before, we will **re-organise** the C-code for polling (on the previous page) to change it into the C-code for interrupt (on the next page).

INT0 (external hardware) interrupt code

```
// The compiler will place this "high priority interrupt" service routine
// at the predefined location 0x0008.

void interrupt my_INT0_isr (void)
{
  INTCONbits.INT0IF = 0; // clear flag
  PORTDbits.RD0 = ~PORTDbits.RD0;
}

void main (void)`
{
  TRISBbits.TRISB0 = 1;
  TRISDbits.TRISD0 = 0;
  INTCONbits.GIE = 1;
  INTCONbits.INT0IE = 1;
  while (1)
  {
    // micro-controller freed to do other things here
    Statement_1;
    Statement_2;
    Statement_3;
  }
}
```

Figure 7.9 – INT0 interrupt code

- Note that the **INT0IF flag** must be cleared in the interrupt service routine.

☐ Show the **sequence of execution** i.e. which C statement is executed first, which second etc, for the case where the first rising edge at RB0 occurs when the PIC is executing Statement_2 for the first time. 

Where	C statement	Sequence
my_INT0_isr	INTCONbits.INT0IF = 0;	
my_INT0_isr	PORTDbits.RD0 = ~PORTDbits.RD0;	
main	TRISBbits.TRISB0 = 1;	1
main	TRISDbits.TRISD0 = 0;	
main	INTCONbits.GIE = 1;	
main	INTCONbits.INT0IE = 1;	
main	Statement_1;	
main	Statement_2;	
main	Statement_3;	

Figure 7.10 – INT0 interrupt code execution sequence

☐ For INT0 (and also INT1 and INT2), it is also possible for interrupt to occur when there is a **falling edge** (i.e. a high to low transition) at RB0. This is done by clearing the INTEDG0 bit of the INTCON2 register.

INTCON2 (Interrupt Control Register 2)

	INTEDG0						
	0						

INTEDG0 = **0**: INT0 interrupt on **falling** edge at RB0
INTEDGO = 1: INT0 interrupt on rising edge at RB0 (power-on reset default)

7.4 PIC18F4550's other interrupt features

7.4.1 Multiple interrupt sources

☐ How do you use both Timer0 and INT0 (external hardware) interrupts together in a program? The program outline below shows how:

```
// function prototypes
void my_Timer0_isr (void);
void my_INT0_isr (void);

// code at location 0x0008

void interrupt my_isr (void)
{
  if (TMR0IF == 1)
    my_Timer0_isr;
  if (INT0IF == 1)
    my_INT0_isr;
}

void my_Timer0_isr (void)
{
  // service Timer0 overflows
  INTCONbits.TMR0IF = 0; // clear flag
}
```

```
void my_INT0_isr (void)
{
  // service rising edge at RB0
  INTCONbits.INT0IF = 0; // clear flag
}

void main (void)`
{
  // set up
  // enable interrupts
  while (1)
  {
    // do other things here
  }
}
```

Figure 7.11 – Program (outline) to service multiple interrupt sources

7.4.2 Other interrupt sources

- PIC18F4550 have many **interrupt sources**

    1. Timer1/2/3 interrupts (beside Timer0)
    2. External hardware interrupts INT1/2 (beside INT0)
    3. Serial comm. receive/transmit interrupts
    4. PORTB change interrupt (any pin of PORTB changed, interrupt occurs)
    5. ADC interrupt
    6. CCP (compare/capture/PWM) interrupts

7.4.3 Interrupt priority

- By default, all interrupts are "high priority", to be served from 0x0008.

- It is also possible to make some interrupts "high priority" (to be served from 0x0008) and others "low priority" (to be served from 0x0018). This is done by setting the **IPEN** (Interrupt Priority ENable) bit in the RCON register.

- When interrupt priority is enabled, we must classify each interrupt source as high priority or low priority. This is done by putting 0 (for low priority) or 1 (for high priority) in the **IP** (interrupt priority) bit of each interrupt source.

- The IP bits for the different interrupt sources are spread across several registers – INTCON, INTCON2, INTCON3, IPR1 and IPR2. We will not go into the details of all these.

- A higher priority interrupt can interrupt a low priority interrupt but NOT vice-versa.

- So, if you are reading the newspaper (your while(1) loop…) and your hand phone buzzes (low priority interrupt), you stop reading the newspaper to reply an SMS (low priority interrupt service routine).

- While doing that, the door bell rings (high priority interrupt), you stop messaging to open the door for your mom (high priority interrupt service routine).

- After that, you can return to your message, followed by the newspaper.

Summary

- In the interest of time, this chapter only focuses on Timer0 interrupt and INT0 external hardware interrupt.
- We discuss the differences between polling and interrupt, how to enable interrupt, how to write interrupt code and the sequence of code execution when interrupt occurs.

Chapter 8 – PIC18F4550's Serial Port (a brief intro to USART)

Chapter Overview

8.1 Basics of serial communication
8.2 PIC18F4550 connection to RS232
8.3 The registers associated with serial port operations
8.4 C-code for serial communication

8.1 Basics of serial communication

8.1.1 Serial vs. parallel

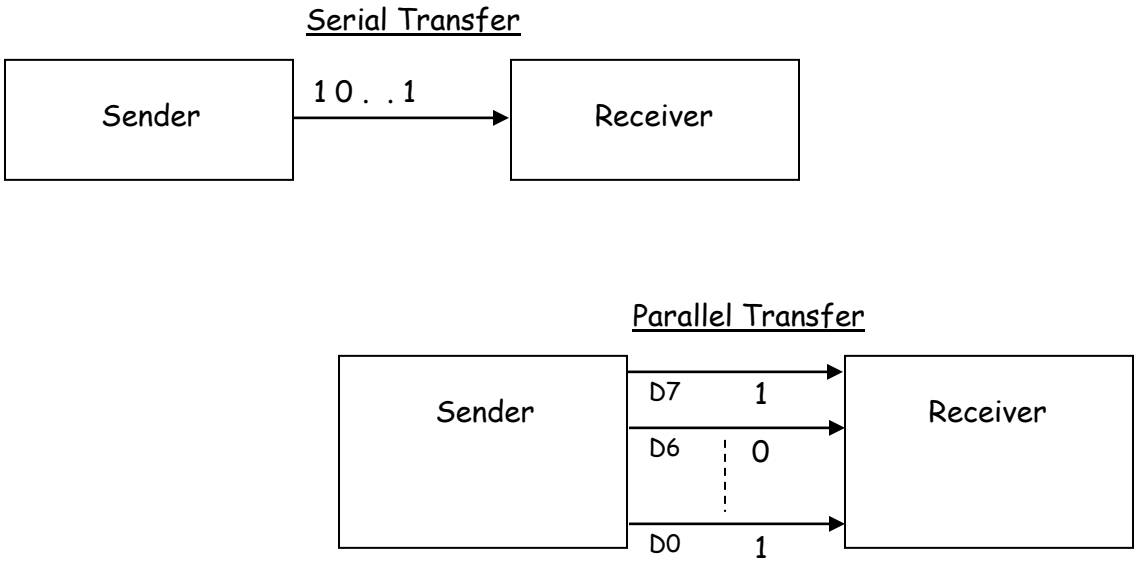


Figure 8.1 – Serial vs. parallel data transfer

☐ Serial data transfer is **slower**, but costs less as **less wire** is required.

☐ PIC18F4550 is an 8-bit micro-controller. The 8-bit data that it usually handles must be converted using a **parallel-in-serial-out** shift register, before serial transmission.

☐ What do you think the receiver must have to convert the serial data it receives into 8-bit data that it usually handles?

Answer: _____

8.1.2 Modulation / demodulation

- If the serial data is to be transferred over the **telephone line**, a **modem** is required.
- Modulation means converting the binary data (0's and 1's) into "audio tones" while demodulation means the reverse.

8.1.3 Synchronous vs. asynchronous

- The **synchronous method** transfers a block of data (multiple bytes) at a time whereas the **asynchronous method** transfers a single byte at a time.
- Special IC chips are made to make it easier to do serial data communication. They are called **UART** (universal asynchronous receiver-transmitter) and **USART** (universal synchronous-asynchronous receiver-transmitter).
- PIC18F4550 has a **built-in** USART, to be described in a later section.

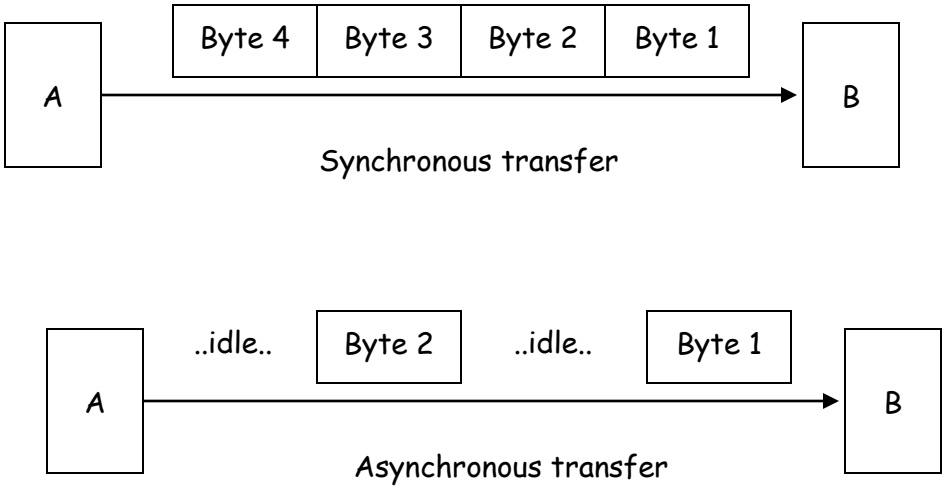


Figure 8.2 – Synchronous vs. asynchronous data transfer

8.1.4 Simplex, half-duplex and full-duplex

☐ The following shows how two devices can communicate with each other.

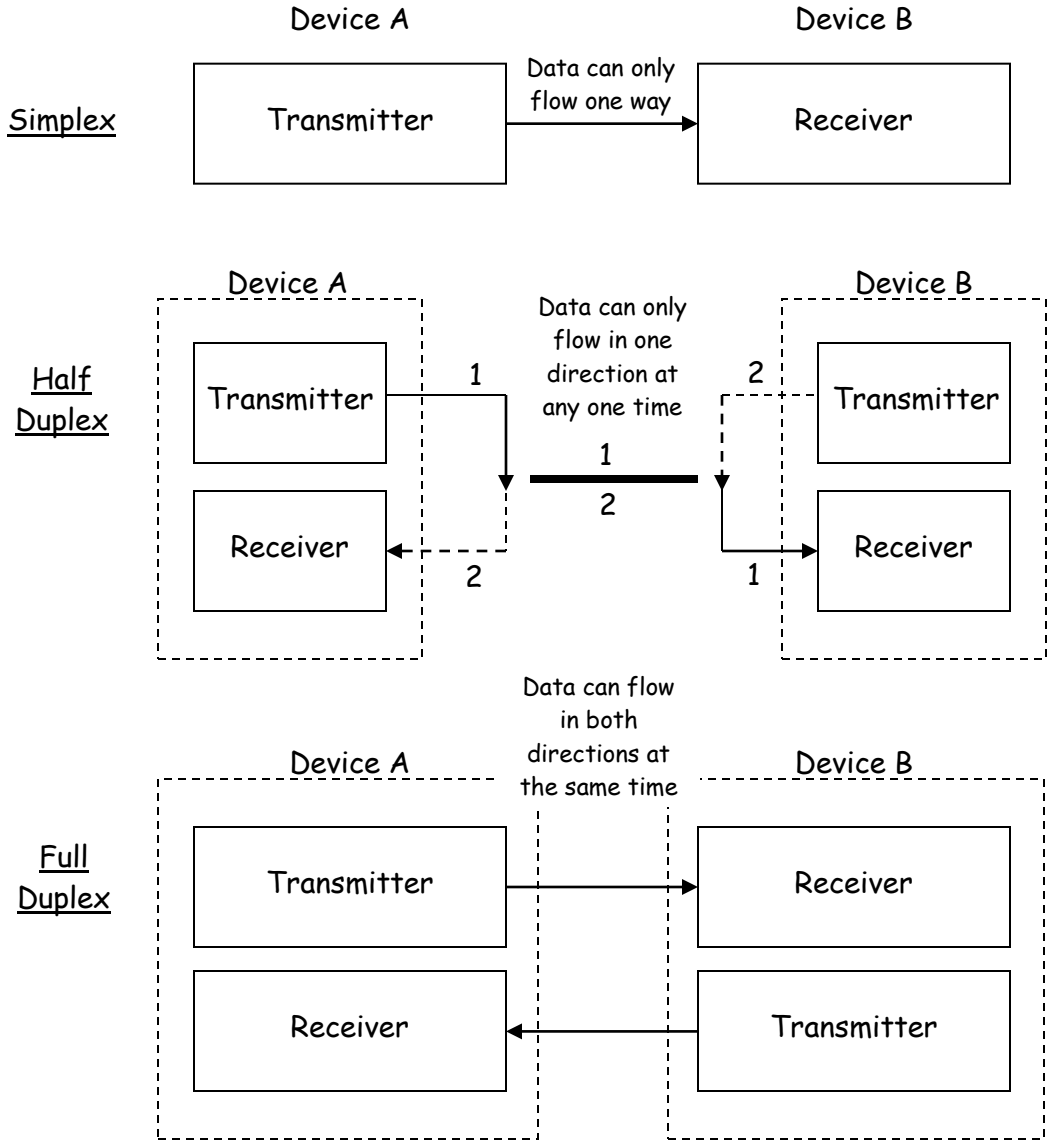


Figure 8.3 – Simplex, half- and full-duplex transfers

8.1.5 Serial communication protocol

☐ Two parties that communicate with one another must agree on a **protocol** i.e. a set of rules to make sense of the serial data – how the data is packed, how many bits constitute a character, and when the data begins and ends.

- For instance, the **ASCII** 'A' or 0x41 or 0b01000001 can be **framed** in the following way:

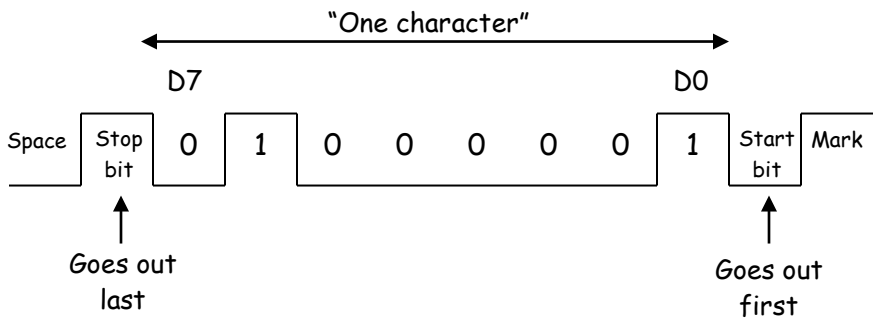


Figure 8.4 – Framing ASCII 'A' with one start bit and one stop bit

- The **start bit** is always one bit but the **stop bit** can be one or two bits. The start bit is always a '0' (low) and the stop bit(s) is '1' (high).

- Note that **LSB** (D0) is sent out first.

- When there is no transfer, the signal is '1' (high) which is referred to as **mark**.

- The data can be 7 bits wide or 8 bits wide (as in the above example).

- In some system, the **parity bit** of the character byte is included in the data frame (before the stop bit) to maintain data integrity. The parity bit is odd or even. In the case of **odd parity** bit, the number of 1's in the data bit, including the parity bit, is odd.

- If **even parity** is used, what is the parity bit for the ASCII 'A' i.e. 0b01000001?

  Answer: ____

- The rate of data transfer in serial data communication is stated in **bps** (bits per second). Sometimes, this is referred to as the **baud rate**.

- We will later learn how to specify the data transfer rate, how many data bit, whether to use odd/even/no parity and how many stop bit when serial communication is used in PIC18F4550.

8.1.6 RS232 interfacing standards

- **RS232** is the most widely used serial I/O **interfacing** standard, which allows PCs and numerous types of equipment made by different manufacturers to be connected to one another.

- In RS232, a '1' is represented by -3 to -25 V while a '0' is represented by +3 to +25 V.

- To connect any RS232 equipment to a micro-controller that produces TTL voltages (0 V for '0' and 5 V for '1'), a **voltage converters** such as **MAX232 or MAX233** (a kind of line driver) can be used.

- The **connector** used for the serial data cable can be male / female, 9-pin (so called DB9) or 25 pins (DB25).

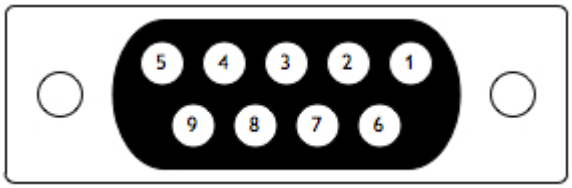


Figure 8.5 – DB9 connector

- The simplest connection between a PC and a micro-controller requires a minimum of 3 pins: **Tx** (Transmit), **Rx** (Receive) and ground, as shown below. Ensure that the Tx of one equipment goes to the Rx of the other equipment. Sometimes, other pins e.g. CTS (Clear To Send) are also used for "hand-shaking".

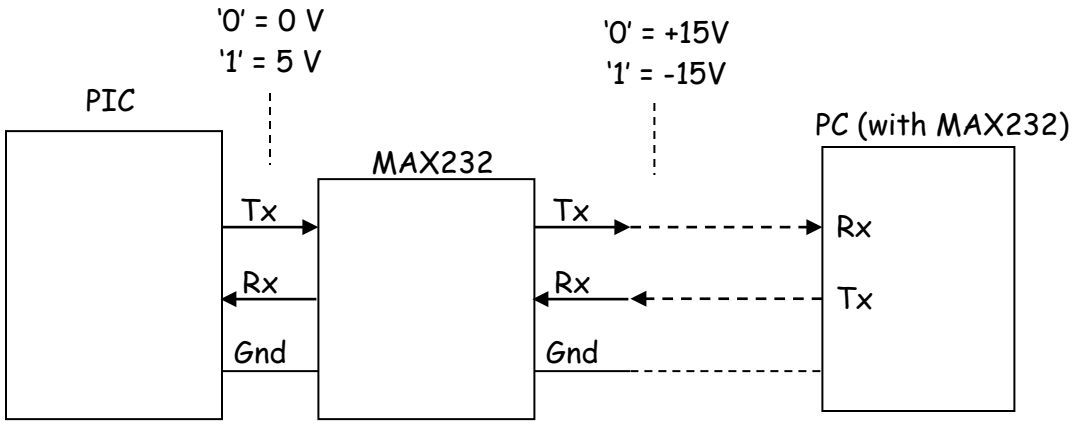


Figure 8.6 – Micro-controller serial port to PC COM port

- A PIC communicating with another PIC may do away with the MAX232, so that they communicate using TTL voltages. Still, Tx and Rx must be interchanged.

- Nowadays, ***COM*** ports (RS232 ports on a PC) are disappearing and replaced by **USB** ports. A "COM-to-USB converter" allows PC with only USB port to control devices with only RS232 interface e.g. a thermal label printer.

8.2 PIC18F4550 connection to RS232

- As can be seen below, **Tx** (transmit) shares pin 25 with **RC6** etc while **Rx** (receive) shares pin 26 with **RC7** etc.

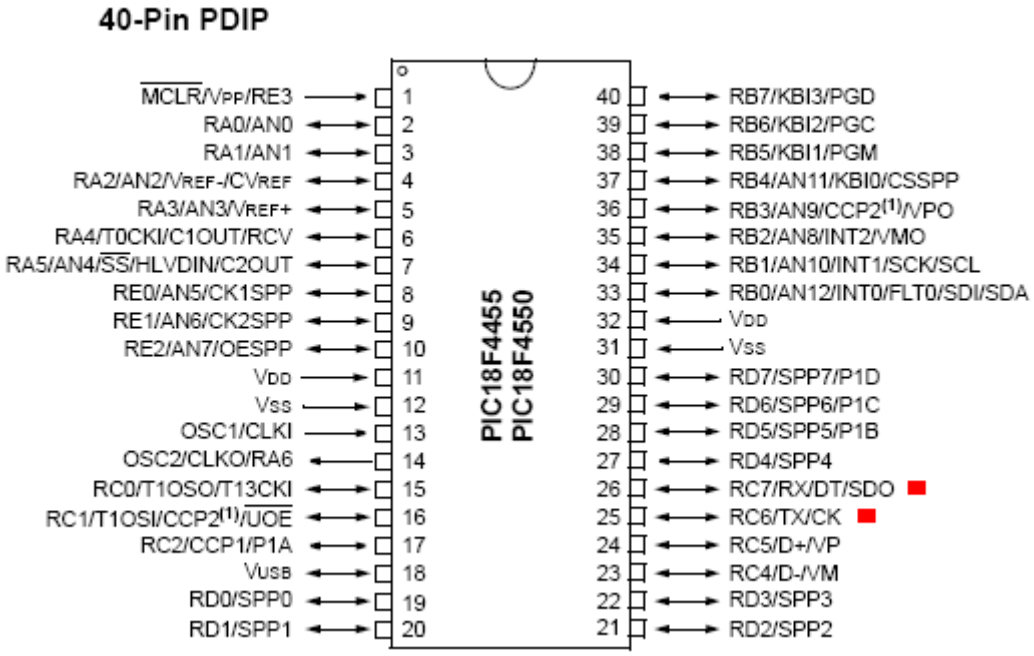


Figure 8.7 – Tx and Rx pins on PIC18F4550

- The following diagram shows how a PIC18F4550 can be connected to a **MAX 233** (voltage converter) and then to a **DB9** connector.

- One advantage of using MAX 233 (or MAX 232) is that only a **5 V** supply is needed although it converts TTL voltages to higher RS232 voltages.

- The voltage converter has **2 sets** of line drivers, but only one set is used below.

- MAX232 requires 4 **capacitors** while the more expensive MAX233 does not require any capacitor (so save PCB space!). However, the two are **not** pin-to-pin **compatible**.

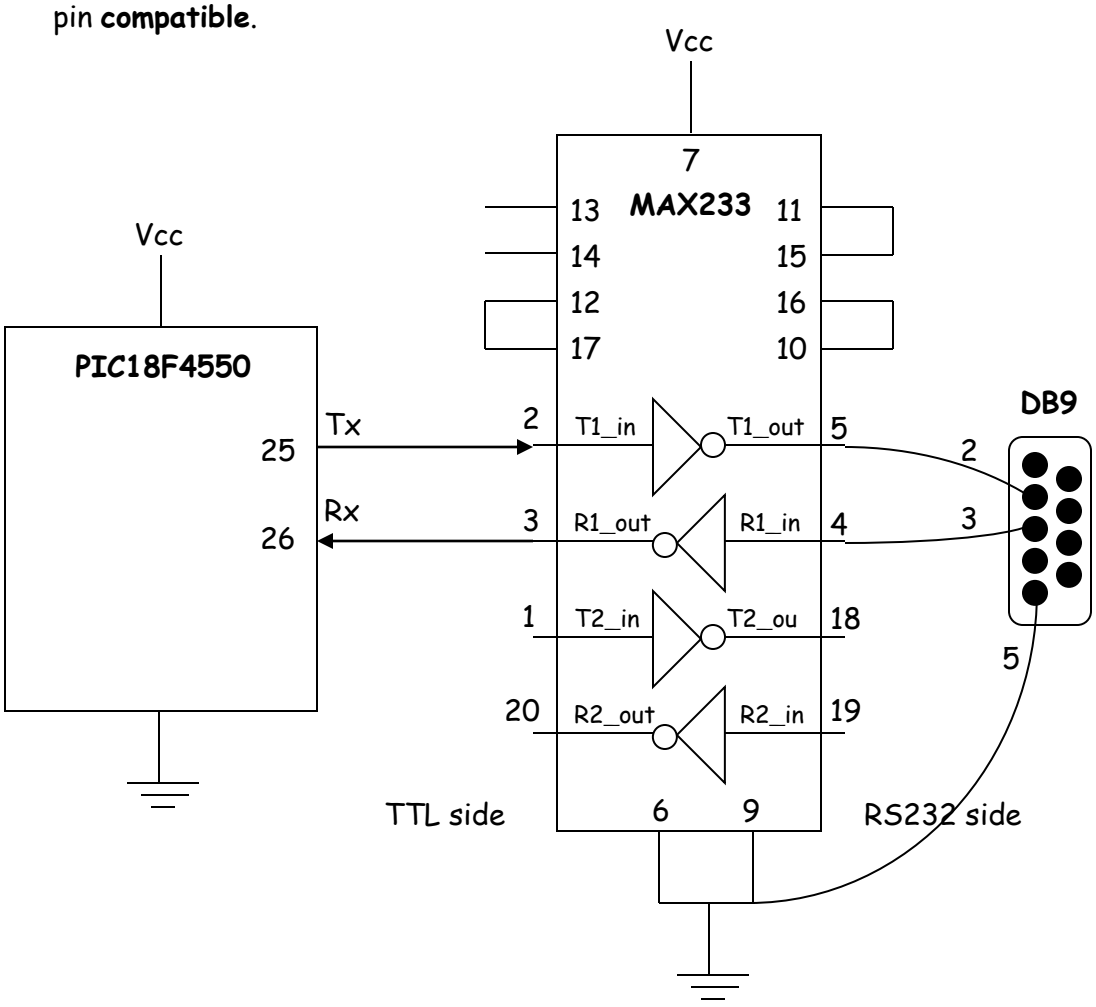


Figure 8.8 – Connecting PIC18F4550 to MAX233 and then to DB9

8.3 The registers associated with serial port operations

- PIC18F4550 serial port operations require setting up (or writing to) / checking (or reading from) a number of **registers**. Let's start with "SPBRG".

- Below, the registers will be described in just enough details to get things going. You can always read up the PIC18F4550 data-sheet or a good book, or do a Google-search, if you are working on a PIC18 serial comm. project and get stuck.

8.3.1 SPBRG – setting up the baud rate

- You can check whether your PIC-based system can send or receive data through its serial port by connecting it (see Figure 8.8 / 8.6) to the COM **port** of an **IBM PC/compatible** and running the **Hyper-Terminal** software on the PC.

- Some of the common **baud rates** supported by PC Hyper-Terminal are:

1,200	19,200
2,400	38,400
4,800	57,600
9,600	115,200

- The PIC18 transfers and receives data serially at many different baud rates.

- The baud rate in the PIC18 is programmable with the help of the 8-bit register called **SPBRG**.

- For a given **crystal frequency**, the value loaded into the SPBRG decides the baud rate. The relation is given by the formula:

$$\textbf{Desired Baud Rate} = \textbf{Fosc} / [\ 64\ (\ X + 1\ )\ ]$$

  Where X is the value loaded into the SPBRG register.

- Assuming Fosc = 48 MHz and desired baud rate = 9,600

$$9{,}600 = 48 \times 10^6 / [\ 64\ (\ X + 1\ )\ ]$$

So, X = 77.125 = 77 (**nearest integer**)

☐ The **C-code** is simply SPBRG = 77;

☐ The **error** introduced by the rounding can be determined as follows.

With X = 77 (instead of 77.125),

baud rate = 48×10^6 / [64 (77 + 1)]

= 9615.385 (= 0.16% error - insignificant)

☐ Determine the number to put into the SPBRG register for a baud rate of 9,600 assuming Fosc = 8 MHz. 

Answer: _____

8.3.2 TXREG – depositing data to be transmitted

☐ For a byte of data to be **transmitted** via the **TX pin**, it must be written to the **TXREG** register – an 8-bit register.

☐ The **C-code** is simply TXREG = byte_out;

☐ When this happens, the byte is fetched into the Transmit Shift Register (**TSR** - not accessible by the programmer) which **frames** it with the start and stop bits and then transfers the 10-bit data serially out via the TX pin.

8.3.3 RCREG – retrieving data received

☐ When the data bits are **received** serially via the **RX pin**, the PIC "**de-frames**" them by eliminating the stop and start bits, making a byte out of the data received, and then placing it in the **RCREG** register.

☐ The **C-code** to read the data received is byte_in = RCREG;

8.3.4 TXSTA – transmit status and control

☐ This is an 8-bit register used to select the **synchronous/asynchronous** mode, **8-bit/9-bit** data **transmission**, **high/low baud** rate, enable/disable **transmit**, among other things.

TXSTA: TRANSMIT STATUS AND CONTROL REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-1	R/W-0
CSRC	TX9	TXEN(1)	SYNC	SENDB	BRGH	TRMT	TX9D
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 7 **CSRC:** Clock Source Select bit
Asynchronous mode:
Don't care.
Synchronous mode:
1 = Master mode (clock generated internally from BRG)
0 = Slave mode (clock from external source)

bit 6 **TX9:** 9-Bit Transmit Enable bit
1 = Selects 9-bit transmission
0 = Selects 8-bit transmission

bit 5 **TXEN:** Transmit Enable bit(1)
1 = Transmit enabled
0 = Transmit disabled

bit 4 **SYNC:** EUSART Mode Select bit
1 = Synchronous mode
0 = Asynchronous mode

bit 3 **SENDB:** Send Break Character bit
Asynchronous mode:
1 = Send Sync Break on next transmission (cleared by hardware upon completion)
0 = Sync Break transmission completed
Synchronous mode:
Don't care.

bit 2 **BRGH:** High Baud Rate Select bit
Asynchronous mode:
1 = High speed
0 = Low speed
Synchronous mode:
Unused in this mode.

bit 1 **TRMT:** Transmit Shift Register Status bit
1 = TSR empty
0 = TSR full

bit 0 **TX9D:** 9th bit of Transmit Data
Can be address/data bit or a parity bit.

Figure 8.9 – TXSTA register

☐ To select the asynchronous mode, 8-bit data, low baud rate and to enable transmit, write 0b00100000 (or 0x20) to TXSTA.

☐ Bit 2 or **BRGH** can be used to select a **higher baud rate** (explained in the boxed below).

Higher Baud Rate

☐ There are 2 ways to **increase** the baud rate of data transfer in the PIC18F4550:

1. Use a **higher-frequency crystal**
2. **Set** bit 2 (or **BRGH**) of the TXSTA register and use a different formula to compute the value to be put into **SPBRG**

☐ The new formula with **BRGH = 1** is

Desired Baud Rate = Fosc / [16 (X + 1)]

☐ The previous formula with **default BRGH** value of **0** is

Desired Baud Rate = Fosc / [64 (X + 1)]

☐ How many times has the baud rate been increased, by setting BRGH = 1 (and without changing the value deposited into the SPBRG register)?

Answer: _____

8.3.5 RCSTA – receive status and control

☐ This is an 8-bit register used to enable/disable the **serial port**, select **8-bit/9-bit** data **reception**, enable/disable **receiver**, among other things.

RCSTA: RECEIVE STATUS AND CONTROL REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-0	R-0	R-x
SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 7 **SPEN:** Serial Port Enable bit
1 = Serial port enabled (configures RX/DT and TX/CK pins as serial port pins)
0 = Serial port disabled (held in Reset)

bit 6 **RX9:** 9-Bit Receive Enable bit
1 = Selects 9-bit reception
0 = Selects 8-bit reception

bit 5 **SREN:** Single Receive Enable bit
Asynchronous mode:
Don't care.
Synchronous mode – Master:
1 = Enables single receive
0 = Disables single receive
This bit is cleared after reception is complete.
Synchronous mode – Slave:
Don't care.

bit 4 **CREN:** Continuous Receive Enable bit
Asynchronous mode:
1 = Enables receiver
0 = Disables receiver
Synchronous mode:
1 = Enables continuous receive until enable bit CREN is cleared (CREN overrides SREN)
0 = Disables continuous receive

bit 3 **ADDEN:** Address Detect Enable bit
Asynchronous mode 9-bit (RX9 = 1):
1 = Enables address detection, enables interrupt and loads the receive buffer when RSR<8> is set
0 = Disables address detection, all bytes are received and ninth bit can be used as parity bit
Asynchronous mode 9-bit (RX9 = 0):
Don't care.

bit 2 **FERR:** Framing Error bit
1 = Framing error (can be updated by reading RCREG register and receiving next valid byte)
0 = No framing error

bit 1 **OERR:** Overrun Error bit
1 = Overrun error (can be cleared by clearing bit CREN)
0 = No overrun error

bit 0 **RX9D:** 9th bit of Received Data
This can be address/data bit or a parity bit and must be calculated by user firmware.

Figure 8.10 – RCSTA register

☐ To enable the serial port, select 8-bit data, and enable receiver, write 0b10010000 (or 0x90) to RCSTA.

8.3.6 PIR1 – peripheral interrupt request (flag) register 1

☐ Two bits of the PIR1 register are used by USART - TXIF and RCIF.

☐ **TXIF** (transmit interrupt flag) == 1 indicates that the **previous byte** has been **transmitted** and a new byte can be written to the TXREG register.

☐ **RCIF** (receive interrupt flag) == 1 indicates that a **new byte** has been **received** and can be read from the RCREG register.

☐ Instead of **polling**, we can use **interrupt**. But this will not be discussed here.

PIR1: PERIPHERAL INTERRUPT REQUEST (FLAG) REGISTER 1

R/W-0	R/W-0	R-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0
SPPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 5 **RCIF:** EUSART Receive Interrupt Flag bit
1 = The EUSART receive buffer, RCREG, is full (cleared when RCREG is read)
0 = The EUSART receive buffer is empty

bit 4 **TXIF:** EUSART Transmit Interrupt Flag bit
1 = The EUSART transmit buffer, TXREG, is empty (cleared when TXREG is written)
0 = The EUSART transmit buffer is full

Figure 8.11 – PIR1 register

8.4 C-code for serial communication

8.4.1 Programming the PIC18 to transmit data serially

☐ The steps to transmit 8-bit data serially is as follows:

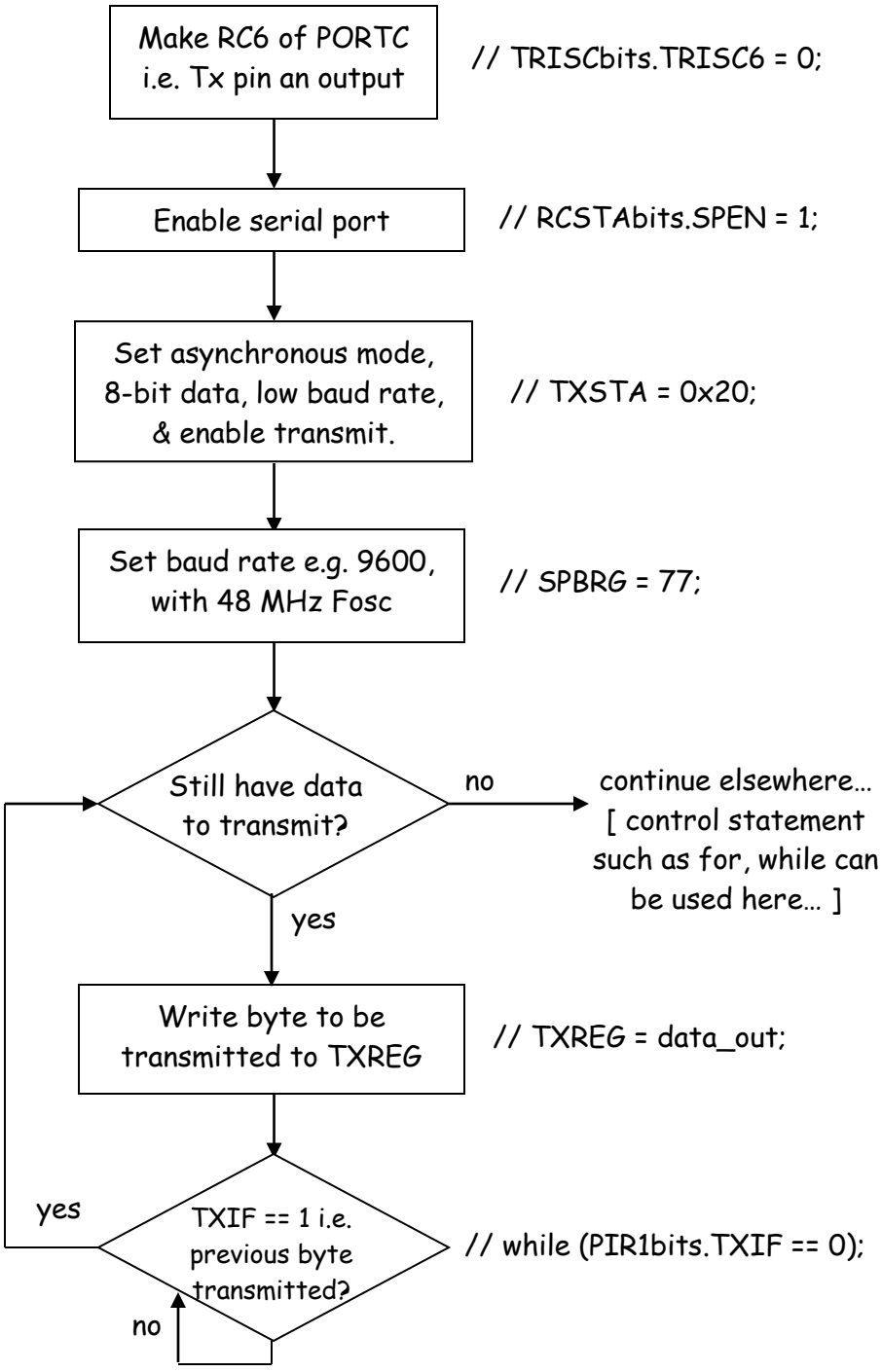


Figure 8.12 – Flowchart for serial transmission

8.4.2 Programming the PIC18 to receive data serially

☐ The steps to receive 8-bit data serially is as follows:

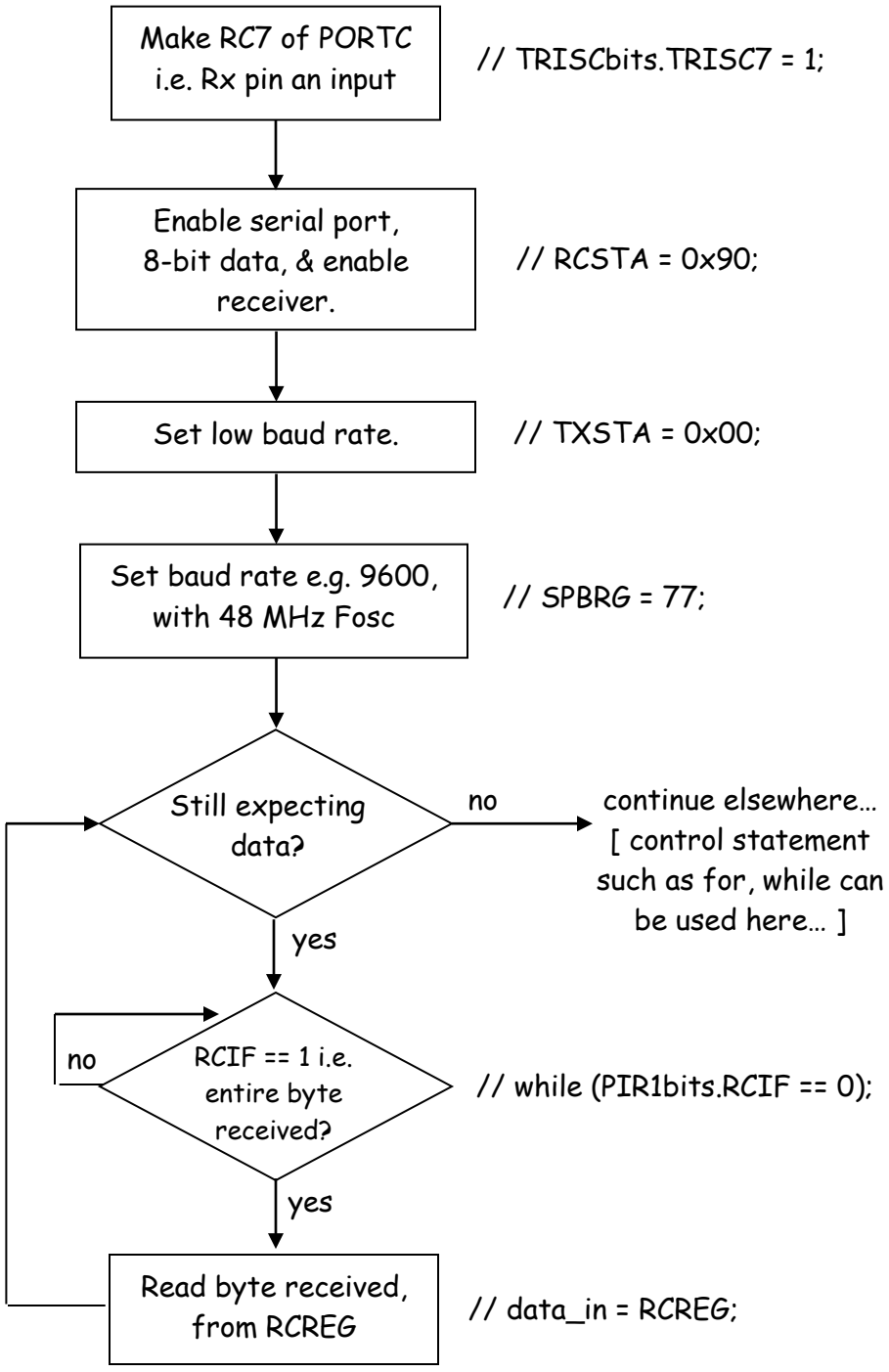


Figure 8.13 – Flowchart for serial reception

8.4.3 Programming the PIC18 to transmit & receive data serially (optional)

☐ The following program continuously transmits data from an array ("data_out") and receives data into another array ("data_in").

☐ Comment the program (using the comments below) to show that you have understood what each line does. 

A. Write one byte to be transmitted to TXREG
B. Check if byte received
C. Read byte received from RCREG
D. Make RC6 i.e. Tx of PORT C an output pin
E. Set baud rate = 9600
F. Enable serial port, 8-bit reception and enable receiver
G. Enable 8-bit transmission, select asynchronous mode & low baud rate
H. Make RC7 i.e. Rx of PORT C an input pin
I. Check if OK to transmit byte

```
TRISCbits.TRISC6 = 0;  // _____________________________
TRISCbits.TRISC7 = 1;  // _____________________________

RCSTA = 0x90;  // _____________________________
TXSTA = 0x20;  // _____________________________
SPBRG = 77;   // _____________________________

t = 0; // t is index to array of data to be transmitted
r = 0; // r is index to array for data received

while (1)
{
   if (PIR1bits.TXIF == 1)  // _____________________________
   {
      TXREG = data_out [t];  // _____________________________
      t++;
   }

    if (PIR1bits.RCIF == 1)  // _____________________________
    {
      data_in [r] = RCREG; // _____________________________
      r++;
   }

   // do other things here…
}
```

Lab 1 – Introduction to PIC18F4550 Board, MPLABX-IDE, C-compiler and USB downloader.

Objectives

- ☐ To illustrate the procedures to create a Microchip's PIC micro-controller project in MPLABX IDE, and to create, edit and compile a C program using XC8.
- ☐ To show the steps to setup the USB link with the PIC18F4550 micro-controller, and to download a program to the micro-controller and to execute it.

Introduction / Briefing

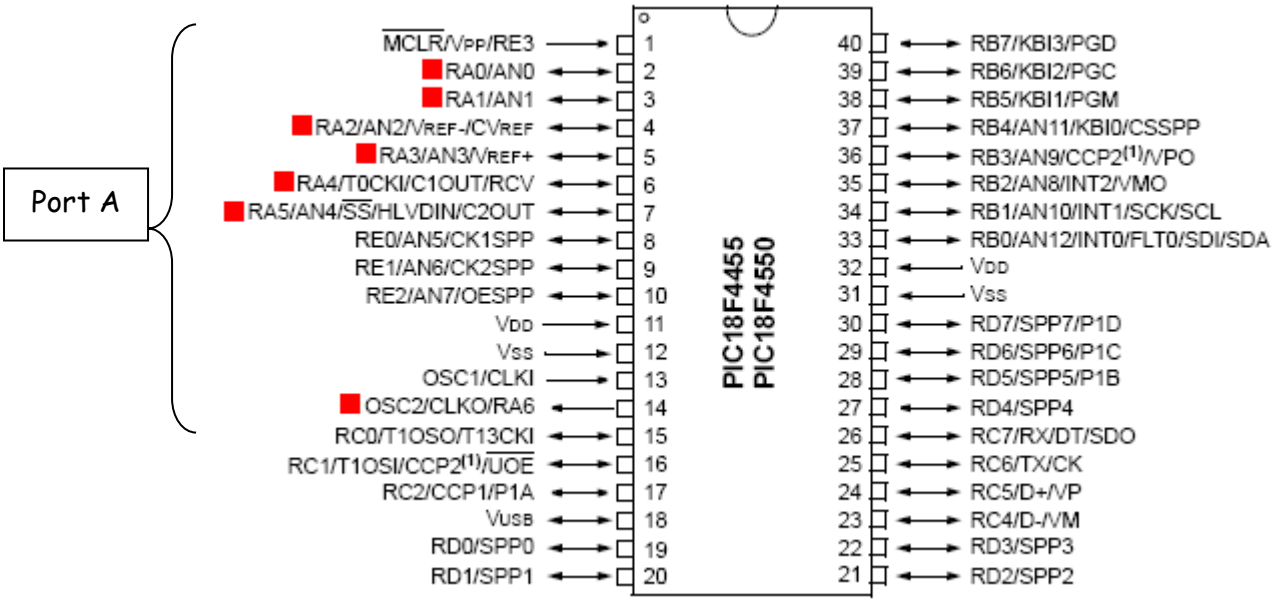
⇨
- ☐ At the beginning of each lab session, your lab lecturer will go through a short **briefing** before you begin the experiment.
- ☐ The discussion will help you in the MST, the lab test as well as the project. So, please pay attention and participate in the discussion.

⇨
- ☐ This lab sheet contains many **screen captures** to show you how to create a project, how to create, edit and compile a C program, and how to download a program to the micro-controller and run it. In subsequent labs, if you forget certain steps, you should refer to this lab sheet again.

⇨
- ☐ To do this lab, the **software tools** required must already be installed on the PC.

PIC18F4550 I/O ports

- ☐ You will learn more about the I/O ports in Chapter 3. The following is a brief summary.
- ☐ PIC18F4550 has five I/O ports: A to E. Many pins have multiple functions. For instance, pin 14 is RA6 (Port A Pin 6) and also OSC2 (oscillator input 2).



☐ The table below shows which pins can be used as general purpose I/O pins and whether they are, by default (i.e. after power on reset), analogue or digital, input or output.

Port	Available pins	Not available as general purpose I/O (- reasons)	After power on reset
A	RA6-0	RA6 (– oscillator)	RA5, 3-0: Analogue inputs (*). RA4: Digital input.
B	RB7-0	RB4 (– "Boot" button)	RB4-0: Digital / Analogue inputs (#). RB7-5: Digital inputs.
C	RC7-4, 2-0	RC5-4 (– USB connector)	RC7-4, 2-0: Digital inputs.
D	RD7-0		RD7-0: Digital inputs.
E	RE3-0	RE3 (– "Reset" button)	RE2-0: Analogue inputs (*). RE3: Digital input.

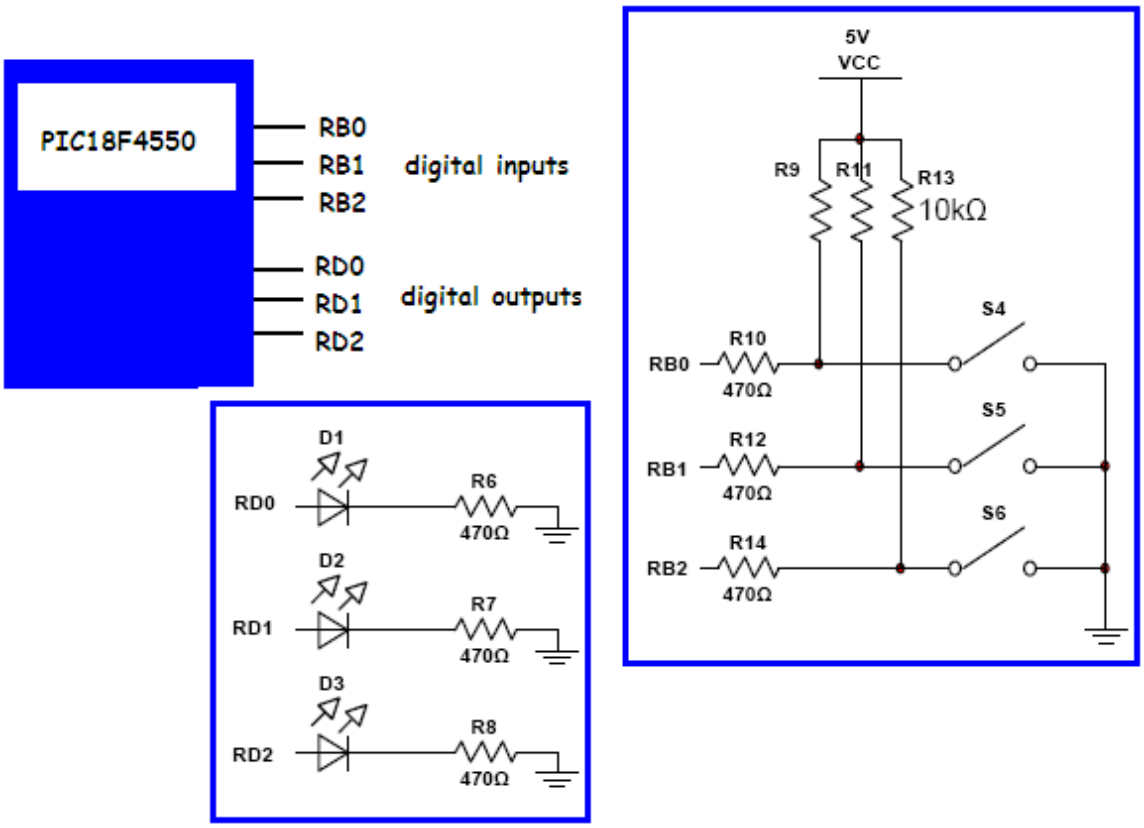
(*) ADC (Analogue to Digital Conversion) will be discussed in details in the future.
(#) For the PIC18 chips used in the labs, RB4-0 are Digital inputs after power on reset.

☐ This lab will only involve ports B and D.

Port configuration

☐ Do you know why the switches connected to RB0-2 are "active low"?

☐ Do you know why the LED's connected to RD0-2 are "active high"?



☐ To use port B to read the switch status (open or closed), port B must be configured as digital inputs.

☐ Referring to the table above, RB4-0 are digital inputs after reset.

☐ To use port D to control the LEDs (on or off), port D must be configured as digital outputs.

☐ But, RD7-0 are digital inputs after reset.

☐ The command below must be added to change them into digital outputs:

```
TRISD = 0b00000000;
```

☐ TRISD is the "data directional register" for Port D.

☐ By writing a 0 into a particular TRISD bit, the corresponding PORTD pin become an Output pin.

	7	6	5	4	3	2	1	0
TRISD	0	0	0	0	0	0	0	0

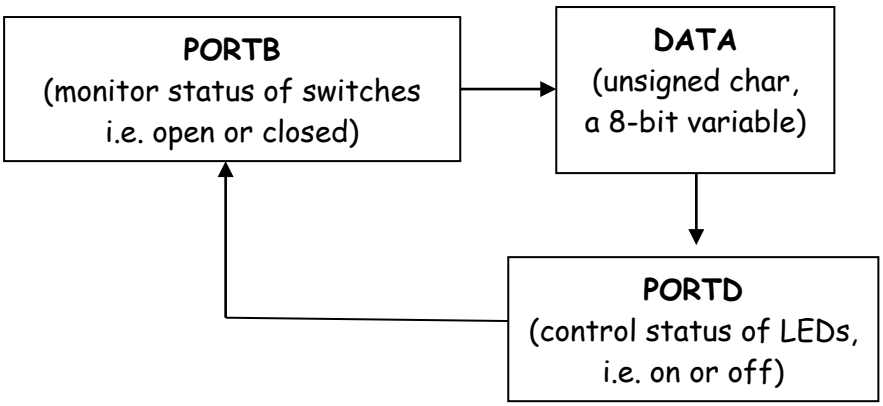
	RD7	RD6	RD5	RD4	RD3	RD2	RD1	RD0
PORTD	Output	Output	Output	Output	Output	Output	Output	Output

☐ Likewise, by writing a 1 into a particular TRISD bit, the corresponding PORTD pin can become an Input pin.

Looping forever

☐ After configuring Port B as digital input and Port D as digital output, the while (1) loop below will be executed over and over:

```
while (1)
 {
   data = PORTB;  // switch status is copied into a variable called DATA
   PORTD = data;  // and used to turn on/off the LEDs
 }
```



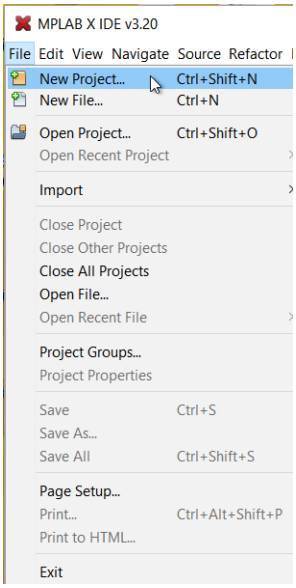
Activites:

1. Before the beginning of each lab lesson, double click on the MAPP icon on the desktop. This will delete the files modified by other students before you and replace them with fresh copies. The (.c & project) files used in all the experiments will be stored in a folder named ProjectX in the D:\ Drive.

2. Double click on the *MPLABX IDE* icon on the desktop to launch the software.

Creating a Microchip's PIC micro-controller project in MPLABX-IDE

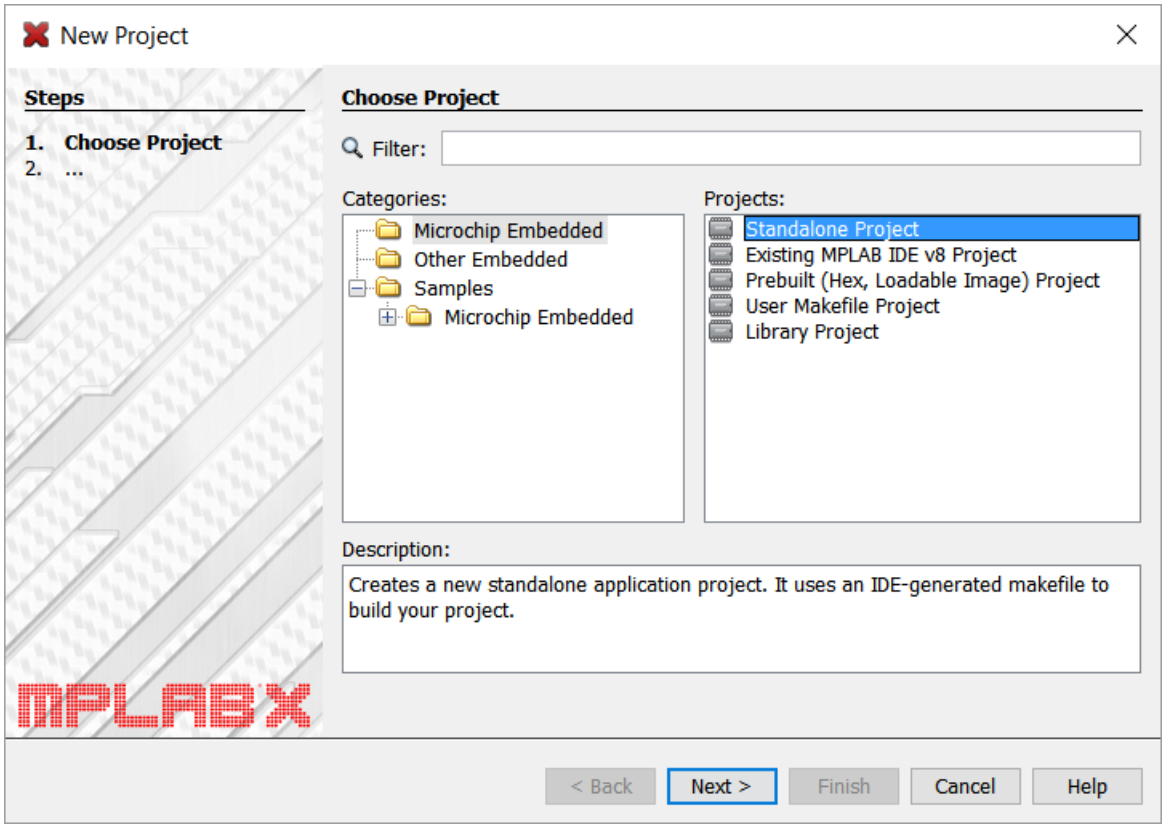
3. Click *File -> New Project …* to create a new project.



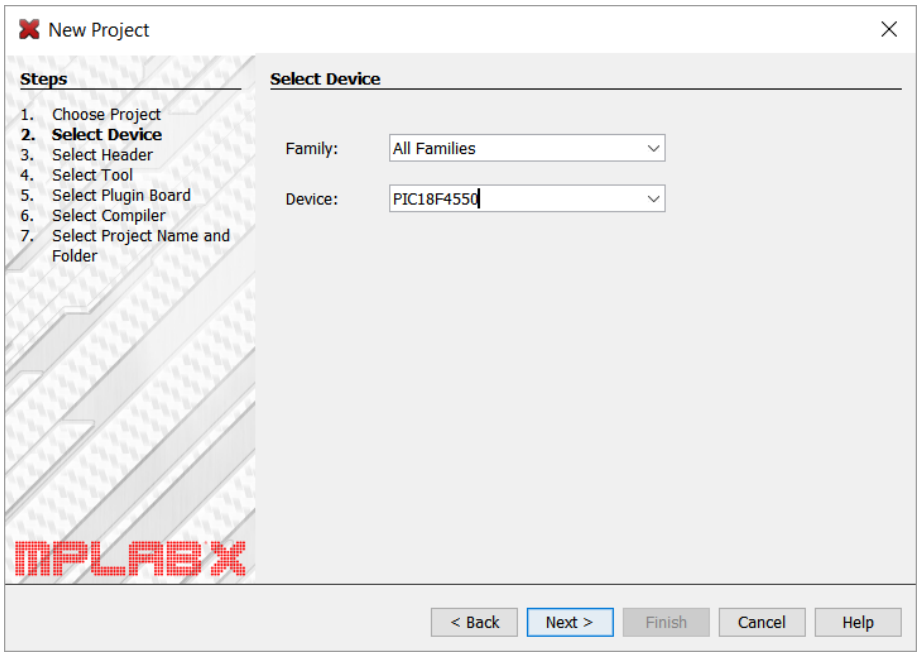
4. You will see the *New Project* window.

We will use the default setting *Microchip Embedded – Standalone Project.*

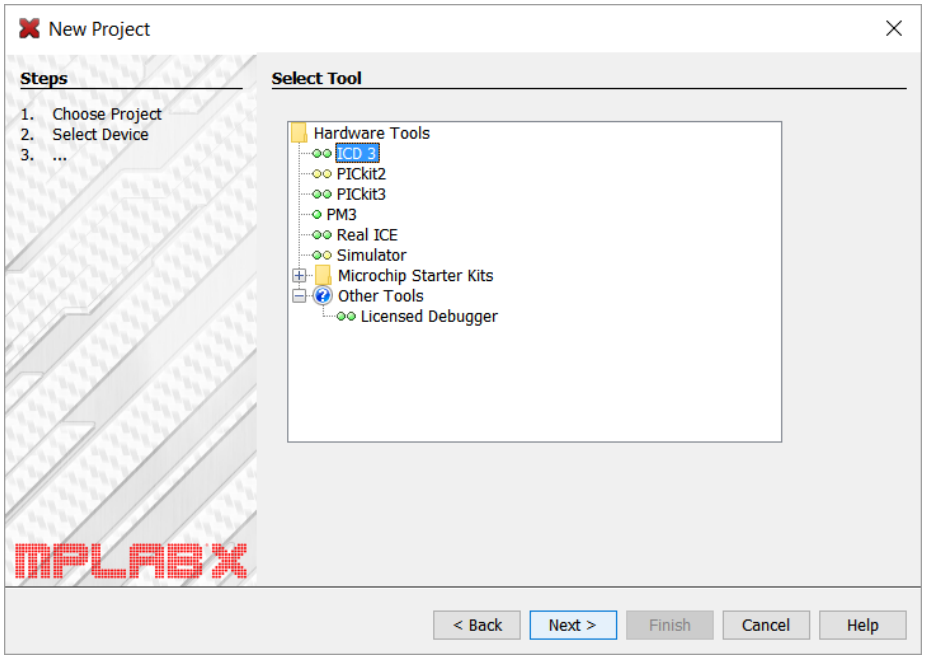
Click *Next.*



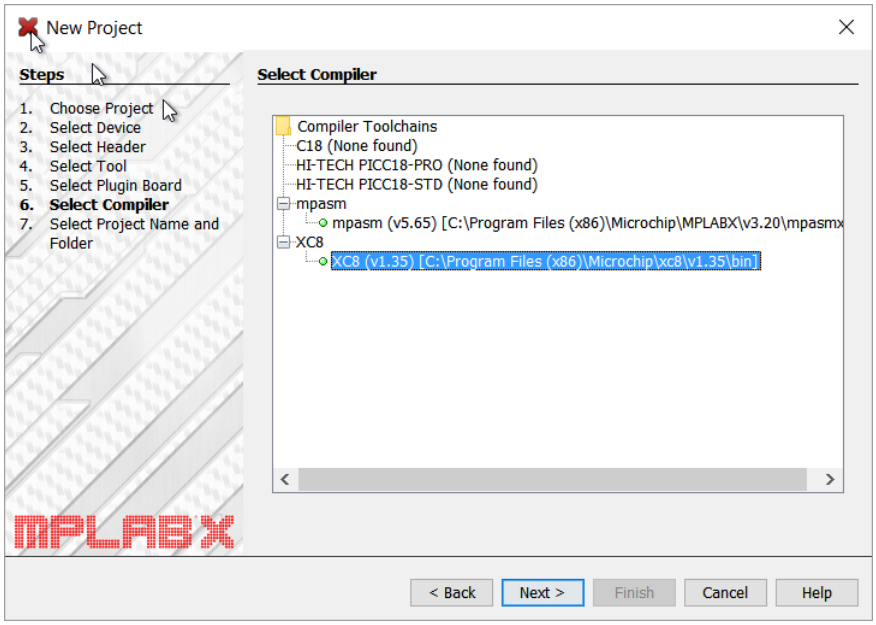
5. Select *PIC18F4550* as the *Device* (i.e. the microcontroller) to use for this project. [Hint: You can type the number 4550 to filter down the list.] Then, click *Next*.



6. We are not using any debugging tools, so just click *Next*.

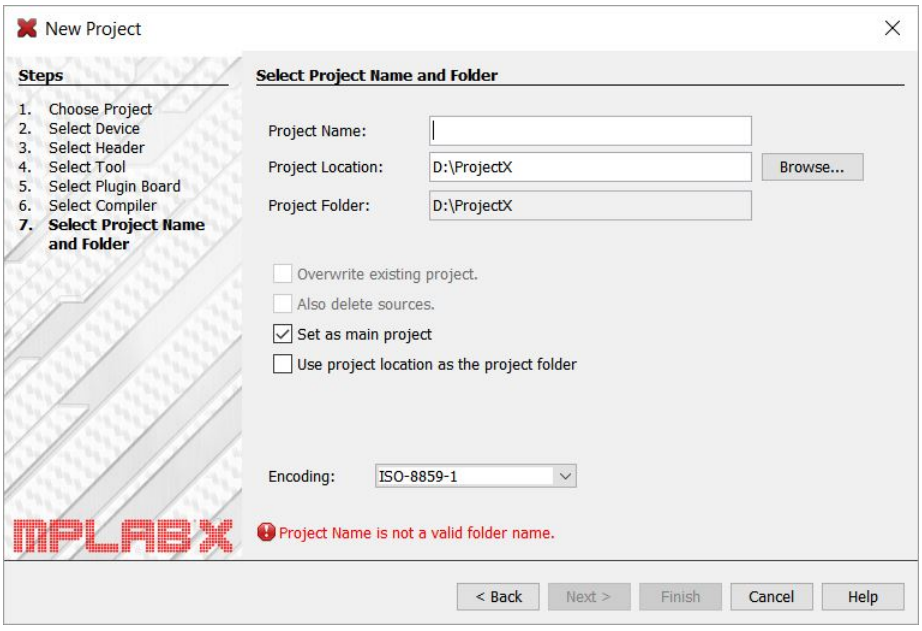


7. Select the *XC8* Compiler as follows and click *Next*.



8. Browse to the *Project Location* as follows and enter the *Project Name*:

Project Name : *Lab1*
Project Location : *D:\ProjectX*

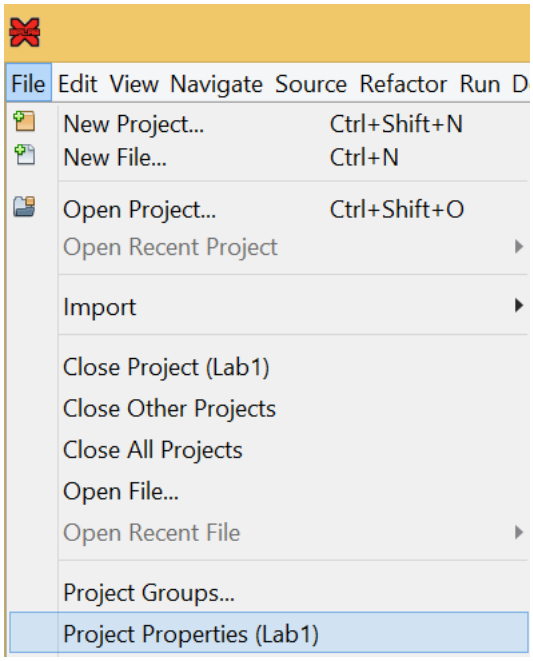


Then, click *Finish*.

9. Set the Codeoffset to 1000 as follows.

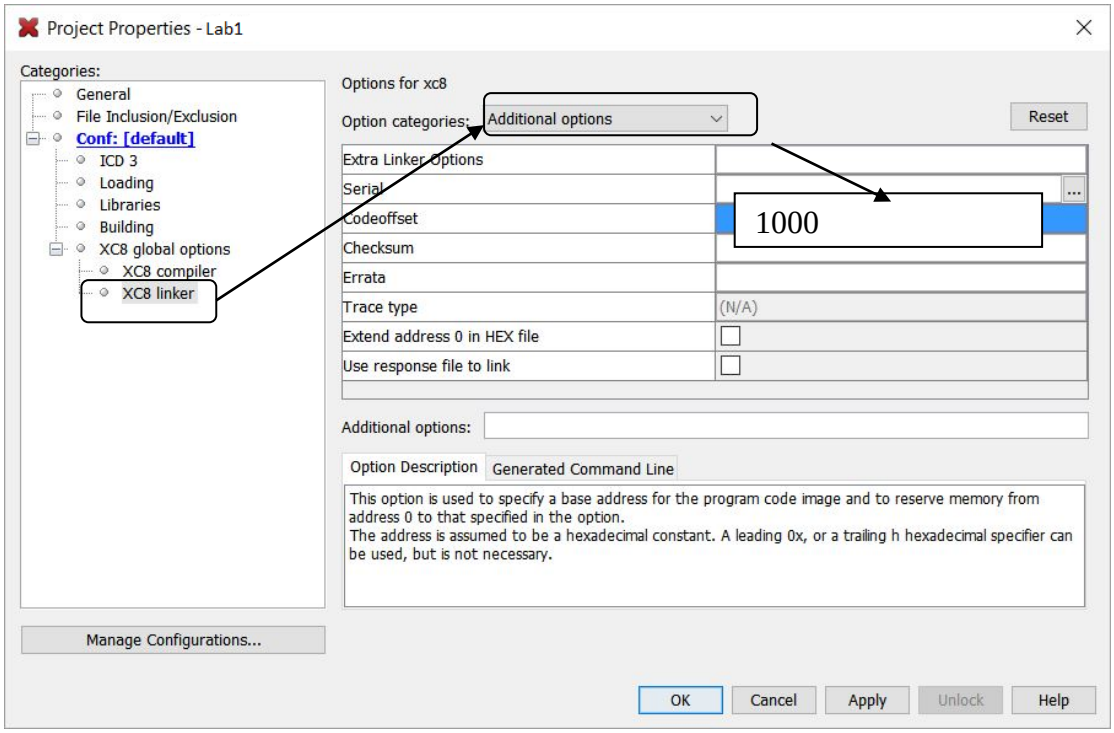
The microcontroller that we are using already has a program in the ROM called the bootloader. It is used for communicating with a program in the PC using the USB ports. The bootloader in the ROM occupies the locations from 0x0000 to 0x0FFF. The user program for the microcontroller must therefore starts from location 0x1000. To do this, we must set the Codeoffset to 1000.

Click *File -> Project Properties (Lab1)*:

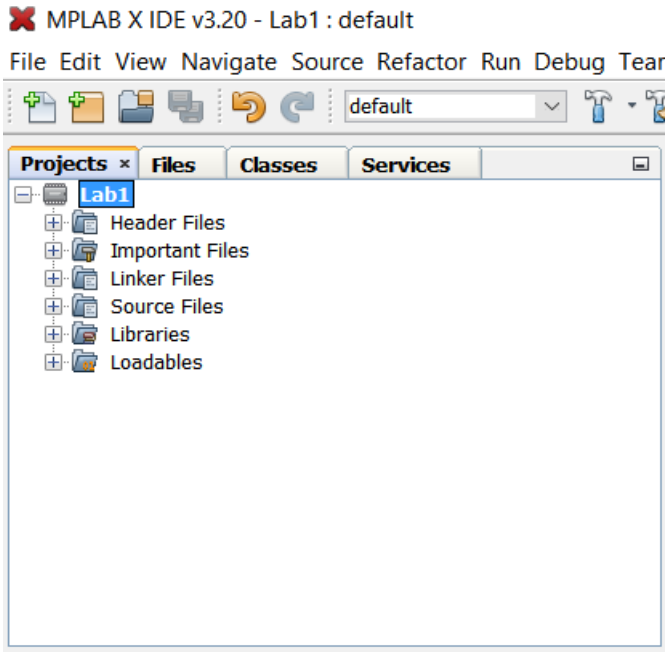


10. You will see the *Project Properties – Lab 1* window:

    Click *XC8 linker*. Then in *Option categories*, select *Additional options*. And enter 1000 for *Codeoffset* and then click *OK*.



11. You will see a summary of the project *Lab1* that you have just created.

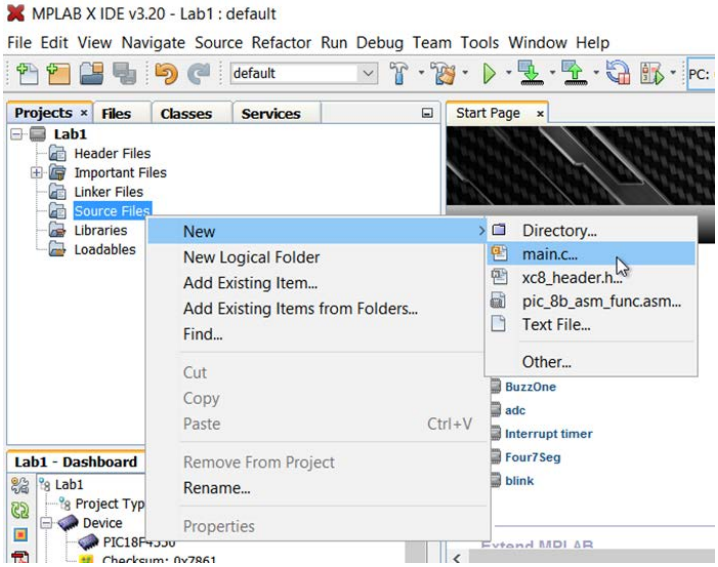


At this point, a new project has been created but there are no files.

Creating, editing and compiling a C-program using XC8

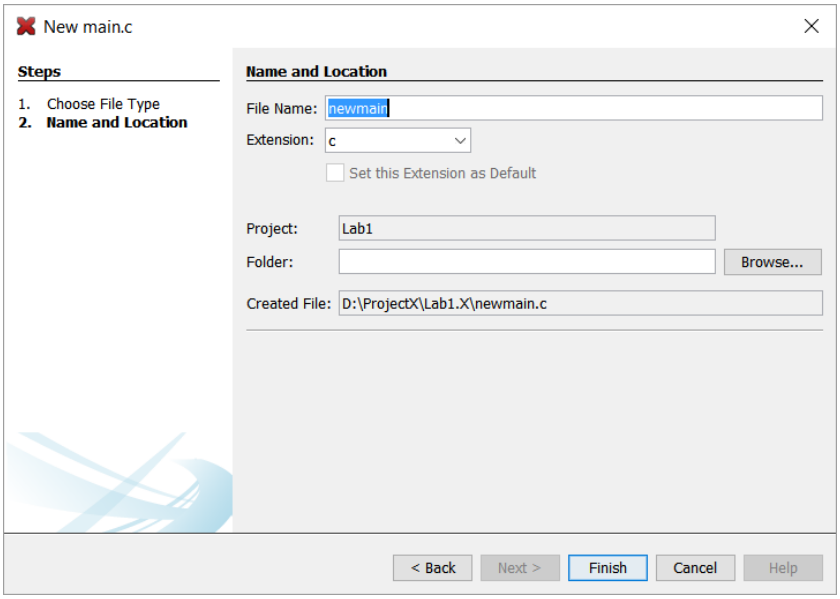
☐ In the next few steps, you will add a new C source file, and edit it. You will then compile the C program.

12. To add a new file, right click on *Source Files -> New -> main.c*

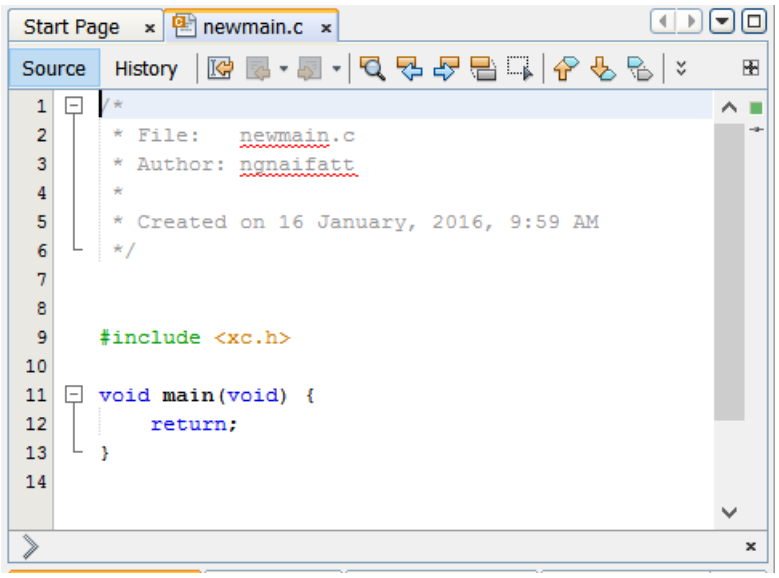


For this lab, we will use the default *File Name*: *newmain* and default *Extension*: *c*.

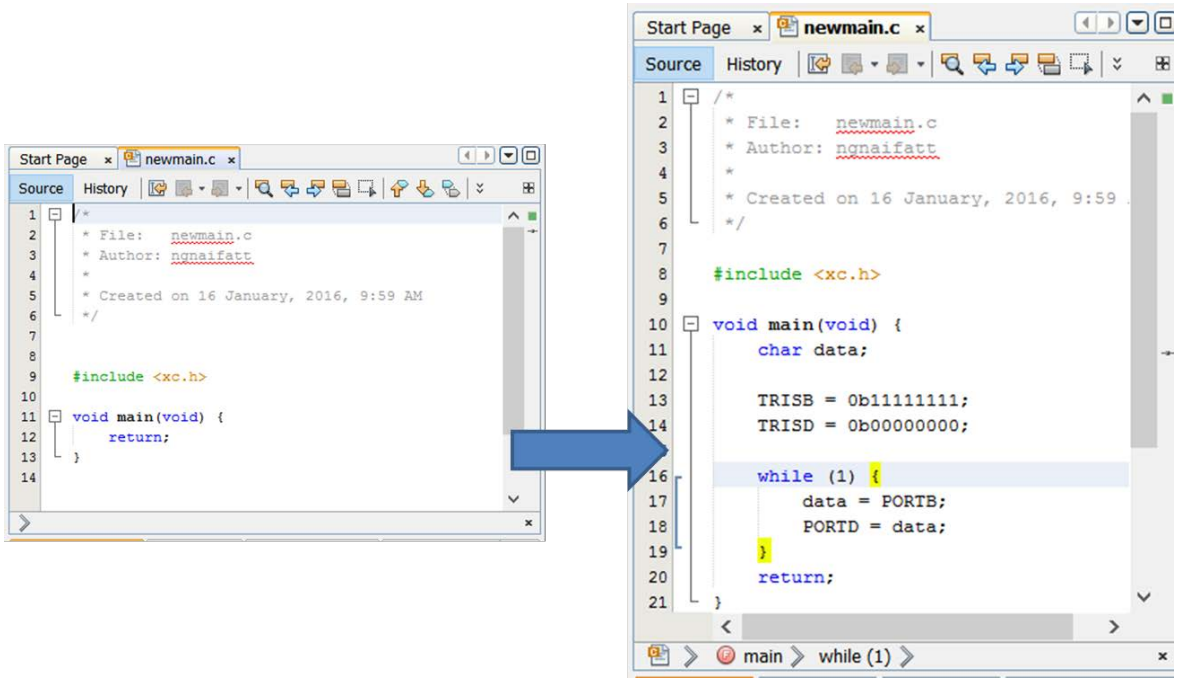
So click *Finish*.



13. The c file generated by the compiler has a brief description and an empty *main()* function:

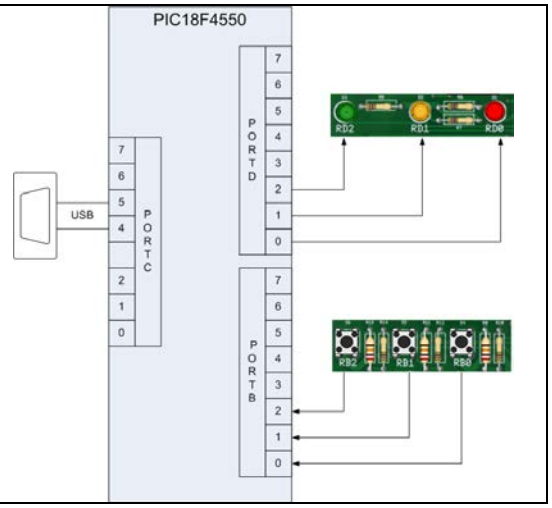


14. Add the lines as shown on the right to *newmain.c*.

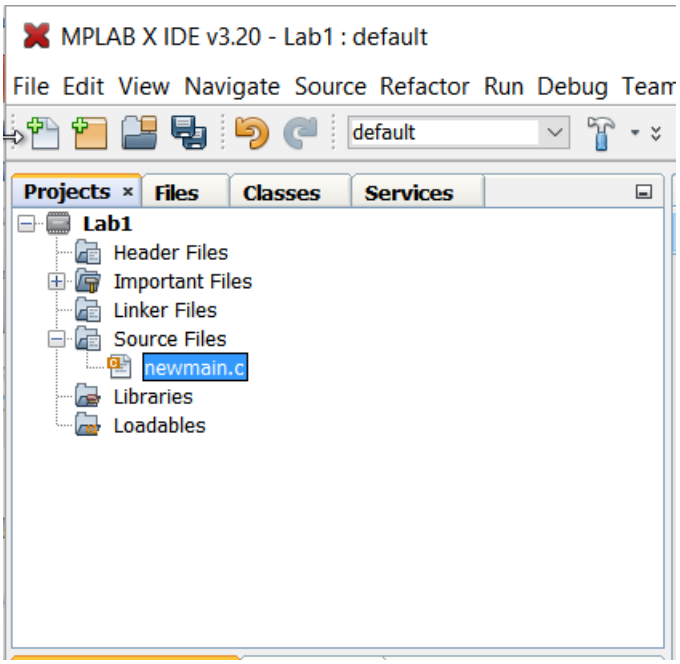


15. What the program is doing? Look at the comments for an explanation.

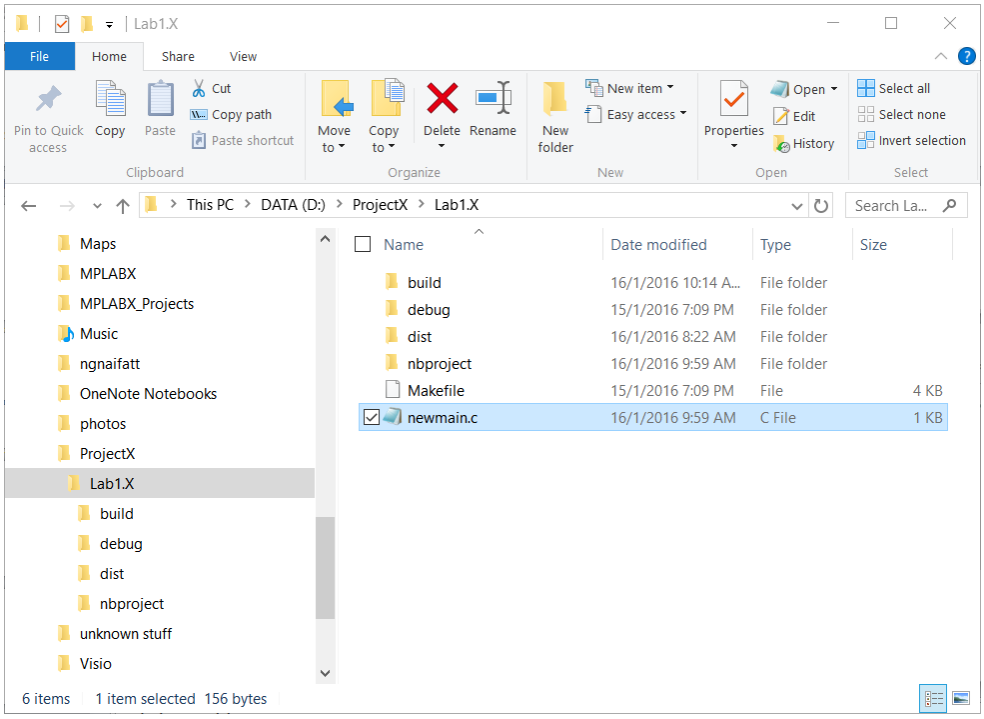
```
TRISB = 0b11111111;// PORTB is configured as all inputs

TRISD = 0b00000000;// PORTD is configured as
                   // all outputs

while (1) {    // repeat forever the code below

    data = PORTB; // read the inputs from PORTB
                  // and save in data

    PORTD = data; // output the data to PORTD
}
```



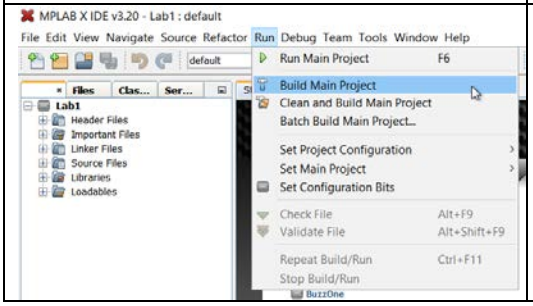
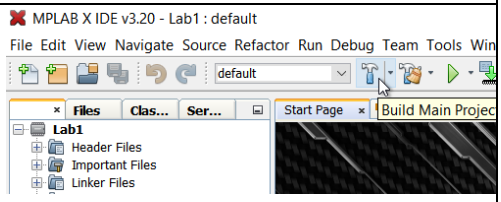
16. In the *Projects* tab, click on the [+] next to *Source Files* to expand it. You should be able to see *newmain.c* under *Source Files*:



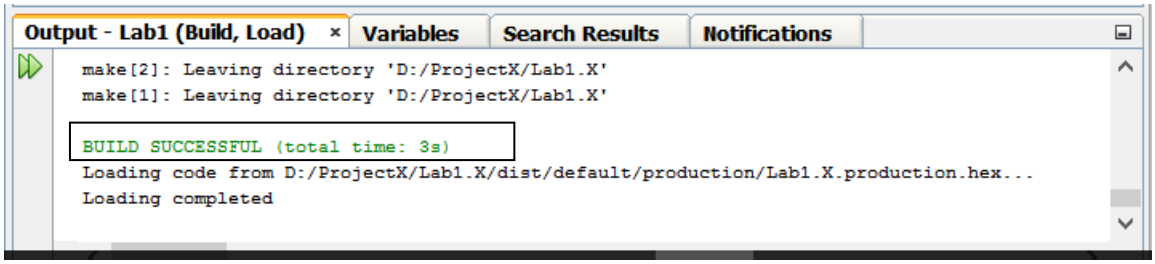
17. The file *newmain.c* is found in the folder *D:\ProjectX\Lab1.X folder*:



18. Let's build the project with a Hammer!

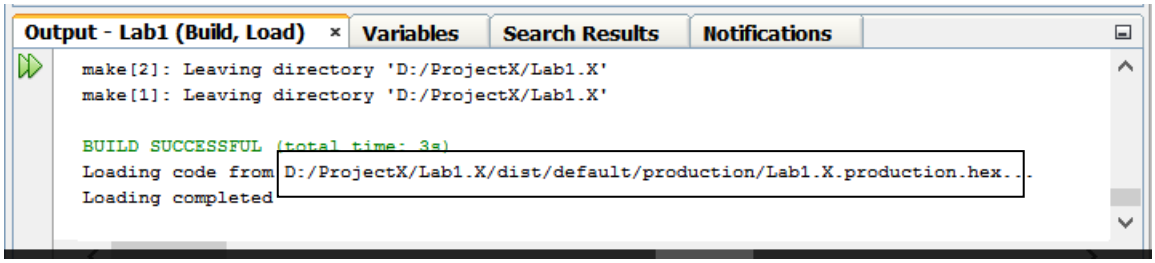
Method 1 : Run -> Build		Method 2: Click Hammer.
		

19. If there are no errors, you will see *BUILD SUCCESSFUL*.

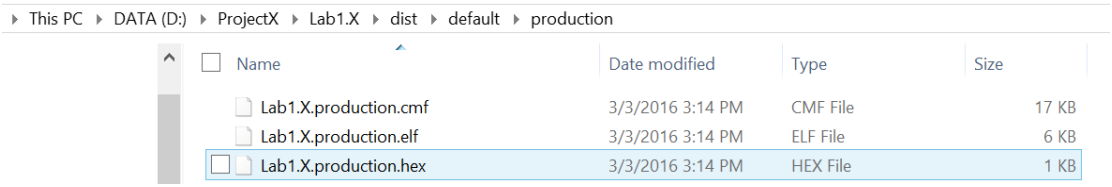


If there are errors, click on the error message and that will lead to the program line that has the error. After correcting the error, try to build again.

20. You can see the machine code/hex file *Lab1.X.production.hex* has been created.



21. Using *window explorer (my computer)*, browse to the folder *D:\ProjectX\Lab1.X\dist\default\production* and you should be able to see that hex file:



At this point, a C program (*newmain.c* which was created earlier) has been compiled into the hex code *Lab1.X.production.hex*.

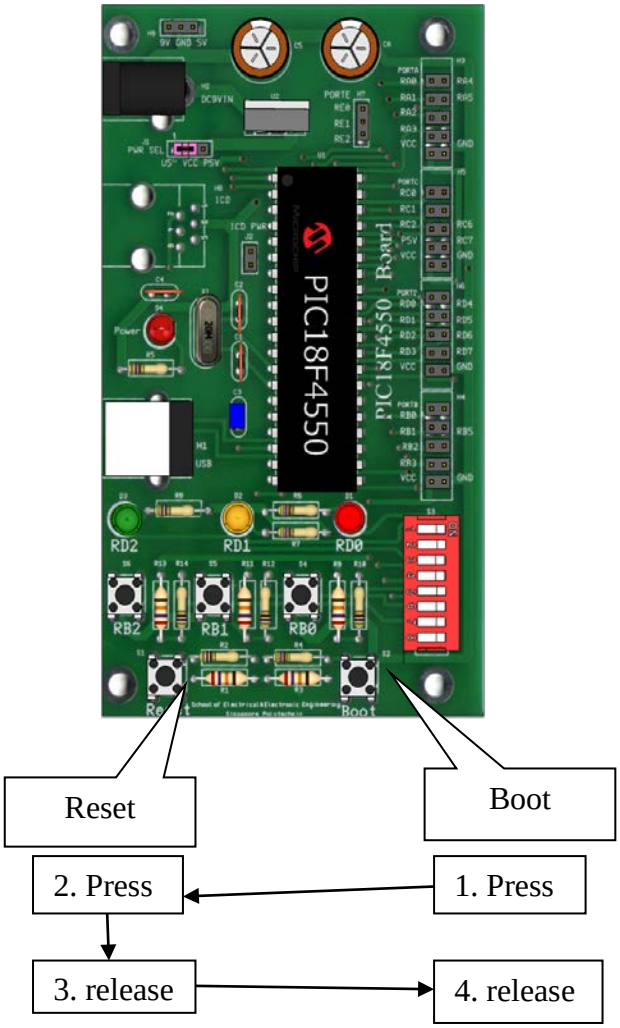
This is a machine code version of the *newmain.c* which must be downloaded to the microcontroller for it to be executed.

Downloading a program (a hex file) to the micro-controller and executing it.

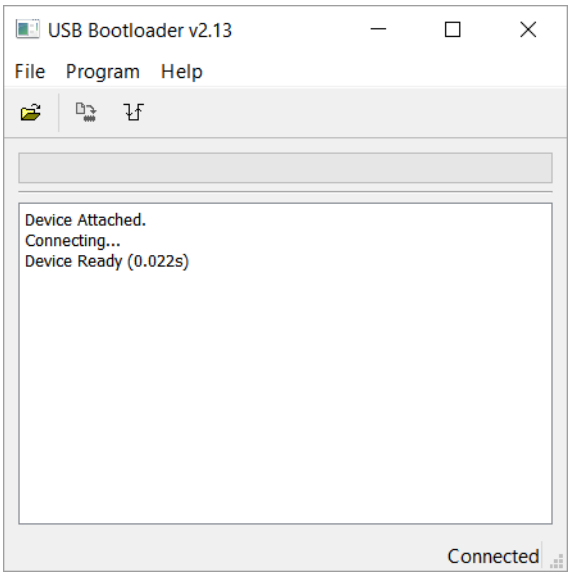
22. Connect the *PIC18F4550 board* to the *PC's USB port* using the *USB cable* provided.

23. Double click on the *HIDBootloader* icon on the desktop to launch the software.

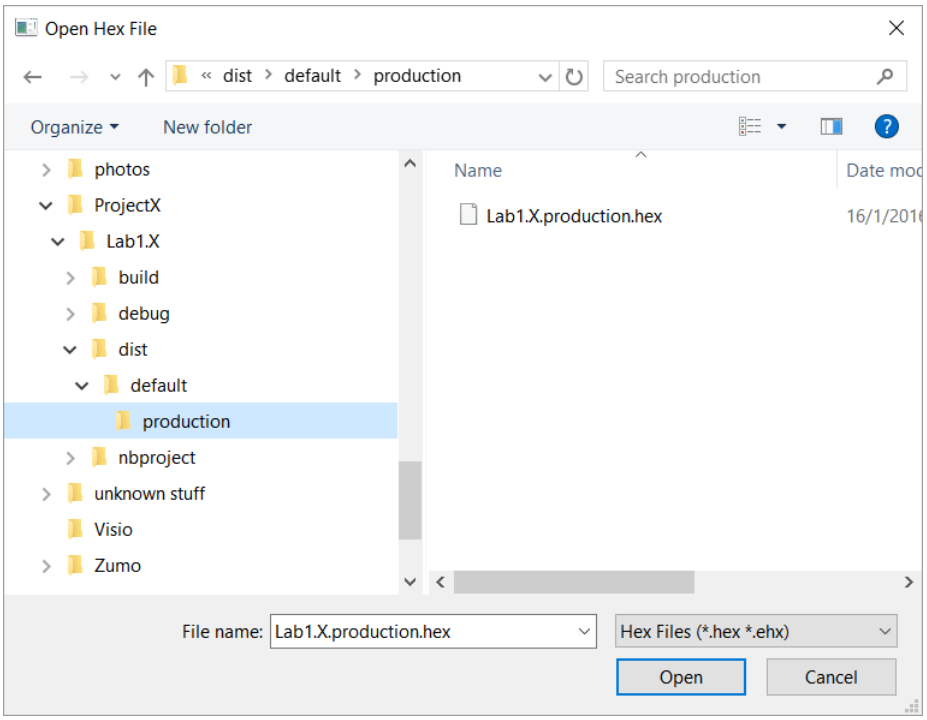
24. On the *PIC18F4550 board*, press & hold the *Boot* button, press and then release the *Reset* button, and finally release the *Boot* button.



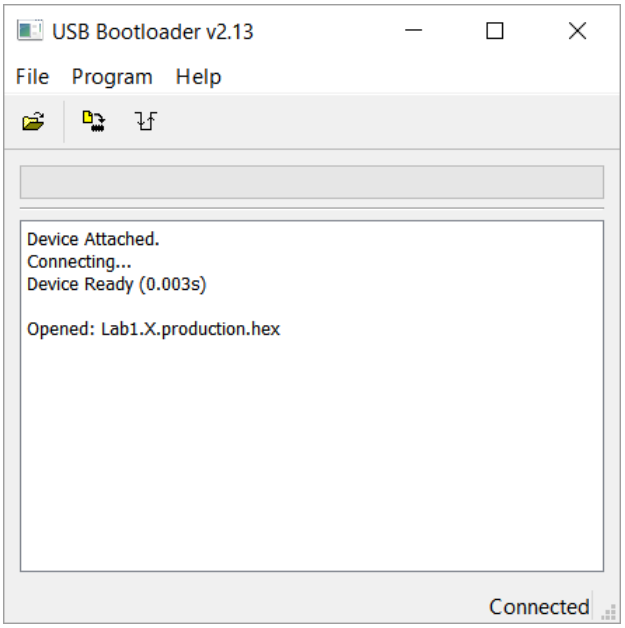
25. The *USB Bootloader* software will automatically detect the *PIC18F4550 board* and the message "*Device Ready*" will be shown:



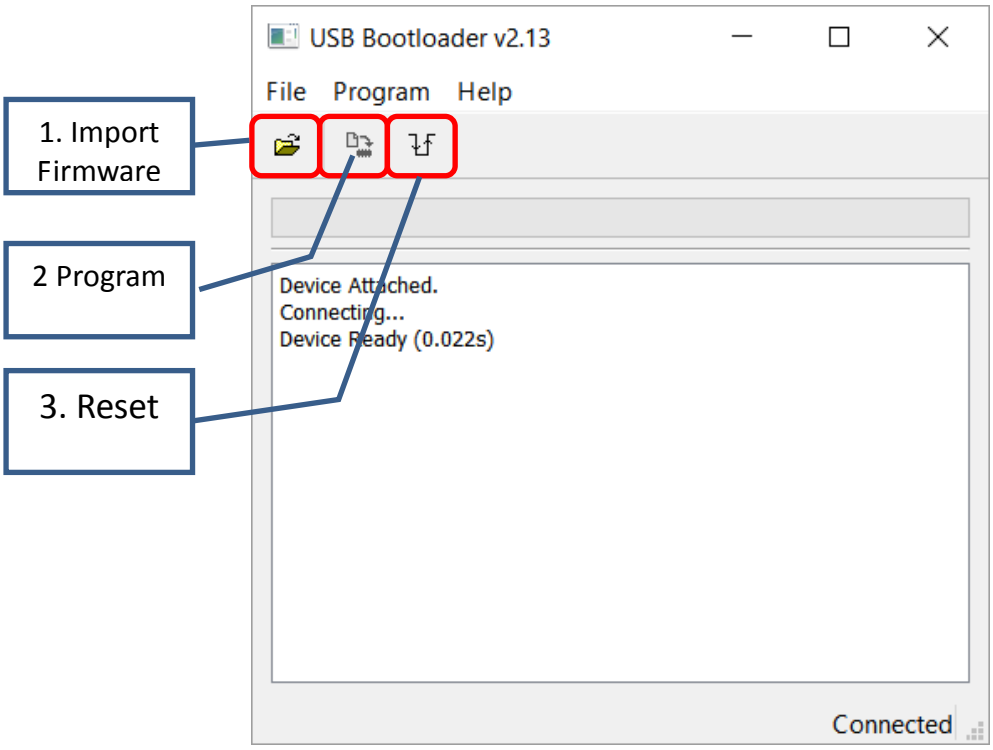
26. Click *File -> Import Firmware Image* and select the HEX file that was just created, *Lab1.X.production.hex*.



27. Click *Open* and the status will be updated as shown.



28. Click *Program Device* to program the micro-controller.



This diagram summaries the three steps needed to download a program.

29. Click *Reset* on the *USB Bootloader* software or pressing the *Reset* button on the *PIC18F4550 board* will run the program in the micro-controller.

30. Can you see the LED's (connected to *RD2*, *RD1* and *RD0*) light up?

31. Press the buttons connected *RB2*, *RB1* and *RB0* one by one. What is the effect of pressing a button?

    Your answer: _____

☐ At the end of each lab, there is an **Extra Exercise** for those who finish early. Doing this Extra Exercise allows you to learn more about micro-controller.

Extra Exercise

☐ In this exercise, you will modify the C-program to light up the LED's one by one.

32. Modify the C program to the following.

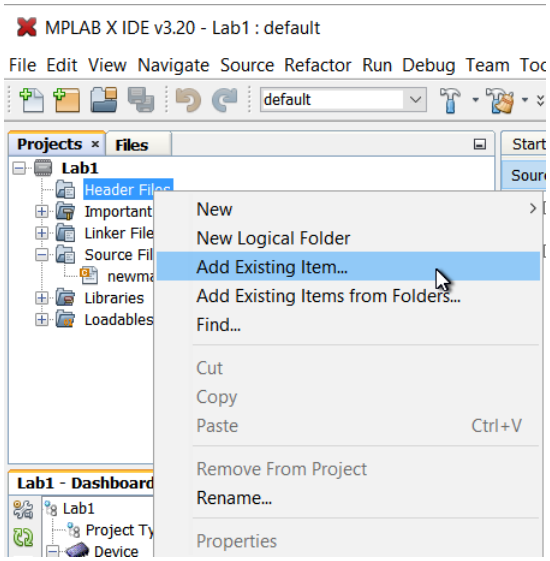
```
#include <xc.h>

void main(void) {
    TRISB = 0b11111111;
    TRISD = 0b00000000;
    while (1) { // repeat
        PORTD = 0b00000001;  // turn on LED in RD0
        delay_ms(500);       // delay for 500ms
        PORTD = 0b00000010; // turn on LED in RD1
        delay_ms(500);       // delay for 500ms
        PORTD = 0b00000100; // turn on LED in RD2
       delay_ms(500);       // delay for 500ms
    }

    return;
}
```

33. Build the program and is it successful?

Your answer: _____

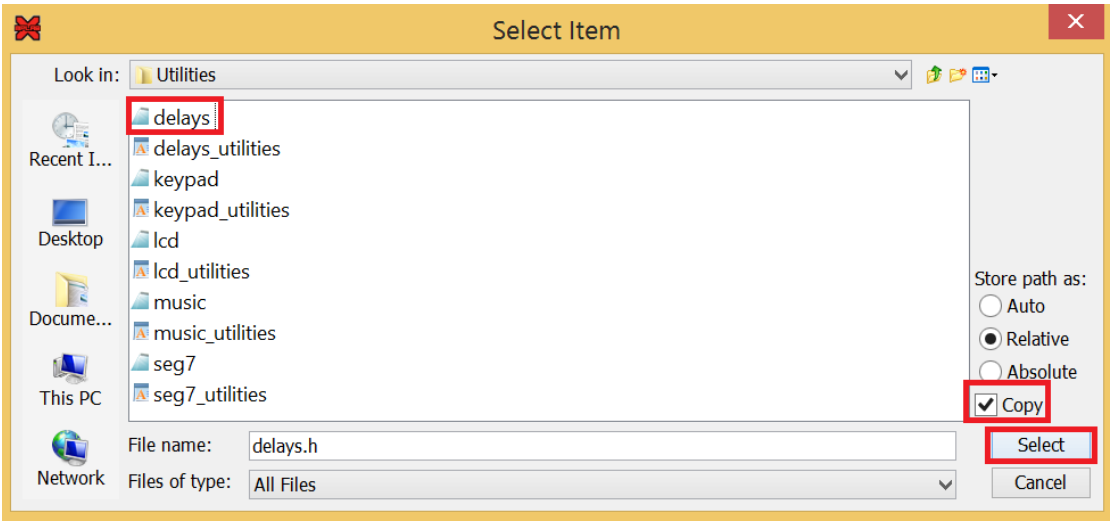
34. The error is because the compiler does not know what the function *delay_ms(500)* is. We need to add in the files *delays.h* to the *Header Files* and *delays_utilities.c* to the *Source files* of the project, as follows.

Right click on *Header Files -> Add Existing Item…*

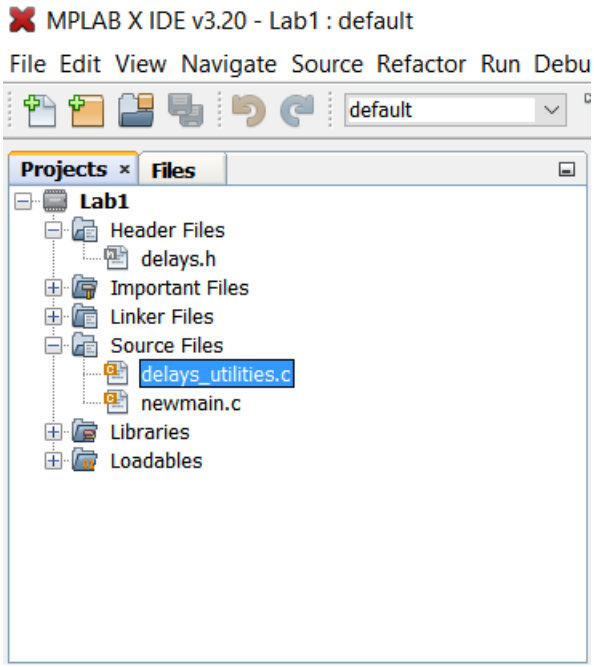


35. In the *Select Item* window that appears, browse to the folder *D:\ProjectX\Utilities* and select the file *delays.h* (the extension may not appear).

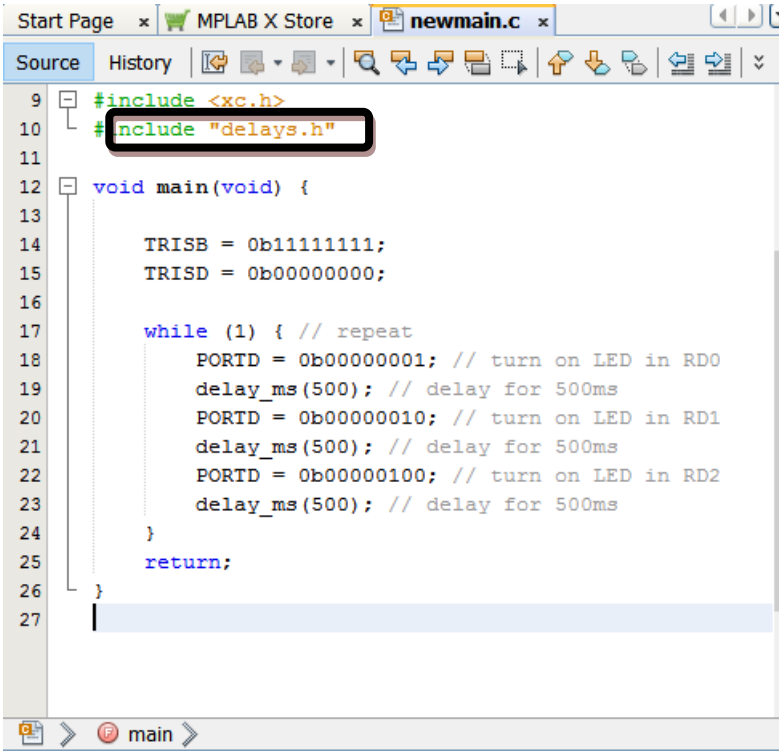
    Click *Copy* to get a copy of this file and then click *Select*.



36. The file *delays.h* is now added to the *Header Files*. Next, add the file *delays_utilities.c* to the *Source Files*. After this, you should see:



37. We also need to add the *#include "delays.h"* statement to the C program.



Build the project again and it should be successful. Download this program to the board and the LEDs should be lighting up one at a time with a delay of about half a second.

38. Try to slow down the rate of blinking by increasing the delays. (Hint: change all the values 500 to 1000).

Rebuild and download the program.

39. Then, make the LEDs blink faster and faster until it stops blinking.

Rebuild and download the program.

40. Finally, make the LEDs blink together by turn on and off all three LEDs at the same time. Use any reasonable delay of your choice.

Rebuild and download the program.

```
// file : delays.h

#define _XTAL_FREQ 48000000

extern void delay_ms(unsigned int i);  // delay in milli-secs, up to
                                       //max 65535 ms

extern void delay_us(unsigned int i);  // delay in micro-secs, up to
                                       //max 65535 us



/*
* File:   delays_utilities.c
*
* Created on 13 January, 2016, 1:31 PM
*/

#include <xc.h>
#define _XTAL_FREQ 48000000

void delay_ms(unsigned int i)
{ unsigned int j;
      if(i!=0)  // check for i=0
      for(j=0;j<i;j++)__delay_ms(1); // call __delay_ms(1) x i times
}

// this delay is too short to be accurate - good luck.
void delay_us(unsigned int i)
{
      unsigned int j,lower;
      // for micro sec, the looping takes too long
      // so split into two parts, 20 seems to work fine
      lower = i;
      lower = lower/20;

      if (i< 5)
      {
            return; // too short no delay
      }
      else if (i<10)
      {
            __delay_us(7); // delay is 5 to 9 so just pick 7
      }
      else if (i< 20)
      {
            __delay_us(15); // delay is 10-19
      }
      else
            for(j=0;j<lower;j++)__delay_us(20);

}
```

Lab 2 – Interfacing to switches and LED's

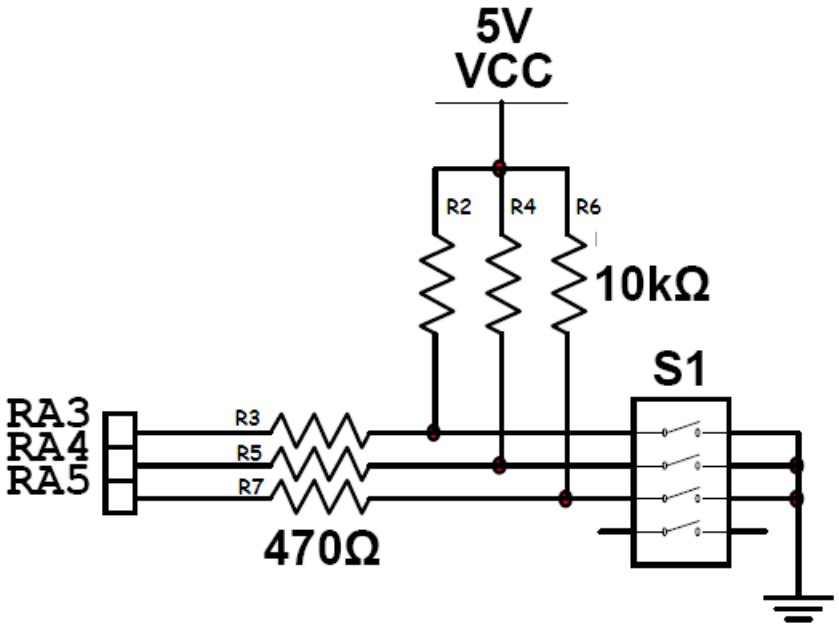
Objectives

- ☐ To learn to configure PIC18F4550's I/O ports as inputs or outputs.
- ☐ To learn to read status of switches – open or closed.
- ☐ To learn to turn on / off a number of LED's, in various sequences.
- ☐ To learn to use the delay functions.

Introduction / Briefing

Switches at Port A

- ☐ In this experiment, you will be reading the status of the dip switches connected to Port A.



Below are the ways in which the switches are named and used in software.
Switch at RA3 has the name PORTAbits.RA3
Switch at RA4 has the name PORTAbits.RA4
Switch at RA5 has the name PORTAbits.RA5
PORTA on its own refers to all 8 bits and not individual bits.

☐ Study the above diagram and answer the following questions:

Q1: How many dip switches are there? _____

Q2: How many are connected to Port A? _____

Q3: What is the purpose of the 470 ohm resistors?

_____

(The above is a tough question. Hint: imagine someone making the mistake of configuring Port A as output AND producing a logic '1' at one of RA3:RA5 AND the corresponding dip switch is closed.)

Q4: Are the switches connected in the "active high" or "active low" manner? (An "active high" switch gives a logic '1' when closed.)

_____

Q5: What will a Port A pin read (logic '1' or '0') when a corresponding dip switch is closed? _____

☐ To allow Port A to read the dip switches, it must be configured as a digital input port. However, Port A is a partially analogue/partially digital input port by default (after power on reset):

[Refer to the "insert" on the next page to understand the last column.]

Port	Available pins	Not available as general purpose I/O (- reasons)	After power on reset
A	RA6-0	RA6 (– oscillator)	RA5, 3-0: Analogue inputs (*). RA4: Digital input.

Q6: Give the C-command to configure Port A as a digital input port (hint: ADCON1):

_____ // configure Port A as digital inp.

REGISTER 21-2: ADCON1: A/D CONTROL REGISTER 1

U-0	U-0	R/W-0	R/W-0	R/W-0(1)	R/W(1)	R/W(1)	R/W(1)
—	—	VCFG0	VCFG0	PCFG3	PCFG2	PCFG1	PCFG0
bit 7							bit 0

bit 3-0 **PCFG3:PCFG0:** A/D Port Configuration Control bits:

Default value of PCFG3 is 0, making AN4-0 all analogue inputs.

AN4 is also RA5, while AN3-0 are also RA3-0.

The C code ADCON1 = 0x0F; puts binary 1111 into PCFG3-0, making AN4-0 i.e. RA5, RA3-0 all digital.

PCFG3: PCFG0	AN12	AN11	AN10	AN9	AN8	AN7(2)	AN6(2)	AN5(2)	AN4	AN3	AN2	AN1	AN0
0000(1)	A	A	A	A	A	A	A	A	A	A	A	A	A
0001	A	A	A	A	A	A	A	A	A	A	A	A	A
0010	A	A	A	A	A	A	A	A	A	A	A	A	A
0011	D	A	A	A	A	A	A	A	A	A	A	A	A
0100	D	D	A	A	A	A	A	A	A	A	A	A	A
0101	D	D	D	A	A	A	A	A	A	A	A	A	A
0110	D	D	D	D	A	A	A	A	A	A	A	A	A
0111(1)	D	D	D	D	D	A	A	A	A	A	A	A	A
1000	D	D	D	D	D	D	A	A	A	A	A	A	A
1001	D	D	D	D	D	D	D	A	A	A	A	A	A
1010	D	D	D	D	D	D	D	D	A	A	A	A	A
1011	D	D	D	D	D	D	D	D	D	A	A	A	A
1100	D	D	D	D	D	D	D	D	D	D	A	A	A
1101	D	D	D	D	D	D	D	D	D	D	D	A	A
1110	D	D	D	D	D	D	D	D	D	D	D	D	A
1111	D	D	D	D	D	D	D	D	D	D	D	D	D

A = Analog input D = Digital I/O

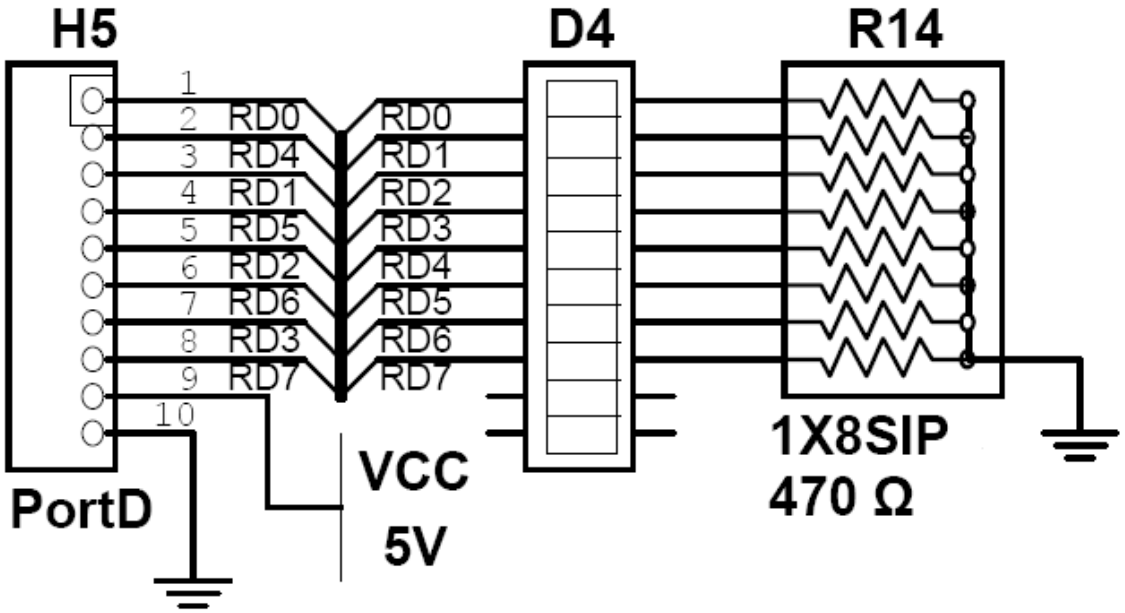
☐ When a dip switch is closed, the corresponding Port A pin will read a logic '0'.

Q7: Give the C-command to check if the dip switch connected to RA3 is closed (hint: PORTA):

```
if _____ // if dip switch @ RA3 is closed
        {
        // do something

        }
```

LED bar at Port D

☐ In this experiment, you will also be turning on and off an LED bar connected to Port D.



☐ Study the above diagram and answer the following questions:

Q8: How many LED's are there in the LED bar? _____

Q9: How many are connected to Port D? _____

Q10: What is the purpose of the 470 ohm resistors in the SIP (Single-In-Line package)? _____

Q11: Are the LED's connected in the "common anode" or "common cathode" manner? _____

Q12: What must a Port D pin produce (logic '1' or '0') to turn on a corresponding LED? _____

☐ To allow Port D to control the LED bar, it must be configured as a digital output port. However, Port D is a digital input port by default (after power on reset):

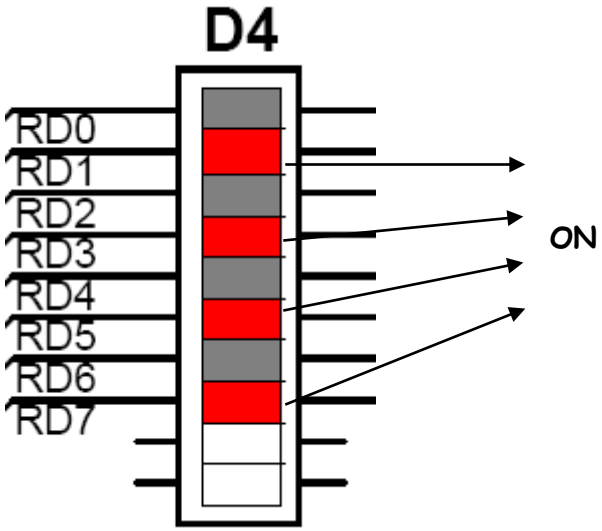
Port	Available pins	Not available as general purpose I/O (- reasons)	After power on reset
D	RD7-0		RD7-0: Digital inputs.

Q13: Give the C-command to configure Port D as a digital output port (hint: TRISD):

_____ // configure Port D as digital outp.

☐ To turn on a particular LED, the corresponding Port D pin must produce a logic '1'.

Q14: Give the C-command to turn on the LED's as follows (hint: PORTD):



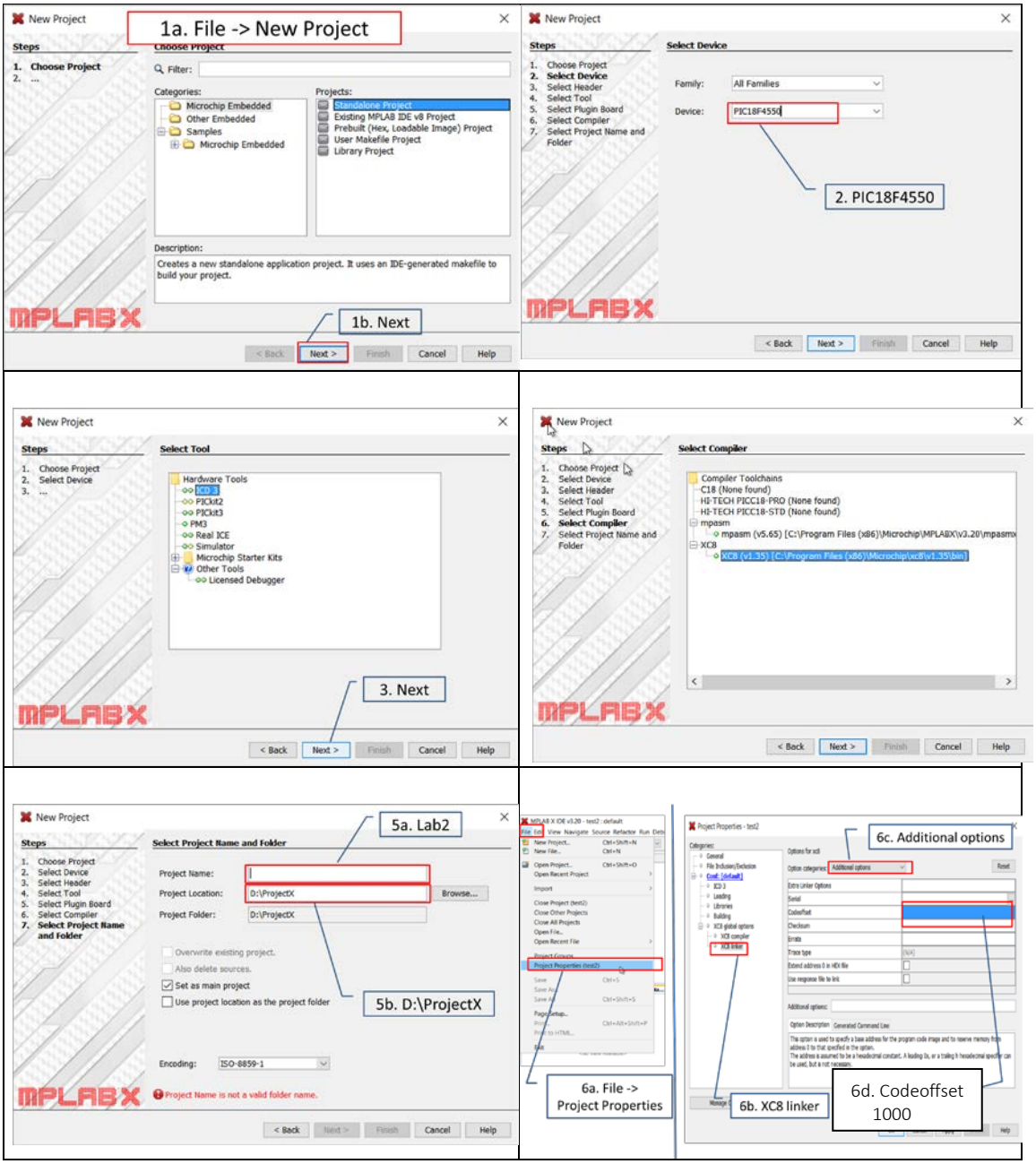
_____ // turn alternate LED's on

Activities:

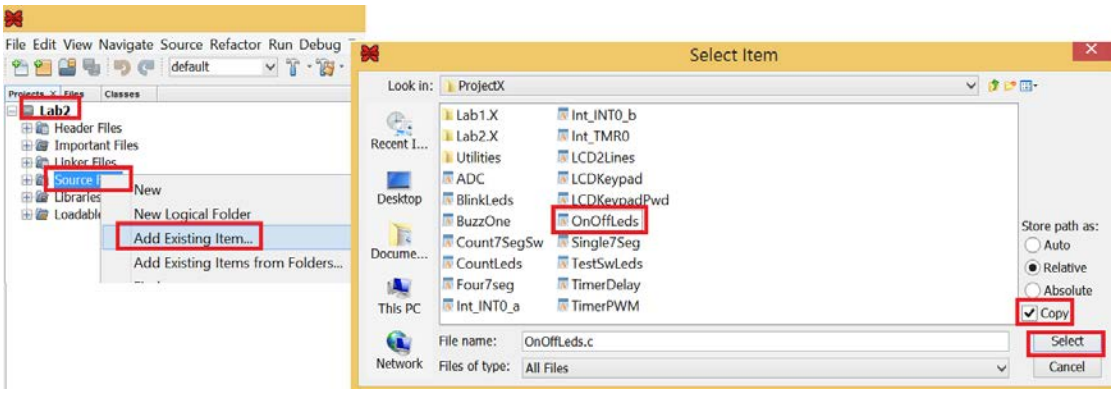
Using LEDs to indicate switch status

Create a New Project – Lab2

1. Launch the *MPLABX IDE* and create a new project called Lab2.
   Below is a summary of the steps needed or you can refer to Lab1.



2. Use "*Add Existing Item*" to add the file *OnOffLeds.c* to the *Lab2* project *Source File* folder, as follows. Make sure *Copy* is ticked before you click *Select*.



3. Double click on *Source Files – OnOffLeds.c* to look at the program.

```
// OnOffLeds.c
// Program to use switches to control 8 leds on General I/O Board

#include <xc.h>

void main(void) {
    ADCON1 = 0x0F; //make Port A digital

    TRISA = 0b11111111; //RA5 to RA3 are connected to On/Off switches

    TRISD = 0b00000000; //RD7 to RD0 are connected to LEDs

  while (1) //repeat
   {
      if (PORTAbits.RA3 == 0) //_____________________
      {
         PORTD = 0xF0;     // PORTD = 0b_____________________
      }
      else
      {
        PORTD = 0x0F;     // PORTD = 0b_____________________
      }
   }
}
```

4. Describe what this program will do:

_____

_____

5. Build, download and execute the program. Observe the result and see if it is as expected.

6. Change the program to use the switch at RA5, such that if RA5 is on, all the LEDs should be on and if RA5 is off, all the LEDs should be off.

7. Build, download and execute the program. Observe the result and see if it is as expected.

LED's blinking / "scanning"

Use delays, toggle LED's.

8. Right click on *OnOffLeds.c* and click "*Remove From Project*". Then add the file *BlinkLeds.c* to the project. ("*Remove From Project*" only removes the file from the project, it is not deleted.)

9. Study the code and describe what this program will do:

_____

10. Note that the program uses the delay function *delay_ms()* and contains *#include "delays.h"*. The file *delays.h* needs to be added to *Header Files* while the file *delays_utilities.c* needs to be added to *Source Files*.

11. Build, download and execute the program. Observe the result and see if it is as expected.

Pause an action

12. Add the following line to the *while(1)* loop:

```
while(1)
{
        while (PORTAbits.RA3 == 0); // loop here when switch is on

        PORTD=0b10101010;
        ……. // other existing lines – don't touch
}
```

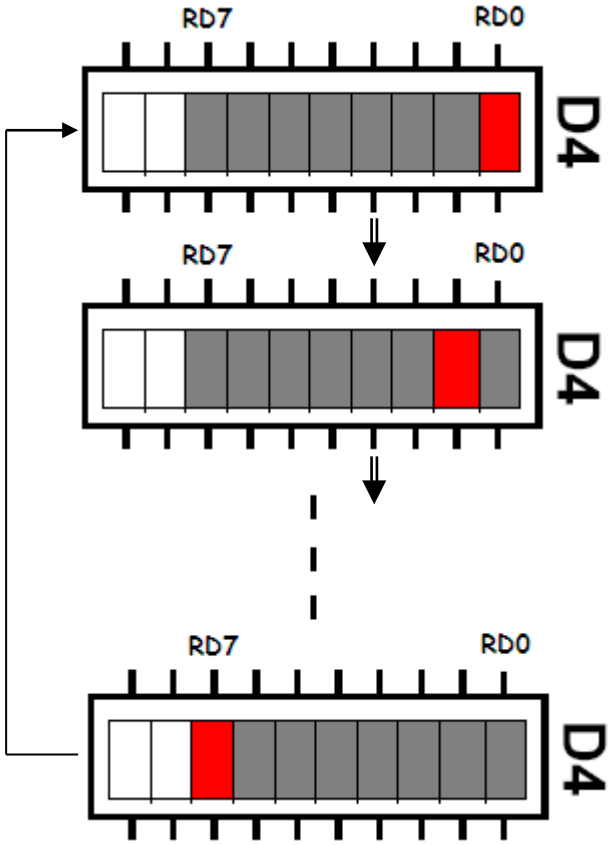
13. Describe what this NEW program will do:

_____

14. Build, download and execute the program. Observe the result and see if it is as expected.

Right to left "scan"

15. Modify the program to do "scanning", such that one LED light repeatedly moves from right to left (after a short delay).



Pause the "scan"

16. Add in a switch control line such that when the switch connected to RA3 is closed, the "scanning" is paused.

17. Debug until the program can work.

2-way "scan"

18. Modify the program to do a "right to left scan", followed by a "left to right scan" repeatedly. Include the switch control line to pause the scanning with a closed switch at RA3.

19. Debug until the program can work.

Slow down the "scan"

20. Modify the program such that a closed switch at RA4 slows down the scanning (while a closed switch at RA3 pauses the scanning). (Hint: add delay if switch at RA4 is closed.)

21. Debug until the program can work.

LED's "counting"

Counting

22. Replace *BlinkLeds.c* with *CountLeds.c*.

23. Study the code and describe what this program will do:

_____

24. Build, download and execute the program. Observe the result and see if it is as expected.

Counting Up/down

25. Modify the program such that a closed switch at RA5 causes counting up while an opened switch causes counting down. Here are some hints:

```
while (1)
{
        ….. // other lines unchanged
        if (PORTAbits.RA5 == 0) // if switch is closed
                _____ // count up
        else
                _____ // count down
}
```

Slow down the counting

26. Add in a line such that a closed switch at RA4 slows down the counting.

Pause the counting

27. Add in another line such that a closed switch at RA3 pauses the counting.

28. Debug until the program can work.

Extra Exercise

29. A left-shift is equivalent to multiplication by 2 while a right-shift is equivalent to division by 2.

2-way scan, with pausing & slowing down

30. If you still have time at the end of this Lab, try to write a LED scanning program (you can modify any existing file – OnOffLeds.c or BlinkLeds.c or CountLeds.c) such that the scanning is normally from right to left, but

    A closed switch at RA5 causes a left to right scan.
    A closed switch at RA4 causes scanning to slow down.
    A closed switch at RA3 causes scanning to pause.

31. Debug until the program can work.

```
// OnOffLeds.c
// Program to use 1 switch to control 8 leds on General I/O Board

#include <xc.h>

void main(void) {
    ADCON1 = 0x0F;       //make Port A digital

    TRISA = 0b11111111; //RA5 to RA3 are connected to On/Off switches

    TRISD = 0b00000000; //RD7 to RD0 are connected to LEDs

    while (1) //repeat
    {
      if (PORTAbits.RA3 == 0) //_____________________
        {
            PORTD = 0xF0;      // PORTD = 0b_____________________
        }
        else
        {
            PORTD = 0x0F;      // PORTD = 0b_____________________
        }
    }
}
```

// BlinkLeds.c

```
// Program to light up alternate leds on General I/O Board

#include <xc.h>
#include "delays.h"

void main(void) {
    ADCON1 = 0x0F; //make Port A digital

    TRISA = 0b11111111; //RA5 to RA3 are connected to On/Off switches

    TRISD = 0b00000000; //RD7 to RD0 are connected to LEDs

    while (1) //repeat
    {
        PORTD = 0b10101010;
        delay_ms(1000);
        PORTD = 0b01010101;
        delay_ms(1000);

    }
}
```

// CountLeds.c

```
// Program to make 8 leds on General I/O Board do binary up counting

#include <xc.h>
#include "delays.h"

unsigned char j;  // 8 bit data type, range 0 to 255

void main(void)
{
      ADCON1 = 0x0F;    //make Port A digital

      TRISA=0b11111111; //RA5 to RA3 are connected to On/Off switches

      TRISD=0b00000000; //RD7 to RD0 are connected to LEDs

      j=0;                    //beginning

      while(1)                //repeat
      {

            PORTD = j;        // Output value of j to PORTD

           delay_ms(500);

            j++;              // Increment j
      }
}
```

Lab 3 – Interfacing to 7-segment displays and buzzer

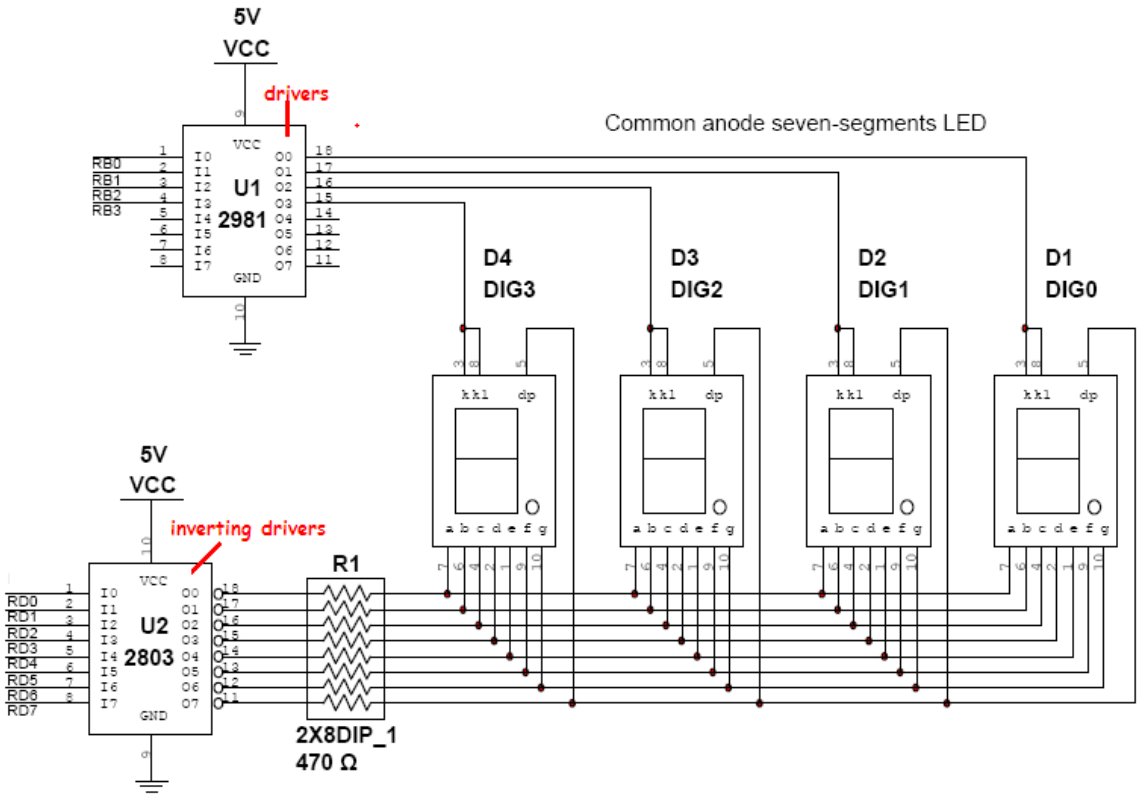
Objectives

- ☐ To learn to display a decimal number on a 7-segment display.
- ☐ To learn to use multiplexing technique to display several digits on several 7-segment displays.
- ☐ To learn to implement a "queue number system".
- ☐ To learn to produce a tone on a buzzer.

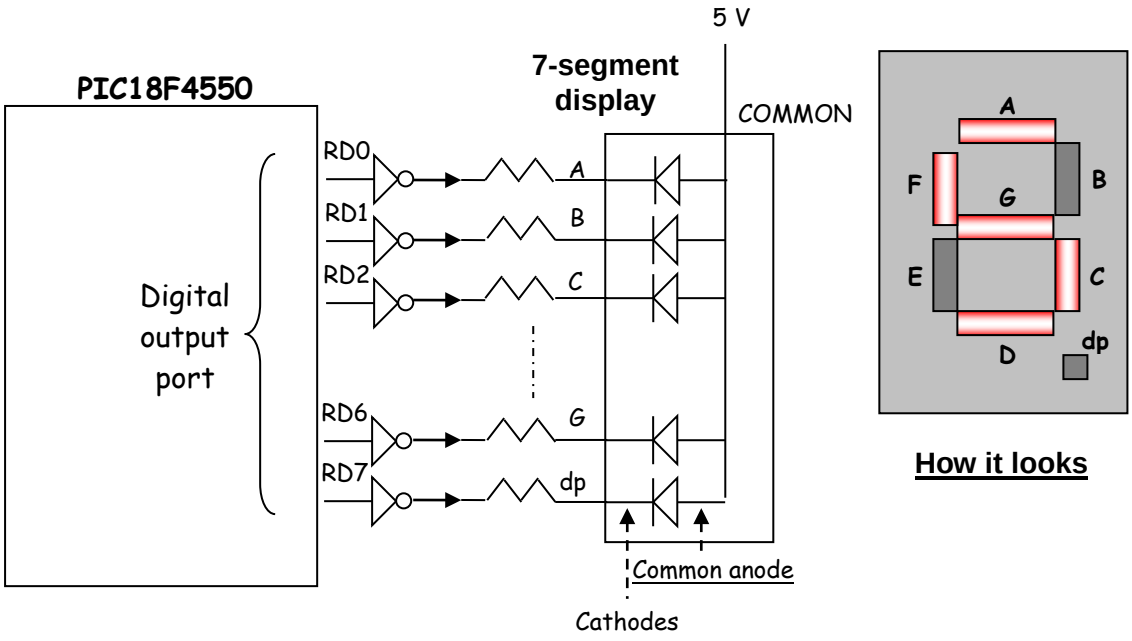
Introduction / Briefing

7-segment display at Ports B & D

- ☐ In this experiment, you will be turning on and off segments in four 7-segment displays connected to Ports B & D, to display some numbers.



☐ Considering only one 7-segment display (shown below) and answer the following questions:



Q1: Are the LED's in the 7-segment display connected in the "common anode" mode or the "common cathode" mode? _____

Q2: What must RD0 produce (logic '0' or logic '1') to turn on segment A?
_____

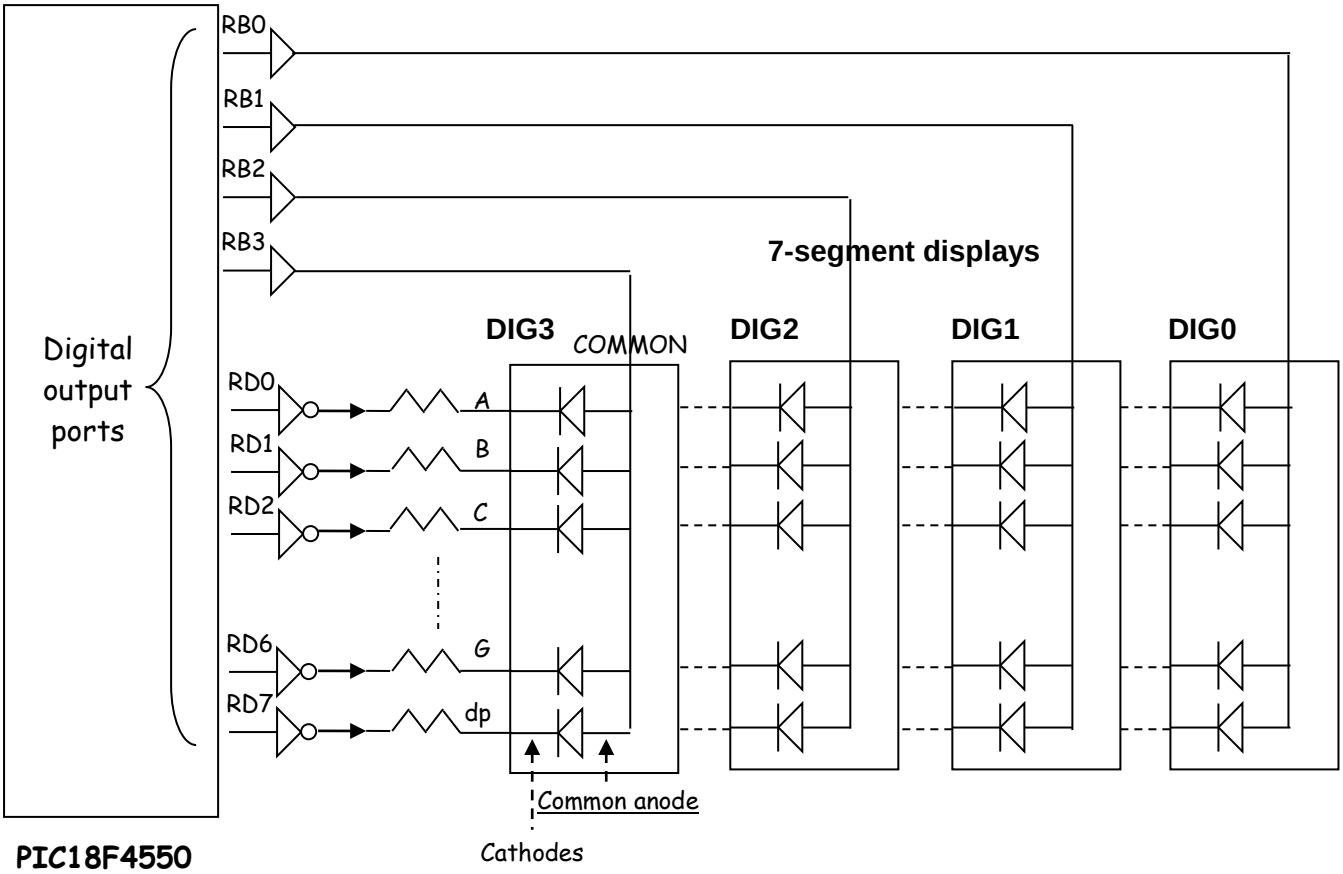
Q3: What must PORT D produce (in binary format) to show the digit "1" on the 7-segment display? PORTD = 0b_____.

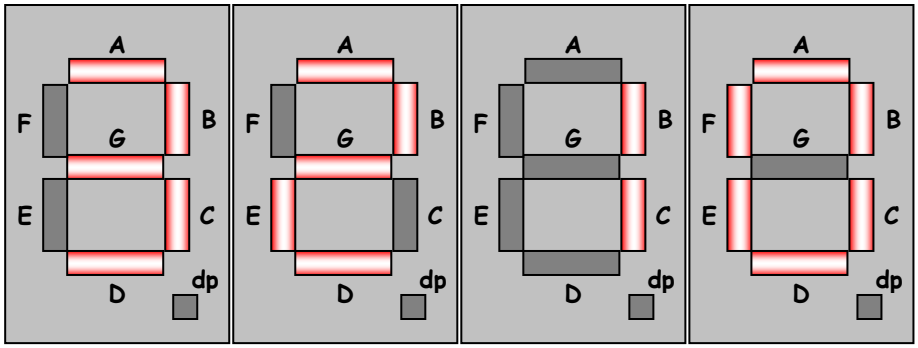
Q4: Of course, PORT D must be configured as an output port. Give the 2-line C command to configure PORT D as a digital output port and to show the digit "5" on the 7-segment display:

TRISD = 0b_____ // configure Port D as digital outp.

PORTD = 0b_____ // display "5"

☐ Considering four 7-segment displays together (shown below) and answer the following questions:





Q5: What will be shown on the 7-segment displays if PORT D outputs 0b01001111 while PORT B outputs 1000 to its lower 4 bits?

DIG3 shows _____
DIG2 shows _____
DIG1 shows _____
DIG0 shows _____

Q6: What must PORT B and PORT D produce to show 2 on DIG2?

PORT D = 0b _____

PORTBbits.RB3 = _____
PORTBbits.RB2 = _____
PORTBbits.RB1 = _____
PORTBbits.RB0 = _____

Q7: What will be shown on the 7-segment displays if following C program is run?

```
TRISB = 0b11110000;        // lower 4 bits are outputs
TRISD = 0b00000000;        // all bits are outputs

while (1)
{
        PORTB = 0b00000001;        // enable DIG0
        PORTD = 0b00111111;        // display 0
        // Some delay

        PORTB = 0b00000010;        // enable DIG1
        PORTD = 0b00000110;        // display 1
        // Some delay

        PORTB = 0b00000100;        // enable DIG2
        PORTD = 0b01011011;        // display 2
        // Some delay

        PORTB = 0b00001000;        // enable DIG3
        PORTD = 0b01001111;        // display 3
        // Some delay
}
```

Your answer:_____

_____

Q8: What do you think will happen if the delay is increased?

Your answer: _____

Q9: What do you think will happen if the delay is decreased?

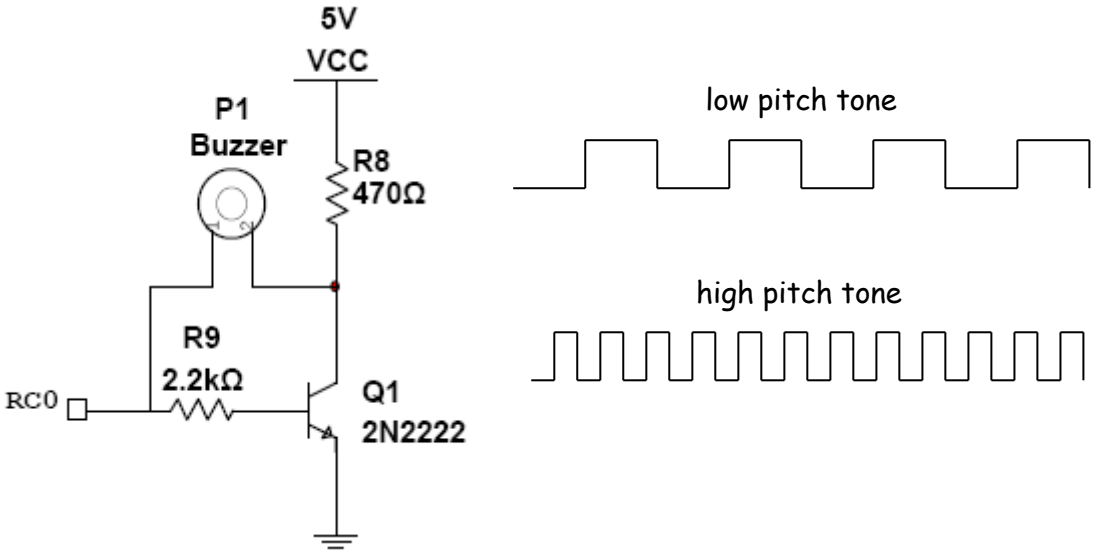
Your answer:

_____

You will find out the answer to the two previous questions in the experiment.

Buzzer at Port C

☐ In this experiment, you will also be turning on and off a buzzer connected to PORT C to produce a "tone".



☐ Study the above diagram and answer the following questions:

Q10: What happens when RC0 outputs logic '1'?

The transistor is turned on and (assuming $V_{CE[sat]}$ = 0.2V,) pin 2 of the Buzzer is at _____ V while pin 1 of the Buzzer is at _____ V. So the Buzzer will be turned _____. If RC0 outputs logic '0', the Buzzer will be turned _____.

☐ By toggling (on -> off -> on -> off ….) RC0 continuously, a tone can be produced by the buzzer.

☐ If the rate of toggling is high, a high pitch tone is produced.

Activites:

Before you begin, ensure that the Micro-controller Board is connected to the General I/O Board. The General I/O Board is further connected to a 7-Segment/Switch Board.

PORTD – 8 segments ('a' to 'g', and decimal point) of all 4 digits, active high ('1' turns on a segment, '0' turns off a segment).

PORTB – RB0 to RB3 – COM pins of all 4 digits (DIG0 to DIG3), active high ('1' enables digit, '0' disables digit).

PORTB – RB5 – push button switch, active low (pressed gives '0', released gives '1').

So PORTD controls the number e.g. '8' to be displayed on a digit, while PORTB controls which digit displays the number.

Displaying a decimal number on a 7-segment display

1. Launch the *MPLABX IDE* and create a new project called *Lab3*.

Display "0" on DIG0. ⇨

2. Add the file *Single7Seg.c* to the *Lab3* project *Source File* folder. Make sure *Copy* is ticked before you click *Select*. If you have forgotten the steps, you will need to refer to the previous lab sheet.

3. Study the code and describe what this program will do:

   _____

4. Build, download and execute the program. Observe the result and see if it is as expected.

Display "1" on DIG1. ⇨

5. Modify the code to display the digit "*1*" on the next 7-segment i.e. DIG1. Build, download and execute the program to verify your coding.

6. Describe what will happen when *PORTB = 0b00001111*. Why?

   Answer: _____

Display 4 decimal numbers on four 7-seg.'s

Displaying 4 different decimal numbers on four 7-segment displays

7. Replace *Single7Seg.c* with *Four7Seg.c*. Note that the program uses the delay function *delay_ms()* and contains *#include "delays.h"*. The files *delays.h* and *delays_utilities.c* need to be added to the Project.

8. Study the code and describe what this program will do:

_____

9. Build, download and execute the program. Observe the result and see if it is as expected.

Increase delay

10. Increase the delay between digits. What do you observe?

_____

Decrease delay

11. Decrease the delay between digits. What do you observe?

_____

12. As can be seen, multiplexing technique here involves turning on only one digit of display at a time, and after a short delay, move on to the next digit etc:

Show '0' on digit DIG0.
Delay
Show '1' on digit DIG1.
Delay
Show '2' on digit DIG2.
Delay
Show '3' on digit DIG3.
Delay
Repeat above

The delay is to give time for the LED's to light up and the number to be seen. Too long a delay will cause the numbers to flicker and too short and the display will become blur, as the LED's do not have time to turn on properly and be seen.

13. You may try to display today's date as DDMM and show it to your classmates.

Extra Exercise - Implementing a "queue number system"

(Do this only if you still have time. Otherwise, skip to the next section to try out the "buzzer".)

Q-no. system

14. Replace *Four7Seg.c* with *Count7SegSw.c*.

15. Study the code and describe what this program will do:

_____

16. Read the following explanation if you are stuck.

17. The decimal numbers to display on the four 7-segments are stored in an array of 4 unsigned chars

    unsigned char val[4]; // i.e. val [0], val [1], val [2], val [3]

    These are initialised to *val [3] = 9; val [2] = 8; val [1] = 7; val [0] = 6;* in the *main* program. So, the initial display should be "9 8 7 6".

18. 9876 are what you want to see. However, what the 7-segments want to be told are the binary patterns *0b01101111 [9], 0b01111111 [8], 0b00000111 [7], 0b01111101 [6].*

19. The function *convert* (in the *seg7_utilities*) produces the binary pattern required to show a decimal number on a 7-segment. E.g. if decimal number ("*digit*") = "0", binary pattern ("*leddata*") = 0b00111111.

20. As you know by now, multiplexing technique is used to enable each of the 4 digits in turn, so that the number of PIC pins required to display 4 digits (including the decimal points) is fewer than 4 x 8.

21. Here an unsigned char variable *point* is used to control which digit is lighting up.

22. It is initialised to *0b00000001* i.e. *DIG0* will light up first.

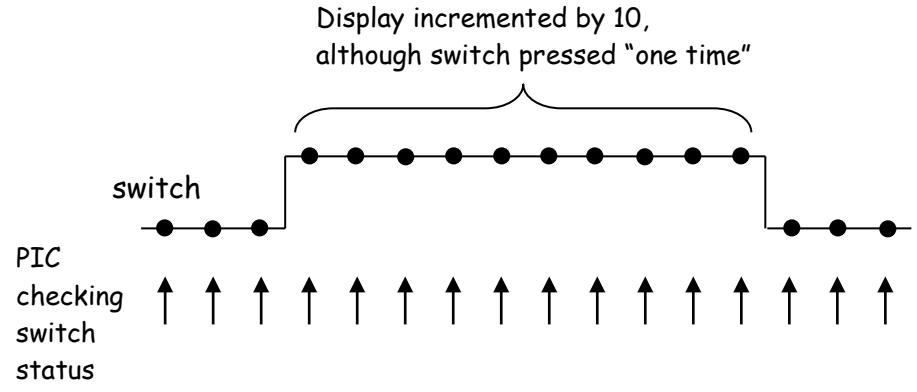
23. Putting all these ideas together, you get the following chunk of codes:

```
point = 0b00000001;  // enable DIG0 first
for (i = 0; i < 4; i++)  // loop from DIG0 to DIG3
{
 PORTB = point;    // enable one DIG
 outchar = val [i];  // get one decimal number to display from the array
                     // convert to corresponding binary pattern for the 7-seg,
 PORTD = convert (outchar);  // and send the binary pattern to the enabled DIG

 point = point << 1; // shift left by 1 bit, to enable the next DIG,
        // so 0b00000001 becomes 0b00000010, then 0b00000100,
        // then 0b00001000

 … // some delay
}
```

24. Whenever the switch connected to *RB5* is pressed, the 4-digit display is incremented by 1. This is done by the following lines of code and the function *update*:

```
if (PORTBbits.RB5 == 0) // if switch is pressed
{
 press = 1; // this is explained below
 val [0] = val [0] + 1;  // increment the lowest digit by 1
 update ();  // update the other digits accordingly
}
```

25. A micro-controller can work very fast. When a switch is pressed "one time", a micro-controller could have read it several times, and increment the display several times, as shown below:



26. To solve this problem, a variable *press* is used to control the flow of the program:

```
…
press = 0; // initially… in the "switch not pressed" state
while (1)
{
  … // display the decimal numbers

  if (press == 0) // starting from the "switch not pressed" state
   {
     if (PORTBbits.RB5 == 0) // if switch pressed
     {
       press = 1; // change to the "switch pressed" state
       … // increment the 4-digit number to display
     }
   }

  if (PORTBbits.RB5 == 1) // switch released
    press = 0; // change to the "switch not pressed" state, ready for next round
}
```

Display is only incremented the first time the switched is found to be pressed.

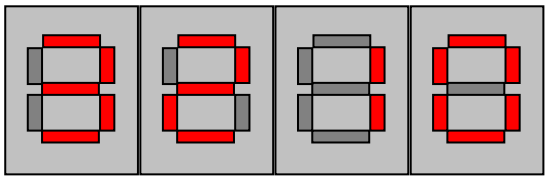
Subsequently, only check for the release of the switch.

27. The best way to check whether you have understood this program is to try to explain it to a classmate.

28. Once you have understood it, build, download and execute the program. Observe the result and see if it is as expected.

29. Describe how you can use this in a Q-number system. What else do you need?

I still need _____

a.) display 

b.) UP button

Your Q-number is
3208

c.) ticket printer + "counter" + user button

Producing a tone on a buzzer

Tone on buzzer

30. Replace *Count7SegSw.c* with *Buzzone.c*.

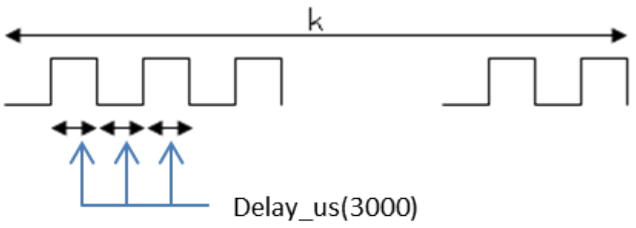
31. Study the code and describe what this program will do:

_____

32. Note that in *Buzzone.c* (under the function *onetone*), the variable is used: *k*.

    The "for" loop i.e. *k* determines the duration of the buzzing, while the delay_us() determines the pitch of the buzzing.

```
void onetone(void) //Function to generate one tone
{
   unsigned int k;

    for (k = 0; k < 100; k++) //Determines duration of tone
    {
        delay_us(3000);  // useable values from 100  to 5000
        PORTCbits.RC0 = !PORTCbits.RC0; //Invert logic level at RC0
    }
}
```



33. Build, download and execute the program. Observe the result and see if it is as expected.

Different tone

34. Modify the program by adding another function named *twotone* with a different *value* in *delay_us(value)*. Include *twotone* in the *main* program and test out the sound effect, as follows:

```
while (1)
{
        Onetone ();
        PORTD = 0b10101010; // pattern on LEDs
        delay_ms(500);
        Twotone ();
        PORTD = 0b01010101; // another pattern on LEDs
        delay_ms(500);

        while (1); // loop forever to stop music!
}
```

35. Debug until the program can work.

```
// Single7Seg.c
// Program to test 1 7-segment display

#include <xc.h>

void main(void)
{
      ADCON1 = 0x0F;

      TRISB=0b11110000; //RB3 to RB0 are connected DIG3 to DIG0
                        //RB5 is connected to a switch

      TRISD=0b00000000; //RD7 to RD0 are connected to segment LEDs


      while(1)                //repeat
      {

            PORTB = 0b00000001;     //enable DIG0

            PORTD = 0b00111111;     //display 0

      }
}
```

```
/*
 * File:   Four7seg.c
 *
 * Created on 13 January, 2016, 1:52 PM
*/


#include <xc.h>
#include "delays.h"
void main(void)
{

      TRISB=0b11110000; //RB3 to RB0 are connected DIG3 to DIG0
                        //RB5 is connected to a switch

      TRISD=0b00000000; //RD7 to RD0 are connected to segment LEDs


      while(1)                      //repeat
      {

            PORTB = 0b00000001;     //enable DIG0
            PORTD = 0b00111111;     //display 0
            delay_ms(1000);         //LEDs on for a while

            PORTB = 0b00000010;     //enable DIG1
            PORTD = 0b00000110;     //display 1
            delay_ms(1000);         //LEDs on for a while

            PORTB = 0b00000100;     //enable DIG2
            PORTD = 0b01011011;     //display 2
            delay_ms(1000);         //LEDs on for a while

            PORTB = 0b00001000;     //enable DIG3
            PORTD = 0b01001111;     //display 3
            delay_ms(1000);         //LEDs on for a while

      }
}
```

```
// Count7SegSw.c
// Counting on 4 7-segment display by a switch on 7-seg Board

#include <xc.h>
#include "delays.h"
#include "seg7.h"

unsigned char point, outchar, press;

void main(void) {
    char i;
    TRISB = 0b11110000; //RB3 to RB0 are connected DIG3 to DIG0
                        //RB5 is connected to a switch

    TRISD = 0b00000000; //RD7 to RD0 are connected to segment LEDs

    val[3] = 9; //contents of DIG3
    val[2] = 8; //contents of DIG2
    val[1] = 7; //contents of DIG1
    val[0] = 6; //contents of DIG0
    press = 0;

    while (1) //repeat
    {

       point = 0b00000001; //enable DIG0

        for (i = 0; i < 4; i++)

        {
            PORTB = point; //enable one DIG
            outchar = val[i]; //get one value for the DIG
            PORTD = convert(outchar); //convert to LED code

            point = point << 1; //point to the next DIG
            delay_ms(1);

        }

        if (press == 0) //switch press first time
        {
            if (PORTBbits.RB5 == 0) //if RB5sw is ON
            {
                press = 1; //switch being pressed
                val[0] = val[0] + 1; //increase DIG0 value
                update(); //adjust the rest of values

            }
        }

     if (PORTBbits.RB5 == 1) press = 0; //switch released
    }
}
```

```
// file : seg7.h

unsigned char val[4]; // variable used

extern void update(void) ; // update the above variable

extern char convert(char outchar); // converts the outchar to 7 segment
                                   //display pattern


/*
 * File:seg7_utilities.c

 * Created on 14 January, 2016, 7:59 PM
*/

#include <xc.h>

extern unsigned char val[4];

char convert(char digit)
{
      char leddata;

       if(digit==0)leddata=0b00111111;
       if(digit==1)leddata=0b00000110;
       if(digit==2)leddata=0b01011011;
       if(digit==3)leddata=0b01001111;
       if(digit==4)leddata=0b01100110;
       if(digit==5)leddata=0b01101101;
       if(digit==6)leddata=0b01111101;
       if(digit==7)leddata=0b00000111;
       if(digit==8)leddata=0b01111111;
       if(digit==9)leddata=0b01101111;
      return(leddata);
}


void update(void)              //Function to adjust DIG values
{
       if(val[0]>=10)
       {
             val[1]=val[1]+1;
             val[0]=0;
       }

       if(val[1]>=10)
       {
             val[2]=val[2]+1;
             val[1]=0;
       }
       if(val[2]>=10)
       {
             val[3]=val[3]+1;
             val[2]=0;
       }
       if(val[3]>=10)
       {
             val[3]=0;
       }
}
```

```
// BuzzOne.c
// Program to activate buzzer with one tone
// For project using USB interface with Bootloader

#include <xc.h>
#include "delays.h"

void onetone(void) //Function to generate one tone
{
   unsigned int k;

    for (k = 0; k < 100; k++) //Determines duration of tone
    {
        delay_us(3000);  // useable values from 100  to 5000
        PORTCbits.RC0 = !PORTCbits.RC0; //Invert logic level at RC0
    }
}

void main(void) {

    TRISCbits.TRISC0 = 0; //-- Set RC0 as output

    TRISD = 0x00; //-- Set all pins on PortD as output

    {
        onetone(); //sound ON then OFF
        PORTD = 0b10101010; //pattern on LEDs
        delay_ms(500);

        onetone(); //sound ON then OFF
        PORTD = 0b01010101; //another pattern on LEDs
        delay_ms(500);

        while (1); // loop forever to stop music!
    }
}
```

Lab 4 – Interfacing to keypad and LCD

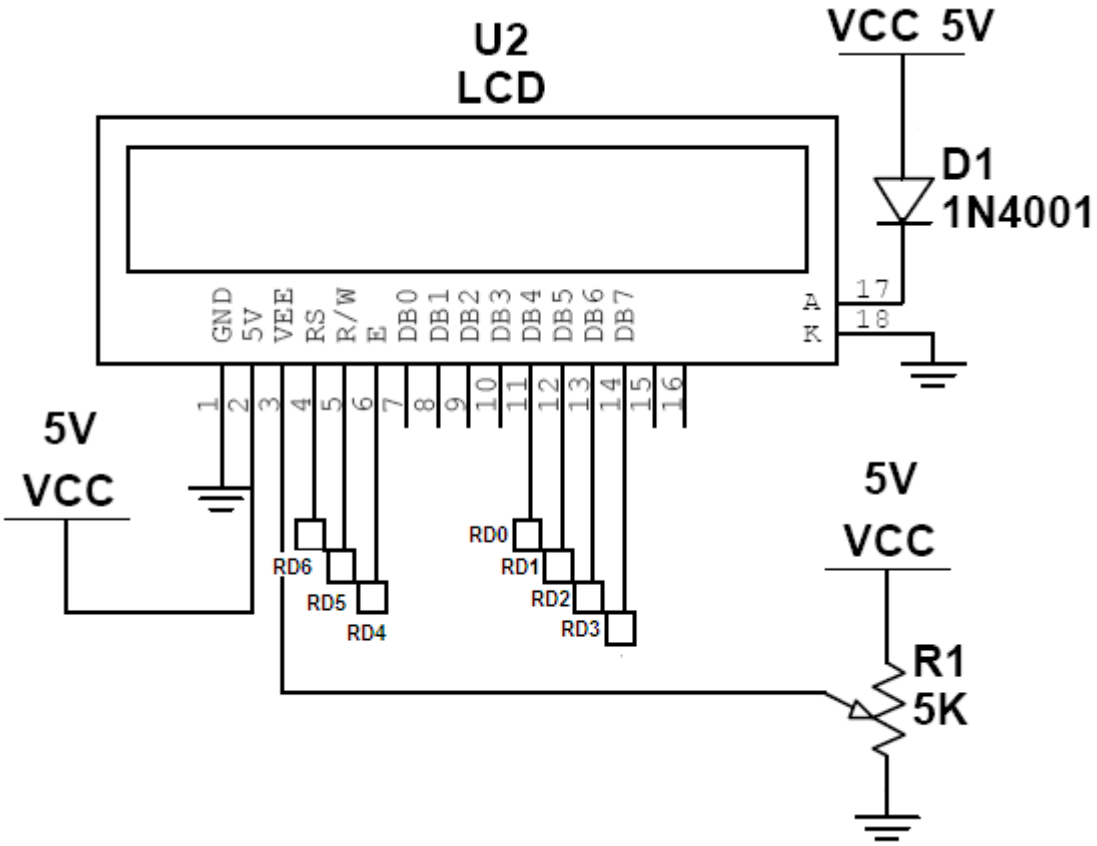
Objectives

- ☐ To learn to display an alphanumeric string on an LCD.
- ☐ To learn to read an input from a 4x4 keypad (using a 74922 keypad encoder).

Introduction / Briefing

LCD at Port D

- ☐ In this experiment, you will be displaying an alphanumeric string (numbers and characters) on an LCD connected to Port D. The LCD can display 2 lines of 16 or 20 characters.
- ☐ Examine the connection below. Other than the power supply pins, you should be able to locate the pins VEE, RS, R/W, E, DB7-0.



☐ The connections & purpose of the pins are shown below:

LCD pin	Connection	Remark / purpose
VEE	Variable resis	For contrast control.
RS	RD6	Register Select. Set RS = 0 to send "command" to LCD. Set RS = 1 to send "data" to LCD.
R/W	RD5	Set R/W = 0 to write to LCD. Set R/W = 1 to read from LCD.
E	RD4	Enable. Apply a falling edge (high to low transition) at E for LCD to latch on data / command at DB pins.
DB7-4	RD3-0	Use only DB7-4 in 4-bit mode, in which a byte of data/command is written as 2 nibbles.
DB3-0	Not connected	Use DB7-0 in 8-bit mode, in which a full 8-bit byte is written in one go.

An example of "command" is 0x01, which will clear the display.
An example of "data" is 0x41 – the character "A" to display on the LCD.

☐ To make it easier for you to use the LCD, 4 functions have been written, based on the table above and the "commands for LCD module" on the next page. **You don't really have to understand the "fine prints" below or the table on the next page**.

void **lcd_write_cmd** (signed char cmd)	▪ A function for writing a command byte to the LCD in 4 bit mode. ▪ *If you look at the code, you will notice that RS is set to 0, and the command byte sent out as two nibbles.*
void **lcd_write_data** (char data)	▪ A function for writing a data byte to the LCD in 4 bit mode. ▪ *If you look at the code, you will notice that RS is set to 1, and the command byte sent out as two nibbles.*
*void **lcd_strobe** (void)*	▪ *A function for generating the strobe signal, i.e. a high to low transition at the Enable (E) pin.*
void **lcd_init** (void)	▪ A function for initialising the LCD. ▪ *The code configures Port D as an output port and set R/W to 0, so that data/command can be written to the LCD.* ▪ *The command lcd_write_cmd(0x28) or 0b001010xx puts the LCD into the 4-bit, 2 lines, 5x7 dots mode.* ▪ *The command lcd_write_cmd(0x0E) or 0b00001110 turns the display & cursor on.* ▪ *The command lcd_write_cmd(0x06) or 0b00000110 causes the cursor position to be incremented after every char.* ▪ *The command lcd_write_cmd(0x01) clears the display and returns the cursor to the home position.*

COMMANDS FOR LCD MODULE

Command	Code										Description	Execution Time
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
Clear Display	0	0	0	0	0	0	0	0	0	1	Clears the display and returns the cursor to the home position (address 0).	82µs~1.64ms
Return Home	0	0	0	0	0	0	0	0	1	*	Returns the cursor to the home position (address 0). Also returns a shifted display to the home position. DD RAM contents remain unchanged.	40µs~1.64ms
Entry Mode Set	0	0	0	0	0	0	0	1	I/D	S	Sets the cursor move direction and enables/disables the display.	40µs
Display ON/OFF Control	0	0	0	0	0	0	1	D	C	B	Turns the display ON/OFF (D), or the cursor ON/OFF (C), and blink of the character at the cursor position (B).	40µs
Cursor & Display Shift	0	0	0	0	0	1	0	*	*		Moves the cursor and shifts the display without changing the DD RAM contents.	40µs
Function Set	0	0	0	0	1	DL	N$	RE	*	#	Sets the data width (DL), the number of lines in the display (L), and the character font (F).	40µs
Set CG RAM Address	0	0	0	1	A_{CG}						Sets the CG RAM address. CG RAM data can be read or altered after making this setting.	40µs
Set DD RAM Address	0	0	1	A_{DD}							Sets the DD RAM address. Data may be written or read after making this setting.	40µs
Read Busy Flag & Address	0	1	BF	AC							Reads the BUSY flag (BF) indicating that an internal operation is being performed and reads the address counter contents.	1µs
Write Data to CG or DD RAM	1	0	Write Data								Writes data into DD RAM or CG RAM.	46µs
Read Data from CG or DD RAM	1	1	Read Data								Reads data from DD RAM or CG RAM.	46µs
	I/D = 1: Increment I/D = 0: Decrement S = 1: Accompanies display shift. S/C= 1: Display shift S/C = 0: cursor move R/L= 1: Shift to the right. R/L= 0: Shift to the left. DL = 1: 8 bits DL = 0: 4 bits N = 1: 2 lines N = 0: 1 line RE = 1: Ext. Reg. Ena. F = 0: 5 x 7 dots BF = 1: Busy BF = 0: Can accept data *# Set to 1 on 24x4 modules* *$ With KS0072 is Address Mode.*										DD RAM: Display data RAM CG RAM: Character generator RAM A_{CG}: CG RAM Address A_{DD}: DD RAM Address Corresponds to cursor address. AC: Address counter Used for both DD and CG RAM address.	Execution times are typical. If transfers are timed by software and the busy flag is not used, add 10% to the above times.

☐ There is no need to start from scratch when you need to use LCD. You can modify an existing "main" function to suit your new application.

☐ In a typical "main" function (e.g. that of LCD2Lines.c below)

- The LCD is first initialised using **LCD_init()**.
- Then, the cursor is move to the desired position
  - **lcd_write_cmd(0x80)** moves it to line 1 position 1 while
  - **lcd_write_cmd(0xC0)** moves it to line 2 position 1.
- The command **lcd_write_data (0x41)** write the letter "A" to the current cursor position etc.

```
// LCD2Lines.c
void main(void)
{
 lcd_init();                    // Initialise LCD module

 while(1)
 {
  lcd_write_cmd(0x80);          // Move cursor to line 1 position 1
  lcd_write_data(0x41);         // write "A" to LCD
…..
```

Binary patterns for different characters

LOWER 4 BITS \ UPPER 4 BITS	0000	0010	0011	0100	0101	0110	0111	1010	1011	1100	1101	1110	1111
0000	CG RAM (1)		0	@	P	`	p		—	タ	ミ	α	p
0001	(2)	!	1	A	Q	a	q	。	ア	チ	ム	ä	q
0010	(3)	"	2	B	R	b	r	「	イ	ツ	メ	ß	θ
0011	(4)	#	3	C	S	c	s	」	ウ	テ	モ	ε	∞
0100	(5)	$	4	D	T	d	t	、	エ	ト	ヤ	μ	Ω
0101	(6)	%	5	E	U	e	u	・	オ	ナ	ユ	σ	ü
0110	(7)	&	6	F	V	f	v	ヲ	カ	ニ	ヨ	ρ	Σ
0111	(8)	'	7	G	W	g	w	ァ	キ	ヌ	ラ	q	π
1000	(1)	(	8	H	X	h	x	ィ	ク	ネ	リ	√	x̄
1001	(2)	)	9	I	Y	i	y	ゥ	ケ	ノ	ル	-1	y
1010	(3)	*	:	J	Z	j	z	ェ	コ	ハ	レ	j	千
1011	(4)	+	;	K	[	k	{	ォ	サ	ヒ	ロ	x	万
1100	(5)	,	<	L	¥	l	|	ャ	シ	フ	ワ	¢	円
1101	(6)	-	=	M	]	m	}	ュ	ス	ヘ	ン	£	÷
1110	(7)	.	>	N	^	n	→	ョ	セ	ホ	゛	ñ	
1111	(8)	/	?	O	_	o	←	ッ	ソ	マ	°	ö	■

Q1: Fill in the blanks below to show how you can display "HELLO" on the first line of the LCD, and "WORLD" on the second line.

```
// Hello World.c
void main(void)
{
  _________________              // Initialise LCD module

  while(1)
  {
    _________________            // Move cursor to line 1 position 1
    lcd_write_data(0x_____);     // write "H" to LCD
    lcd_write_data(0x_____);     // write "E" to LCD
    lcd_write_data(0x_____);     // write "L" to LCD
    lcd_write_data(0x_____);     // write "L" to LCD
    lcd_write_data(0x_____);     // write "O" to LCD

    _________________            // Move cursor to line 2 position 1
    lcd_write_data(0x_____);     // write "W" to LCD
    lcd_write_data(0x_____);     // write "O" to LCD
    lcd_write_data(0x_____);     // write "R" to LCD
    lcd_write_data(0x_____);     // write "L" to LCD
    lcd_write_data(0x_____);     // write "D" to LCD
    while(1);                    //stop here for now
  } // while
} // main
```

Q2: What do you think is achieved by the code below?

Your answer: _____

```
unsigned char K, outchar;
char Message [ ] = "Enter PIN number :  ";    // Defining a 20 char string

void main(void)
{
   lcd_init();                          // Initialise LCD module

   while(1)
   {
      lcd_write_cmd(0x80);              // Move cursor to line 1 position 1
      for (K = 0; K < 20; K++)          // for 20 char LCD module
      {
         outchar = Message[ K ];
         lcd_write_data(outchar);       // write character data to LCD
      }
...
```

Q3: What changes do you need to make to display "Welcome to SP"?

Your answer: _____

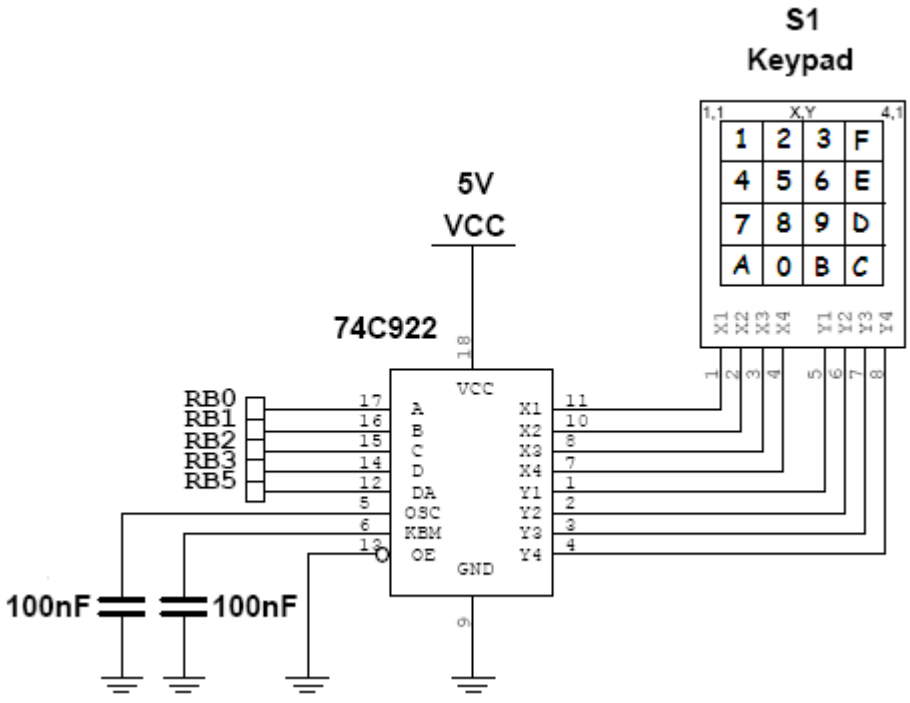
Keypad at Port B

☐ In the second part of this experiment, you will be reading from a 4x4 keypad (with encoder) connected to Port B (pins 0, 1, 2, 3 and 5).

☐ Examine the connection below. See how the 16 keys are labelled. The columns are numbered X1, X2, X3, X4 (from left to right) while the rows are numbered Y1, Y2, Y3, Y4 (from top to bottom). So, the key 2 is X2, Y1 while the key B is X3, Y4.

Q4: Which key corresponds to X2, Y3?

Your answer: _____



☐ These 8 signals (X's and Y's) from the keypad are connected to a **keypad encoder 74C922** which has the **truth table** below. As a result of "encoding", 8 bits become 5 bits.

Keypad Encoder truth table

	Keys															
	X1 Y1	**X2 Y1**	**X3 Y1**	**X4 Y1**	**X1 Y2**	**X2 Y2**	**X3 Y2**	**X4 Y2**	**X1 Y3**	**X2 Y3**	**X3 Y3**	**X4 Y3**	**X1 Y4**	**X2 Y4**	**X3 Y4**	**X4 Y4**
D	0	**0**	0	0	0	0	0	0	1	1	1	1	1	1	1	1
C	0	**0**	0	0	1	1	1	1	0	0	0	0	1	1	1	1
B	0	**0**	1	1	0	0	1	1	0	0	1	1	0	0	1	1
A	0	**1**	0	1	0	1	0	1	0	1	0	1	0	1	0	1
DA	1	**1**	1	1	1	1	1	1	1	1	1	1	1	1	1	1

☐ D (msb), C, B, A identify the key pressed. For instance, if key '2' is pressed, X2, Y1 cause DCBA = 0001. *[Note: This does not tell the key pressed is 2, as binary 0001 is not exactly decimal 2. Further interpretation is needed – see C code below.]* The DA (**data available**) signal will be set to logic '1', whenever a key is pressed.

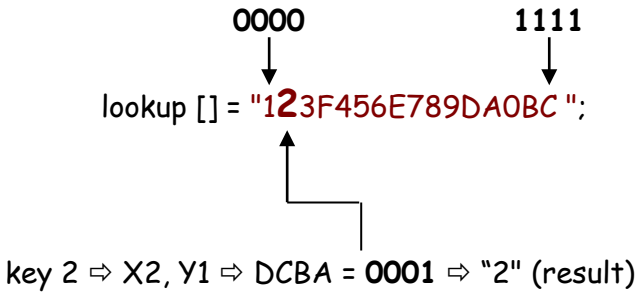
Q5: What happen if key '6' is pressed?

Your answer: key '6' is X__ & Y__. So, DCBA = _____, DA = _____

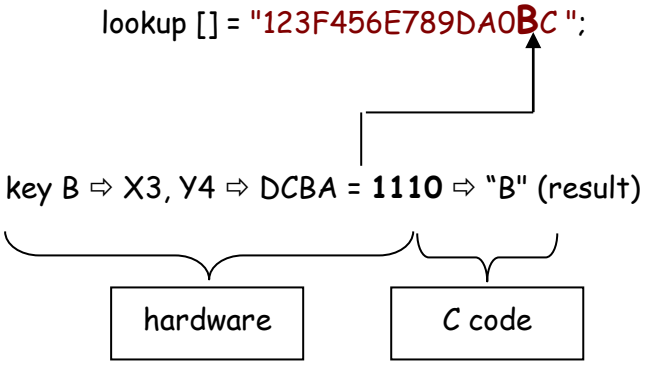
☐ The function to read & interpret the key is this:

```
#define KEY_DA   PORTBbits.RB5  // 74922 DA output
#define KEY_PORT   PORTB  // RB3 to RB0 has keypad data

char getkey (void)
{   char keycode;
    const unsigned char lookup[] = "123F456E789DA0BC ";
    while (KEY_DA==0);  // wait for key to be pressed
    keycode=KEY_PORT & 0x0F;  // read from encoder at portB,
                              // mask off upper 4 bits

    while (KEY_DA==1);  // wait for key to be released
    return( lookup [keycode] ); // look up table to find
                                // key pressed
}
```

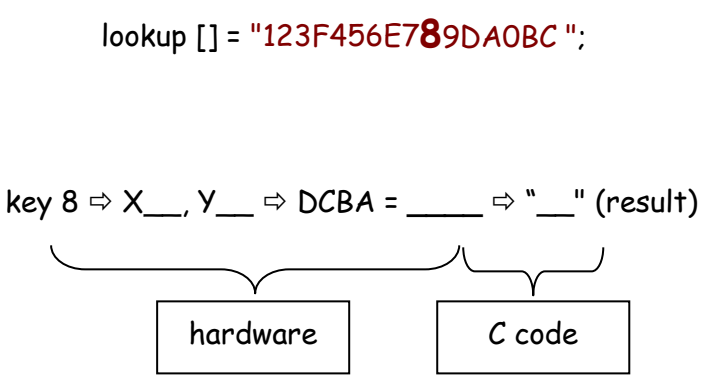
☐ The function waits for DA to become 1 i.e. a key pressed. Then it reads from Port B and mask off the top 4 bits, i.e. only RB3 to RB0 (connected to the signals D, C, B and A) are retained. After that, it waits for the key to be released. Finally, it returns the key pressed by looking up the look-up-table.



☐ Likewise, if key B has been pressed, you get this



Q6: Try key 8…



Q7: Assuming the hardware connections are unchanged, but the 4x4 keypad has been labelled differently, as follows:

What changes to the look up table is necessary for correct interpretation?

Your answer: lookup [] = "1___4____7____*____ ";

Q8. You will come across the following code in the last part of the experiment.

```
lcd_write_cmd(0xC0); // Move cursor to line 2 position 1

for ( i = 0; i < 20; i++) // for 20 number
{
 key=getkey(); // use "getkey" function to read/interpret key pressed
 lcd_write_data(key); // display on LCD
}
```

Describe what happens when the code is executed:

Your answer: _____

Activites:

Before you begin, ensure that the Micro-controller Board is connected to the LCD / Keypad Board.

Displaying an alphanumeric string on LCD

1. Launch the *MPLABX IDE* and create a new project called *Lab4*.

2. Add the file *LCD2Lines.c* to the *Lab4* project *Source File* folder.
Make sure *Copy* is ticked before you click *Select*. If you have forgotten the steps, you will need to refer to the previous lab sheet.

   Note that the program uses the functions *lcd_init (), lcd_write_cmd (), lcd_write_data ()* from *lcd_utilities.c* and contains *#include "lcd.h"*. The files *lcd.h* and *lcd_utilities.c* need to be added to the Project.

Display msg on LCD

3. Study the code (the main function) and describe what this program will do:

   _____

4. Build, download and execute the program. Observe the result and see if it is as expected.

Changing the msg on LCD

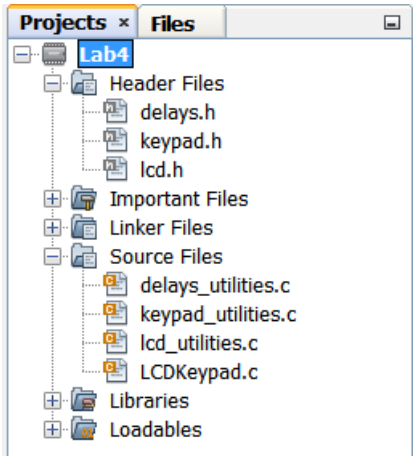
5. Modify the code to show the following on the LCD. Build, download and execute the program to verify your coding.

JOHN 9123456

Reading inputs from keypad and displaying them on LCD

Keypad entries on LCD

6. Replace *LCD2Lines.c* with *LCDKeypad.c*. The files *keypad.h* and *keypad_utilities.c* (which contains the *getkey ()* function) need to be added to the Project. Likewise, the files *delays.h* and *delays_utilities.c* need to be added.



7. Study the code and describe what this program will do:

_____

8. Build, download and execute the program. Observe the result and see if it is as expected.

4-key PIN no. on LCD

9. Modify the code to accept a 4-key PIN number (-- see Hint below). Build, download and execute the program to verify your coding.

Hint:

unsigned char P1, P2, P3, P4; // variables to store a copy of the PIN number
// entered. Put this BEFORE any executable code in main

// put the following after the code to move cursor to line 2 position 1,
// replacing the 2nd for loop
key = getkey(); // get the first key
P1 = key; // save first key in P1
lcd_write_data(key); // display on LCD

key = getkey(); // get the second key
P2 = key; // save second key in P2
lcd_write_data(key); // display on LCD
…

while(1); // add this to prevent program from restarting after 4th key

Hiding the PIN number

10. Further modify the code so that it will not display the actual PIN number entered. Instead, * will be shown after each key.

    Enter PIN number: * * * _

> Optional:
>
> After 4 keys have been entered, the program can show the message:
> "Processing……"
>
> Hint: You need to have a char array: Message2 [] = " Processing….." and a loop to display this message.

11. Build, download and execute the program to verify your coding. Debug until the program can work.

Password protected" access

Password protected door

12. Replace *LCDKeypad.c* with *LCDKeypadPwd.c*.

13. This program will accept a 4-key password (or "PIN number"). If the correct password is entered, the LCD will display "*OPEN*". Otherwise, the LCD will display "*WRONG*"

14. What do you think is the password?

    Your answer: _____

15. Build, download and execute the program. Observe the result and see if it is as expected.

```
// LCD2Lines.c
// Program to test LCD.
// The LCD display with two lines, 24 characters each.
// There are three control lines (RD4:RD6) and four data lines(RD3:RD0).
// RD6 - RS=0 Data represents Command, RS=1 Data represents Character
// RD5 - RW=0 Writing into the LCD module
// RD4 - E =1 Data is latched into LCD module during low to high transition


#include <xc.h>
#include "lcd.h"  // Include file is located in the project directory

void main(void)
{
      lcd_init();                   // Initialise LCD module

      while(1)
      {
      lcd_write_cmd(0x80);          // Move cursor to line 1 position 1
      lcd_write_data(0x41);         // write "A" to LCD
      lcd_write_data(0x42);         // write "B" to LCD
      lcd_write_data(0x43);         // write "C" to LCD

      lcd_write_cmd(0xC0);          // Move cursor to line 2 position 1
      lcd_write_data(0x31);         // write "1" to LCD
      lcd_write_data(0x32);         // write "2" to LCD
      lcd_write_data(0x33);         // write "3" to LCD

      while(1);                     //stop here for now

      }
}
```

// LCDKeypad.c

```
// Program to test LCD and keypad.
// For project using USB interface with Bootloader

#include "lcd.h"
#include <xc.h>
#include "keypad.h"
#include "delays.h"
unsigned char key,outchar;
char Message1 [ ] = "Enter PIN number :  ";        // Defining a 20 char string

// ---- Main Program -----------------------------------------------------------
void main(void)
{
        int i;
        lcd_init();                                 // Initialise LCD module

        while(1)
        {
                lcd_write_cmd(0x80);                // Move cursor to line 1 position 1
                for (i = 0; i < 20; i++)            //for 20 char LCD module
                {
                        outchar = Message1[i];
                        lcd_write_data(outchar); // write character data to LCD
                }


                lcd_write_cmd(0xC0);                // Move cursor to line 2 position 1
                for (i = 0; i < 20; i++)            //for 20 number
                {
                    key=getkey();                   // waits and get ascii key number when pressed
                    lcd_write_data(key);            //display on LCD

                }

                delay_ms(1000);                     // wait 1 second
                lcd_write_cmd(0x01);                // 00000001 Clear Display instruction
        }
}
```

// LCDKeypadPwd.c

```
// Program to test LCD and keypad.
// For project using USB interface with Bootloader

#include <xc.h>
#include "lcd.h"
#include "delays.h"
#include "keypad.h"

unsigned char key, outchar;
unsigned char p1, p2, p3, p4;
char Message1 [ ] = "Enter PIN number :  "; // Defining a 20 char string



void main(void) {
int i;
lcd_init(); // Initialise LCD module
```

```
while (1) {

        lcd_write_cmd(0x80); // Move cursor to line 1 osition 1
       for (i = 0; i < 20; i++)
       {
            outchar = Message1[i];
            lcd_write_data(outchar); // write character data to LCD
       }

       lcd_write_cmd(0xC0); // Move cursor to line 2 position 1

       key = getkey(); // waits and get an ascii key number when
       p1 = key;
       lcd_write_data(key); //display on LCD

       key = getkey(); // waits and get an ascii key number when pressed
       p2 = key;
       lcd_write_data(key); //display on LCD

       key = getkey(); // waits and get an ascii key number when pressed
       p3 = key;
       lcd_write_data(key); //display on LCD

       key = getkey(); // waits and get an ascii key number when pressed
       p4 = key;
       lcd_write_data(key); //display on LCD

       if (p1 == '4' && p2 == '5' && p3 == '5' && p4 == '0')
       {
            lcd_write_data(0x20);
            lcd_write_data('O');
            lcd_write_data('P');
            lcd_write_data('E');
            lcd_write_data('N');
            lcd_write_data(0x20);
       }
       else
       {
            lcd_write_data(0x20);
            lcd_write_data('W');
            lcd_write_data('R');
            lcd_write_data('O');
            lcd_write_data('N');
            lcd_write_data('G');
       }

    }
}
```

```
/*  file : lcd.h
 *     LCD interface header file
 *     See lcd.c for more info
 */

 /* intialize the LCD - call before anything else */
extern void lcd_init(void);

/* write a byte to the LCD in 4 bit mode */
extern void lcd_write_cmd(unsigned char cmd);

//extern void lcd_write(unsigned char i);
extern void lcd_write_data(char data);



/*
 * File:   lcd utilities.c
 *
 * Created on 13 January, 2016, 10:28 AM
*/

//#include "LCD.H"  // Include file is located in the project directory

#include <xc.h>
#define _XTAL_FREQ 48000000
#define LCD_RS PORTDbits.RD6    //  Register Select on LCD
#define LCD_EN PORTDbits.RD4    //  Enable on LCD controller
#define LCD_WR PORTDbits.RD5    //  Write on LCD controller
void lcd_strobe(void);

//--- Function for writing a command byte to the LCD in 4 bit mode -------------

void lcd_write_cmd(unsigned char cmd)
{
  unsigned char temp2;
  LCD_RS = 0;                     // Select LCD for command mode
  __delay_ms(4);                  // 40us delay for LCD to settle down

  temp2 = cmd;
  temp2 = temp2 >> 4;       // Output upper 4 bits, by shifting out lower 4 bits
  PORTD = temp2 & 0x0F;           // Output to PORTD which is connected to LCD

   __delay_ms(8);                 // 10ms - Delay at least 1 ms before strobing
  lcd_strobe();
  __delay_ms(8);                  // 10ms - Delay at least 1 ms after strobing

 temp2 = cmd;                     // Re-initialise temp2
 PORTD = temp2 & 0x0F;            // Mask out upper 4 bits

 __delay_ms(8);                   // 10ms - Delay at least 1 ms before strobing
 lcd_strobe();
 __delay_ms(8);                   // 10ms - Delay at least 1 ms before strobing

}
```

```
//---- Function to write a character data to the LCD ---------------------------

void lcd_write_data(char data)
{
  char temp1;

  LCD_RS = 1;                       // Select LCD for data mode
  __delay_ms(4);                    // 40us delay for LCD to settle down

  temp1 = data;
  temp1 = temp1 >> 4;
  PORTD = temp1 & 0x0F;

  __delay_ms(8);                    //_-_ strobe data in
  lcd_strobe();
  __delay_ms(8);

  temp1 = data;
  PORTD = temp1 & 0x0F;

  __delay_ms(10);                   //_-_ strobe data in
  lcd_strobe();
  __delay_ms(10);
}


//-- Function to generate the strobe signal for command and character----------

void lcd_strobe(void)               // Generate the E pulse
{
  LCD_EN = 1;                       // E = 1
  __delay_ms(8);                    // 10ms delay for LCD_EN to
  LCD_EN = 0;                       settle // E = 0
  __delay_ms(8);                    // 10ms delay for LCD_EN to settle
}
```

```
//---- Function to initialise LCD module -------------------------------------
void lcd_init(void)
{
  int i;
  TRISD = 0x00;
  PORTD = 0x00;                  // PORTD is connected to LCD data pin
  LCD_EN = 0;
  LCD_RS = 0;                    // Select LCD for command mode
  LCD_WR = 0;                    // Select LCD for write mode

  for(i=0;i<100;i++)
    __delay_ms(10);              // Delay a total of 1 s for LCD module to
                                 // finish its own internal initialisation

/* The datasheets warn that LCD module may fail to initialise properly when
power is first applied. This is particularly likely if the Vdd

supply does not rise to its correct operating voltage quickly enough.

It is recommended that after power is applied, a command sequence of
3 bytes of 30h be sent to the module. This will ensure that the module is
in 8-bit mode and is properly initialised. Following this, the LCD module
can be switched to 4-bit mode.
*/

 lcd_write_cmd(0x33);
 lcd_write_cmd(0x32);
 lcd_write_cmd(0x28);       // 001010xx - Function Set instruction
                            // DL=0 :4-bit interface,N=1 :2 lines,F=0 :5x7 dots

 lcd_write_cmd(0x0E);       // 00001110 - Display On/Off Control instruction
                            // D=1 :Display on,C=1 :Cursor on,B=0 :Cursor Blink on

 lcd_write_cmd(0x06);       // 00000110 - Entry Mode Set instruction
                            // I/D=1 :Increment Cursor position
                            // S=0 : No display shift

 lcd_write_cmd(0x01);       // 00000001 Clear Display instruction

 __delay_ms(10);            // 10 ms delay

}
```

```
// file : keypad.h

extern char getkey(void); // waits for a keypress and returns the ascii code


/*
 * File:   keypad utilities.c
 *
 * Created on 13 January, 2016, 10:46 AM
*/


#include <xc.h>


#define KEY_DA PORTBbits.RB5   //  74922 DA output
#define KEY_PORT PORTB           //     RB3 to RB0 has keypad data


//----- Function to obtained wait for key press and returns its ASCII value

char getkey(void){
   char keycode;
   const unsigned char lookup[] = "123F456E789DA0BC ";

   while (KEY_DA==0);              //wait for key to be pressed
   keycode=KEY_PORT & 0x0F;        //read from encoder at portB,mask upper 4 bits

   while (KEY_DA==1);              //wait for key to be released
   return(lookup[keycode]);        //convert keycode to its ascii value for LCD
}
```

Lab 5 – Analogue to digital converter and interfacing high power devices

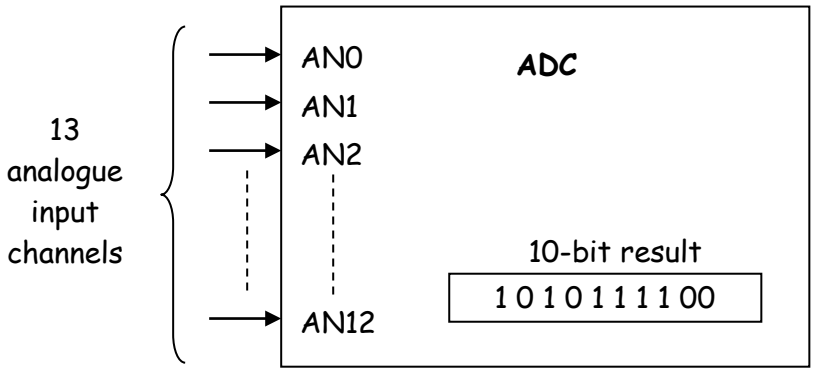
Objectives

- ☐ To learn to use the ADC (Analogue to Digital Converter) in the PIC18F4550 microcontroller.
- ☐ To learn to turn some high power devices (e.g. motor, relay) on and off.

Introduction / Briefing

PIC18F4550's ADC

- ☐ The PIC18F4550 microcontroller has a built-in ADC module capable of converting up to 13 analogue inputs to their corresponding 10-bit digital representations.



- ☐ The analogue input pins AN0 to AN12 could be connected to a variety of sensors for monitoring the environment.
- ☐ For instance, a LDR (Light Dependent Resistor) circuit can be used to sense the ambient brightness. The result is an analogue voltage, which can be converted to a digital equivalent, and then used by the PIC to make a decision – if it is too dark, switch on the lights. Other sensors include sensors for temperature, water level, humidity, PH value etc.
- ☐ In this experiment, a variable resistor is used to give a variable voltage input and the result of AD conversion will be shown on an L E D bar (- a low power output device). Later, you will work on a fish tank "water level monitoring" application – If the water level is too high, turn on the motor (- a high power output device) to pump away the excess water. If the water level is too low, turn on the alarm (- another high power output device) to alert the owner.
- ☐ First, the registers associated with PIC18F4550's ADC:

ADCON0 – This register controls the **operation of the A/D module**.

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	CHS3	CHS2	CHS1	CHS0	GO/$\overline{\text{DONE}}$	ADON
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 7-6 **Unimplemented:** Read as 'o'

bit 5-2 **CHS3:CHS0:** Analog Channel Select bits

0000 = Channel 0 (AN0)
0001 = Channel 1 (AN1)
0010 = Channel 2 (AN2)
0011 = Channel 3 (AN3)
0100 = Channel 4 (AN4)
0101 = Channel 5 (AN5)(1,2)
0110 = Channel 6 (AN6)(1,2)
0111 = Channel 7 (AN7)(1,2)
1000 = Channel 8 (AN8)
1001 = Channel 9 (AN9)
1010 = Channel 10 (AN10)
1011 = Channel 11 (AN11)
1100 = Channel 12 (AN12)
1101 = Unimplemented(2)
1110 = Unimplemented(2)
1111 = Unimplemented(2)

bit 1 **GO/$\overline{\text{DONE}}$:** A/D Conversion Status bit

When ADON = 1:
1 = A/D conversion in progress
0 = A/D Idle

bit 0 **ADON:** A/D On bit

1 = A/D converter module is enabled
0 = A/D converter module is disabled

Q1: What binary pattern must be written to ADCON0 to select Channel 0 for conversion, and to power up the ADC module? (Don't start the conversion yet.)

Your answer: _____

☐ The C code required is ADCON0 = 0b 00 0000 0 1;

☐ Note that to start the conversion, the GO bit must be set to 1. When conversion is finished, the same bit (read as DONE) will be set to 0.

ADCON1 - This register configures the **voltage references** and the **functions of the port pins**.

U-0	U-0	R/W-0	R/W-0	R/W-0(1)	R/W(1)	R/W(1)	R/W(1)
—	—	VCFG0	VCFG0	PCFG3	PCFG2	PCFG1	PCFG0
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 7-6 **Unimplemented:** Read as '0'

bit 5 **VCFG1:** Voltage Reference Configuration bit (VREF- source)
1 = VREF- (AN2)
0 = VSS

bit 4 **VCFG0:** Voltage Reference Configuration bit (VREF+ source)
1 = VREF+ (AN3)
0 = VDD

bit 3-0 **PCFG3:PCFG0:** A/D Port Configuration Control bits:

PCFG3: PCFG0	AN12	AN11	AN10	AN9	AN8	AN7(2)	AN6(2)	AN5(2)	AN4	AN3	AN2	AN1	AN0
0000(1)	A	A	A	A	A	A	A	A	A	A	A	A	A
0001	A	A	A	A	A	A	A	A	A	A	A	A	A
0010	A	A	A	A	A	A	A	A	A	A	A	A	A
0011	D	A	A	A	A	A	A	A	A	A	A	A	A
0100	D	D	A	A	A	A	A	A	A	A	A	A	A
0101	D	D	D	A	A	A	A	A	A	A	A	A	A
0110	D	D	D	D	A	A	A	A	A	A	A	A	A
0111(1)	D	D	D	D	D	A	A	A	A	A	A	A	A
1000	D	D	D	D	D	D	A	A	A	A	A	A	A
1001	D	D	D	D	D	D	D	A	A	A	A	A	A
1010	D	D	D	D	D	D	D	D	A	A	A	A	A
1011	D	D	D	D	D	D	D	D	D	A	A	A	A
1100	D	D	D	D	D	D	D	D	D	D	A	A	A
1101	D	D	D	D	D	D	D	D	D	D	D	A	A
1110	D	D	D	D	D	D	D	D	D	D	D	D	A
1111	D	D	D	D	D	D	D	D	D	D	D	D	D

A = Analog input D = Digital I/O

Q2: What binary pattern must be written to ADCON1 to use VSS (0 Volt) and VDD (5 Volt) as reference voltages, and to make AN2 to AN0 analogue inputs (and the remaining inputs i.e. AN12 to AN3 digital)?

Your answer: _____

☐ The C code required is ADCON1 = 0b 00 00 1100;

ADCON2 - This register configures the **A/D clock source**, **programmed acquisition time** and **justification**.

R/W-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
ADFM	—	ACQT2	ACQT1	ACQT0	ADCS2	ADCS1	ADCS0
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 7 **ADFM:** A/D Result Format Select bit
1 = Right justified
0 = Left justified

bit 6 **Unimplemented:** Read as '0'

bit 5-3 **ACQT2:ACQT0:** A/D Acquisition Time Select bits
111 = 20 TAD
110 = 16 TAD
101 = 12 TAD
100 = 8 TAD
011 = 6 TAD
010 = 4 TAD
001 = 2 TAD
000 = 0 TAD(1)

bit 2-0 **ADCS2:ADCS0:** A/D Conversion Clock Select bits
111 = FRC (clock derived from A/D RC oscillator)(1)
110 = FOSC/64
101 = FOSC/16
100 = FOSC/4
011 = FRC (clock derived from A/D RC oscillator)(1)
010 = FOSC/32
001 = FOSC/8
000 = FOSC/2

Q3: What binary pattern must be written to ADCON2 to select left justified result (*), to set acquisition time of 4 T_{AD}, and to select F_{OSC}/64 as the clock source?

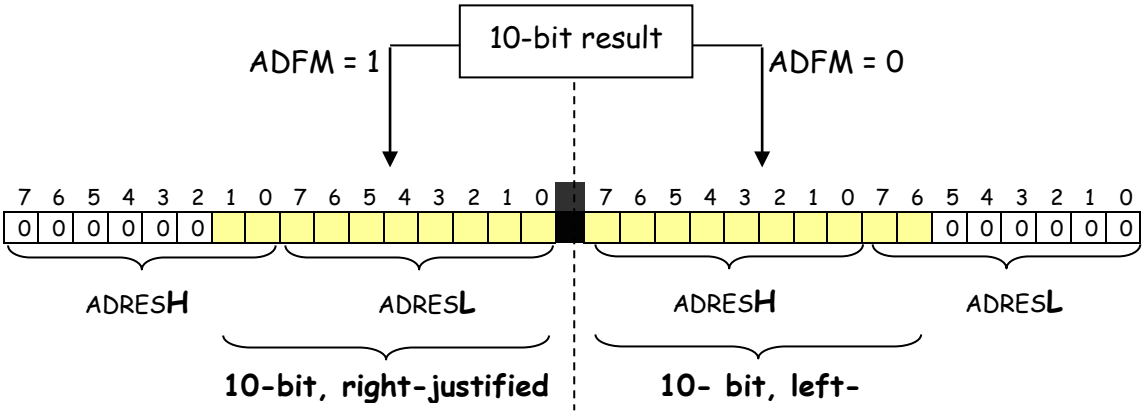
Your answer: _____

☐ The C code required is ADCON2 = 0b 0 0 010 110;

Note: Refer to lecture notes for details of acquisition time & clock source selection.

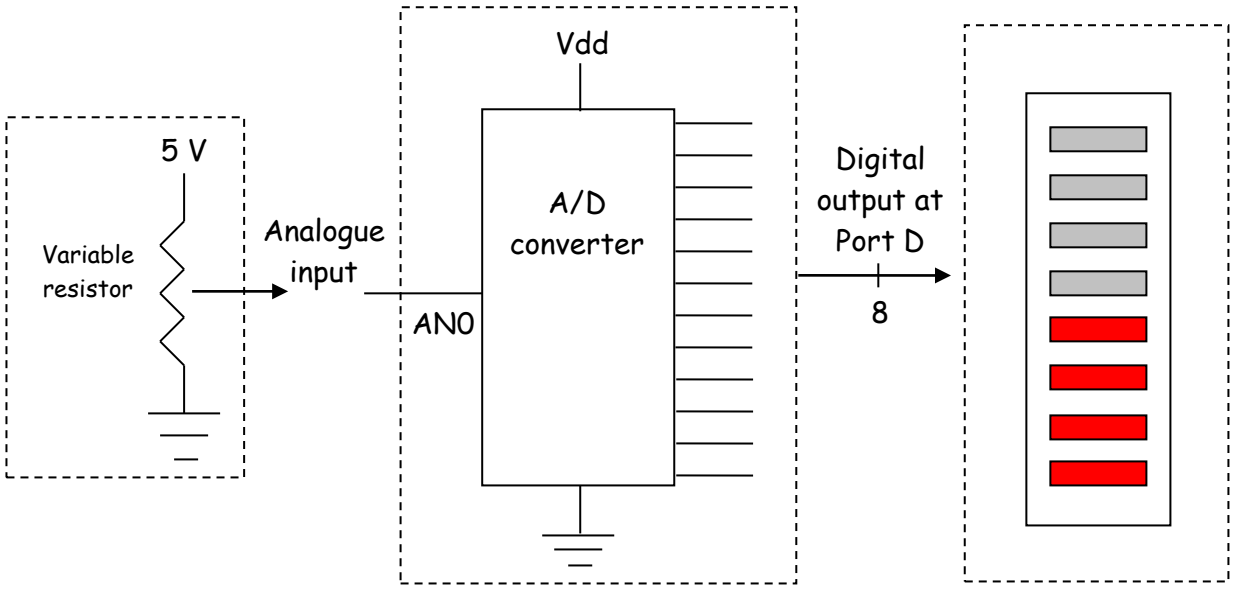
(*) Justification

☐ When an A/D conversion is completed, the 10-bit result is stored in the registers ADRESH and ADRESL, either **left-justified** or **right-justified**, depending on the value of the **ADFM** bit, as shown below. Note that the other bits are filled with 0's.



Example 1 – Showing the result of conversion on an L E D bar

☐ In the circuit below, the variable resistor is used to produce an analogue input – by turning the knob on the variable resistor, the AN0 voltage can be varied. This voltage can then be AD converted into a 10-bit digital equivalent. The most significant 8 bits can then be displayed on the L E D bar.



☐ The complete C code is as follows:

```
void main(void)
{
   TRISD = 0x00;    // Set PORTD to be output
   PORTD = 0x00;   // Initialise PORTD LEDs to zeros

   /* Configuring the ADC */
   ADCON0 = 0b00000001;          // bit5-2  0000 - select channel 0 for conversion
                                 // bit1    A/D conversion status bit
                                 //             1- GO to starts the conversion
                                 //             0 - DONE when A/D is completed
                                 // bit0    Set to 1 to power up A/D

   ADCON1 = 0b00001100;          // bit5    0 -reference is VSS
                                 // bit4    0 - reference is VDD
                                 // bit3-0  1100 - AN2 to AN0 Analog, the rest Digital

   ADCON2 = 0b00 010 110;        // bit7    A/D Result Format:
                                 //             0 - Left justified
                                 //             1 - Right justified
                                 // bit5-3  010 - acquisition time = 4 TAD
                                 // bit2-0  110 - conversion clock = Fosc / 64

   for( ; ; )
   {
      ADCON0bits.GO = 1;         // This is bit2 of ADCON0, START CONVERSION NOW

      while(ADCON0bits.GO == 1); // Waiting for DONE

      PORTD = ADRESH;            / Displaying only the upper 8-bits of the A/D result
.... // continue...
```

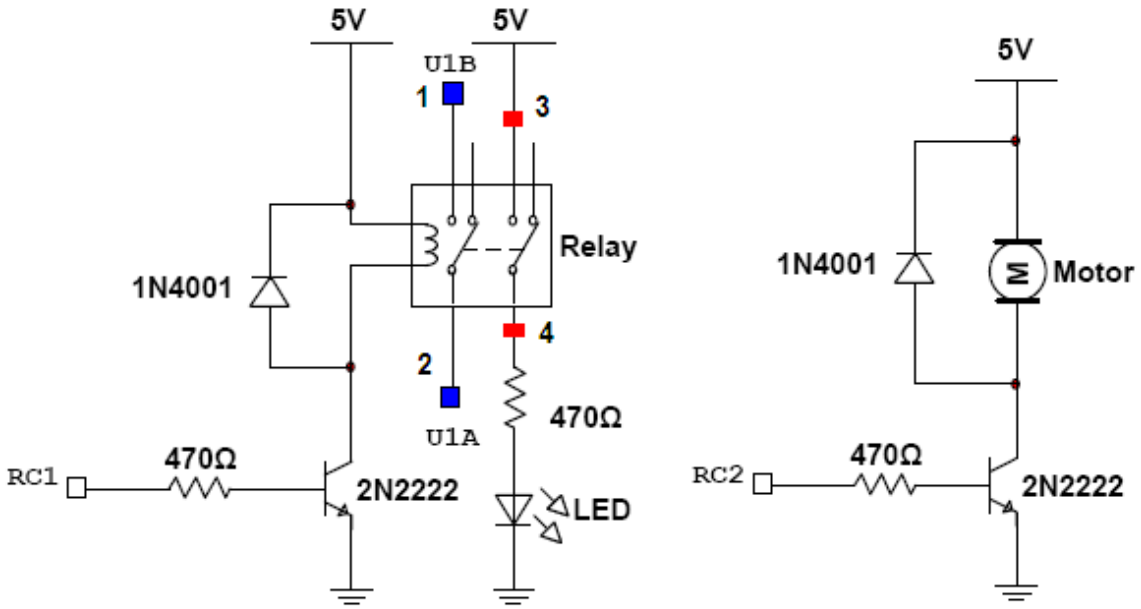
Using transistor as a switch – motor & relay

☐ Low power devices such as LED's can be driven directly by the microcontroller pins. To control a higher power device such as motor / relay, a transistor can be used as a switch as shown below.

☐ For instance, in the motor circuit on the right below, the microcontroller pin RC2 is used to turn the motor on and off.

☐ When RC2 = 1, the transistor (2N2222) is switched on, allowing a current to flow through & turn the motor.

☐ When the transistor is switched off, the diode (1N4001) allows current to continue flowing through the motor, avoiding damage to the motor circuit, especially the transistor.



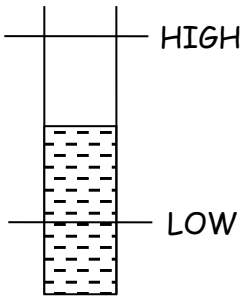
☐ In the relay circuit on the left above, the microcontroller pin RC1 is used to energise the relay.

☐ When RC1 = 1, the transistor is switched on, allowing a current to flow through the coil & the relay is energised. An energised relay closes the contact between points 1 and 2, as well as 3 and 4.

☐ Since 3 & 4 are now in contact, a current can flow through and turn on the LED. Likewise, points 1 & 2 can be used for connecting an alarm or other high power device.

☐ Note that the high power device connected to points 1 & 2 could be powered by a different power source. The relay can thus provide power isolation.

Example 2 – Turning on the pump/motor when water level is high,
turning on the alarm when water level is low



☐ With the above discussion, it will not be difficult for you to build a fish tank "water level monitoring" application. If the water level is too high, the pump motor will be turned on to drain away the excess water. If the water level is too low, an alarm will be activated to alert the owner.

Activites:

Before you begin, ensure that the Micro-controller Board is connected to the General IO Board.

Showing the result of AD conversion on an L E D bar

1. Launch the *MPLABX IDE* and create a new project called *Lab5*.

2. Add the file *ADC.c* to the *Lab5* project *Source File* folder.
Make sure *Copy* is ticked before you click *Select*. If you have forgotten the steps, you will need to refer to the previous lab sheet.

3. Study the code and describe what this program will do:

_____

4. Build, download and execute the program.

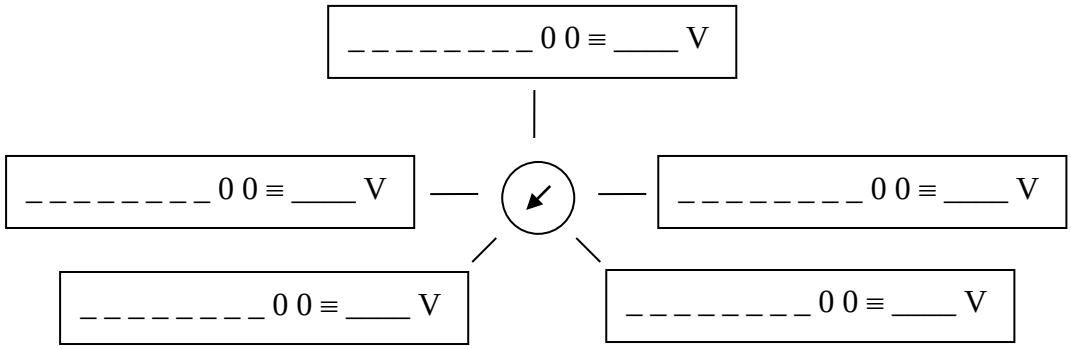
5. Turn the variable resistor R1 (on the General IO Board) to each of the 5 positions marked below.

   Observe & record the 10-bit digital result on the LED bar (also on the General IO Board) – actually only the 8 most significant bits are shown on the LED bar, so the 2 least significant bits can be taken as 0 0.

   The equivalent input voltage can be "back computed" using the formula:

   Voltage = (N-bit digital result x 5 V) / 2^N (N = 10 for 10-bit converter)

   [Note: some said it should be divided by 2^N – 1. It really depends on the coding scheme used. See Wikipedia for a discussion.]



6. Are the results as expected i.e. the 10-bit digital result as well as the equivalent input voltage should increase as the knob is turned clockwise?

   Your answer: _____

Fish tank "water level monitoring"

7. Modify the code above so that if the water level is above a certain level (for instance when the 10-bit digital result exceeds *0xD0*), the pump motor at *RC2* is turned on to pump away the excess water. On the other hand, if the water level is below a certain level (for instance *0x10*), the relay at *RC1* is turned on to sound an alarm to alert the owner.

   Note that you need to define *LOW* in your code. The *if-else* statement must also be expanded to control the relay.

8. Build, download and execute the program to verify your coding. Debug until the program can work.

// ADC.c

```
// Program to use ADC to read variable resistor input and display on LEDs

#include <xc.h>

void main(void)
{
   #define HIGH 0xD0        // HIGH water level indicator

   TRISD = 0x00;            // Set PORTD to be output
   PORTD = 0x00;            // Initialise PORTD LEDs to zeros

   TRISCbits.TRISC1 = 0;  // RC1 as output pin
    PORTCbits.RC1    = 0;   // RC1 is connected to Relay

   TRISCbits.TRISC2 = 0;  // RC2 as output pin
    PORTCbits.RC2    = 0;   // RC2 is connected to Motor

   /* Initialise analog to digital conversion setting */

   ADCON0 = 0b00000001;          // bit5-2  0000 select channel 0 conversion
                                 // bit1    A/D conversion status bit
                                 //             1- GO to starts the conversion
                                 //             0 - DONE when A/D is completed
                                 // bit0    Set to 1 to power up A/D

   ADCON1 = 0b00001100;          // bit5    reference is VSS
                                 // bit4    reference is VDD
                                 // bit3-0  AN2 to AN0 Analog, the rest Digital

   ADCON2 = 0b00 010 110;        // bit7    A/D Result Format:
                                 //             0 - Left justified
                                 //             1 - Right justified
                                 // bit5-3  010 acquisition time = 4 TAD
                                 // bit2-0  110 conversion clock = Fosc / 64

   for( ; ; )
   {
      ADCON0bits.GO = 1;         // This is bit2 of ADCON0, START CONVERSION NOW

      while(ADCON0bits.GO == 1); // Waiting for DONE

      PORTD = ADRESH;            // Displaying only the upper 8-bits of the A/D result

      if(ADRESH > HIGH)
      {
         PORTCbits.RC2 = 1;      // Turn on Motor
      }
      else
      {
         PORTCbits.RC2   = 0;     // Turn off Motor
      }
   }
}
```

Lab 6 – Programmable timer and PWM (Pulse Width Modulation)

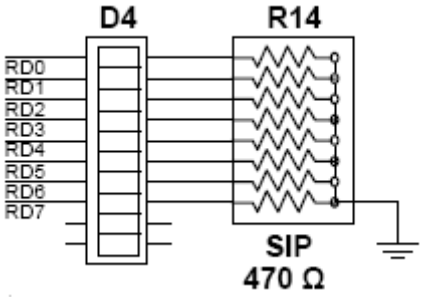
Objectives

- ☐ To learn to introduce a time delay using Timer0 in the PIC18F4550 microcontroller.
- ☐ To learn to use PWM for the speed control of a DC motor.

Introduction / Briefing

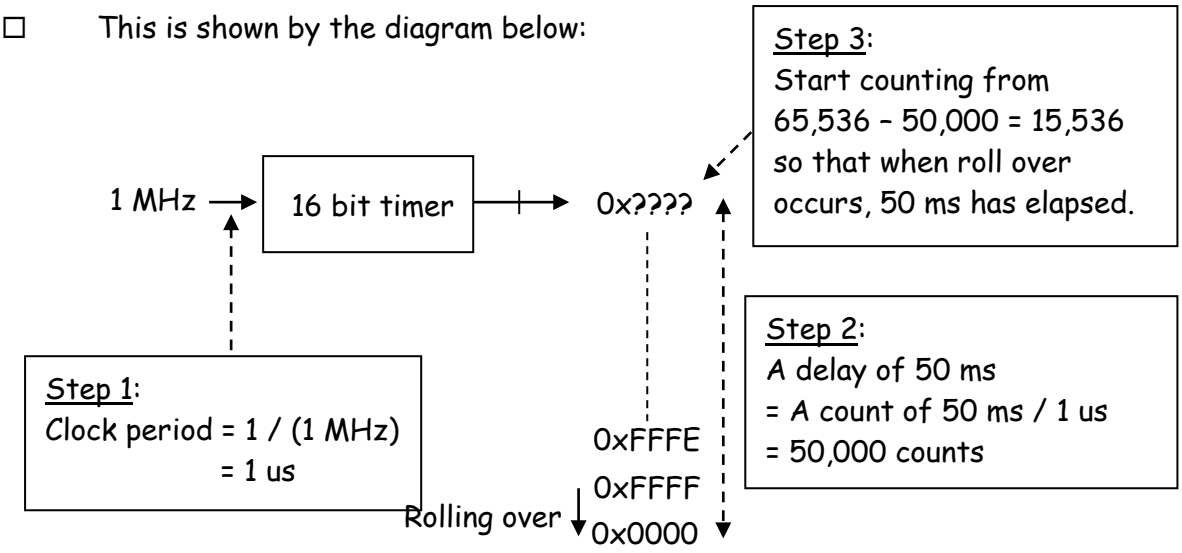
PIC18F4550's Timer0 for delay

- ☐ The PIC18F4550 microcontroller has four internal timers. We will first use Timer0 to introduce a time delay and then use Timer2 to control the speed of a DC motor using Pulse Width Modulation (PWM).
- ☐ In the first part of the experiment, the LED bar at Port D will be blinked (turned ON and OFF repeatedly) at 1 second interval. Let's figure out how the 1 second interval or delay can be created with the help of Timer0.



- ☐ Example 1: Assume that a 16-bit counter/timer is clocked by a 1 MHz clock signal. How long does it take to count from 0000 to FFFF and then roll over? [Roll over means changing from the maximum count of FFFF to 0000.]
- ☐ Counting from 0000 to FFFF and then rolling over is equivalent to 65,536 counts. Since each count takes 1us, this is equal to 65,536 us = 65.536 ms.
- ☐ Example 2: Now try this: what count should the counter starts with, so that exactly 50 ms has elapsed when roll over occurs?
- ☐ Since each count takes 1us, 50 ms is equal to 50 ms / 1us = 50,000 counts. So the count should start from 65,536 – 50,000 = 15,536.

☐ This is shown by the diagram below:



☐ In the case of PIC18F4550's Timer 0:

- The timer/counter clock frequency = Fosc / 4 = 48 MHz / 4 = 12 MHz.
- An interrupt flag TMR0IF will be set to 1 whenever the timer overflows from FFFF to 0000.
- A pre-scaler can be used to slow down the clock. For instance, a pre-scale value of 256 slows down the clock by 256 times, effectively, the timer/counter is clocked by a 12MHz / 256 clock signal.

☐ Example 3: Assume that the PIC18F4550's 16-bit Timer0 is clocked by a 48 MHz / 4 clock signal and a pre-scale value of 256 is used. What count should the timer starts with, so that exactly 1 sec has elapsed when roll over occurs?

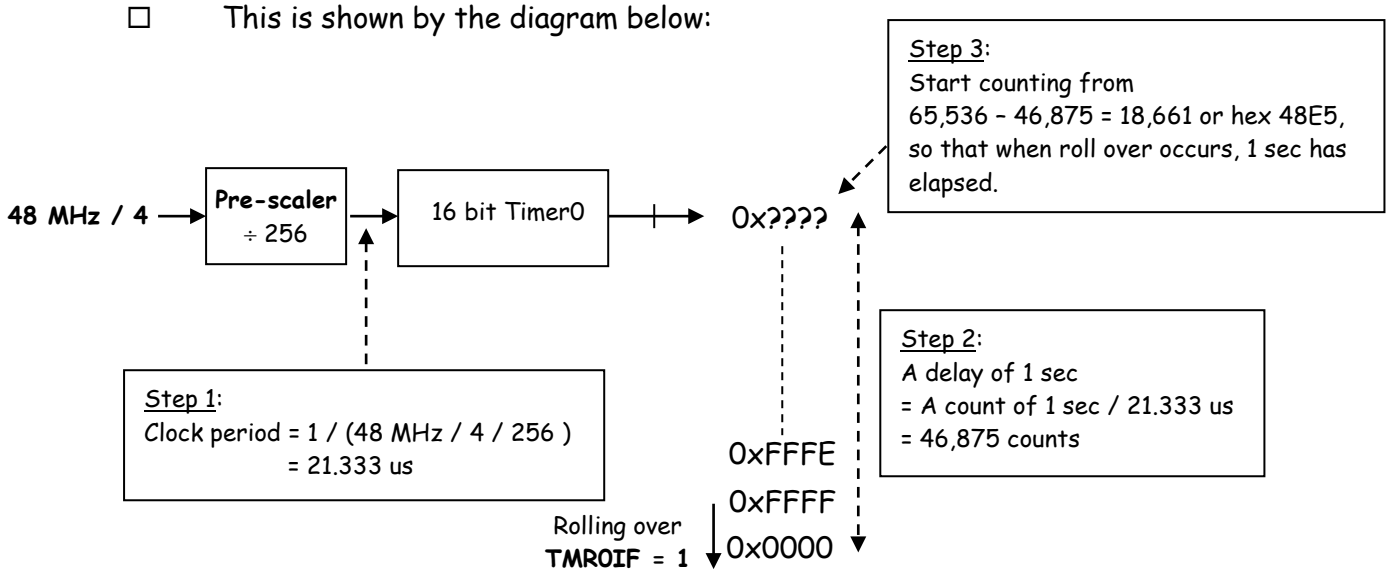
☐ Effective clock frequency for Timer0 = 48 MHz / 4 / 256 = 46875 Hz

☐ Each count = 1 / (46875 Hz) = 21.333 us.

☐ A delay of 1 sec = a count of 1 sec / 21.333 us = 46,875 counts.

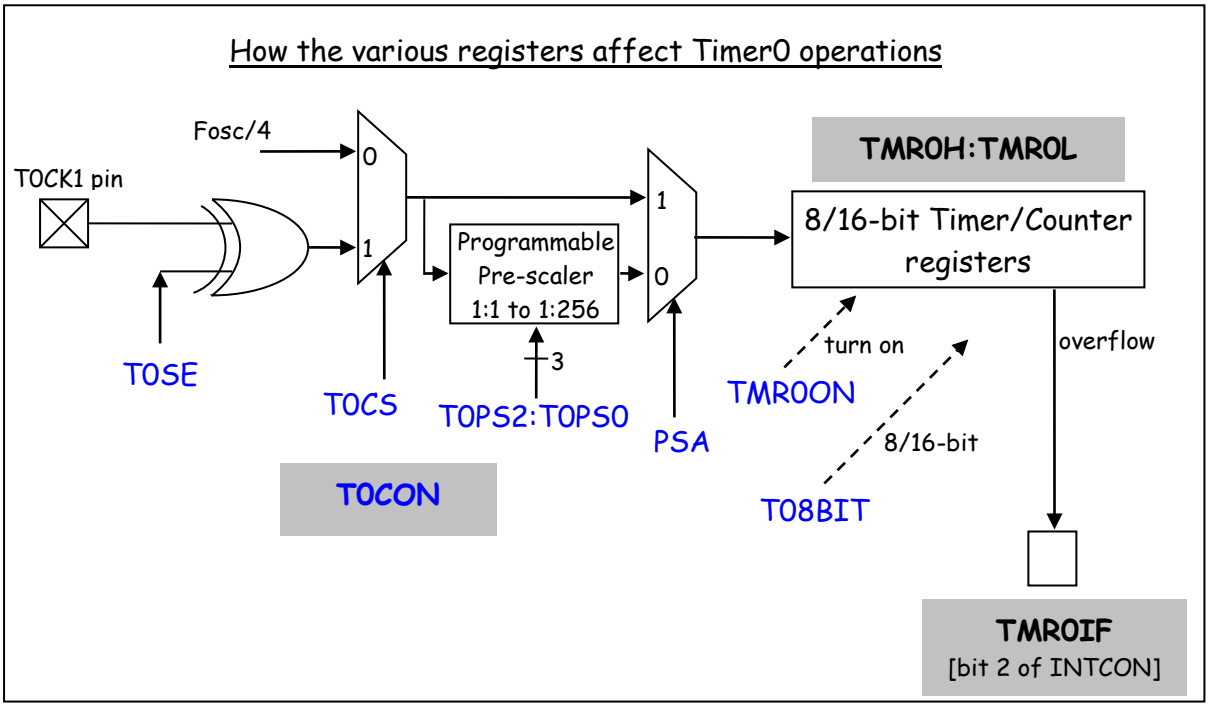
☐ So the Timer0 should start from 65536 – 46875 = 18661 or hex 48E5 – the conversion from decimal to hex can be done using the PC's calculator (Programs -> Accessories -> Calculator).

☐ This is shown by the diagram below:



☐ How do you set a pre-scaler of 256 & a starting count value of hex 48E5? How do you know that Timer0 overflow has occurred i.e. TMR0IF has become 1? To answer these questions, we must take a look at Timer0 registers.

☐ The following diagram shows how the various Timer0 registers affect its operations. You can come back to examine this diagram later, after the individual registers have been described.



TMR0H & TMR0L (Timer0 High & Low Registers) – These two 8-bit registers together form a 16-bit timer/counter.

TMR0H								TMR0L							
D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

☐ To set a starting count value of hex 48E5, hex 48 should be written to TMR0H first, and then hex E5 written to TMR0L.

Q1: Give the C code to set a starting count value of hex 48E5:

Your answer: TMR0H = _____;
_____ ;

INTCON (Interrupt Control Register) bit 2 =TMR0IF (Timer0 Interrupt "overflow" Flag) – This bit is set to 1 whenever the Timer0 overflows i.e. count from FFFF to 0000.

INTCON

					TMR0IF		

Q2: Give the C code to wait for Timer0 overflow to occur:

Your answer: while (_____bits._____ == 0);

T0CON (Timer0 Control Register) – This register controls the Timer0 operation (as described below). It is used to turn the timer ON/OFF and to set the pre-scaler value.

TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0

TMR0ON D7 Timer0 **ON** and **OFF control** bit
1 = Enable (start) Timer0
0 = Stop Timer0

T08BIT D6 Timer0 **8-bit / 16-bit selector** bit
1 = Timer0 is configured as an 8-bit timer/counter
0 = Timer0 is configured as a 16-bit timer/counter

T0CS D5 Timer0 **clock source select** bit
1 = External clock from RA4/T0CK1 pin
0 = Internal clock (Fosc/4 from XTAL oscillator)

T0SE D4 Timer0 **source edge select** bit
1 = Increment on H-to-L transition on T0CK1 pin
0 = Increment on L-to-H transition on T0CK1 pin

PSA D3 Timer0 **pre-scaler assignment** bit
1 = Timer0 clock input bypasses pre-scaler
0 = Timer0 clock input comes from pre-scaler output

T0PS2:T0PS0

D2	D1	D0	Timer0 **pre-scaler selector**
0	0	0 = 1:2	Pre-scale value (Fosc/4/2)
0	0	1 = 1:4	Pre-scale value (Fosc/4/4)
0	1	0 = 1:8	Pre-scale value (Fosc/4/8)
0	1	1 = 1:16	Pre-scale value (Fosc/4/16)
1	0	0 = 1:32	Pre-scale value (Fosc/4/32)
1	0	1 = 1:64	Pre-scale value (Fosc/4/64)
1	1	0 = 1:128	Pre-scale value (Fosc/4/128)
1	1	1 = 1:256	Pre-scale value (Fosc/4/256)

Q3: What binary pattern must be written to T0CON to use Timer0 as a 16-bit timer using internal clock (Fosc/4) and a pre-scale value of 256? The timer is NOT to be turned on at this point.

Your answer:

T0CON (Timer0 Control Register)

TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0

☐ The C code required is T0CON = 0b 0 0 0 0 0 <u>111</u>;

☐ Putting the pieces together, the C code to introduce a time delay of 1 second can be written as follows:

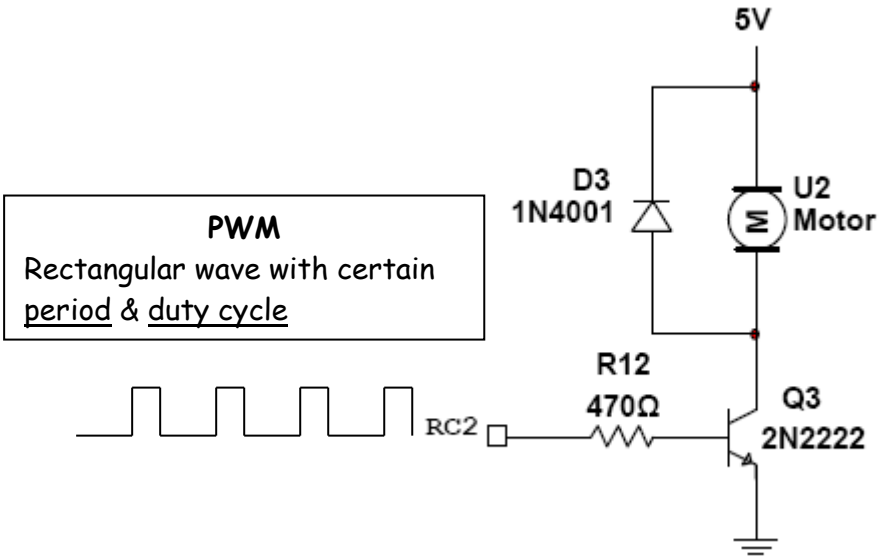
```
T0CON=0b00000111;          // Off Timer0, 16-bits mode, Fosc/4, prescaler of 256

TMR0H=0X48;                // Starting count value
TMR0L=0XE5;

INTCONbits.TMR0IF=0;       // Clear flag first
T0CONbits.TMR0ON=1;        // Turn on Timer 0

while(INTCONbits.TMR0IF==0); // Wait for time's up i.e. TMR0IF==1
T0CONbits.TMR0ON=0;        // Turn off Timer 0 to stop counting
```

☐ Description: 1. First, Timer0 is configured but turned OFF. 2. Then, the starting count value is written toTMR0H, followed by TMR0L. 3. Next, the flag is cleared and Timer0 turned ON. 4. After that, the while loop is used to wait for 1 second to elapse i.e. for the TMR0IF interrupt flag to be set. 5. Finally, Timer0 is turned OFF.

☐ With this, you should be able to figure out the TimerDelay.c used in the first part of the experiment to blink the LED bar at Port D at 1 second interval.

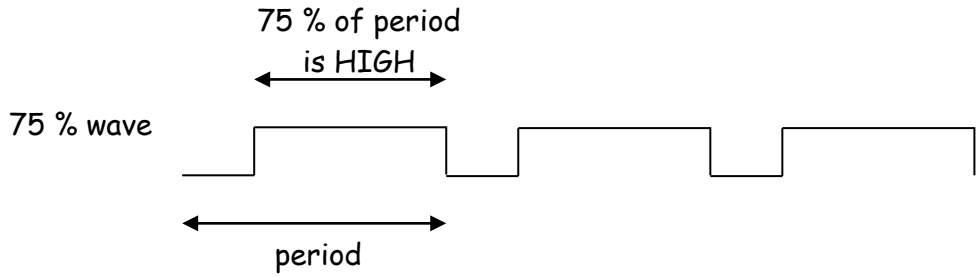
PIC18F4550's Timer2 for PWM

☐ In the second part of the experiment, PWM (Pulse Width Modulation) will be used to control the speed of the DC motor connected to RC2. We will now describe what PWM is and how it can be created with the help of Timer2.



PWM

- PWM (Pulse Width Modulation) is a method used to control the speed of a DC motor.
- When 5V is applied to a small DC motor, it turns at a certain speed.
- A 75% duty cycle rectangular wave is high for 75% of the time (and low for 25% of the time). When this is applied, the motor slows down – effectively, it is getting 5V x 75% or 3.75V d.c.



- When a 50% duty cycle wave is applied, the motor slows down further, as it is effectively getting 2.5V d.c.
- Pulse Width Modulation = varying the duty cycle of the rectangular wave (i.e. varying the pulse width) to control the motor speed.
- In creating a rectangular wave or pulse train, we must know both 1. the period and 2. the duty cycle.

- How do we create a rectangular wave of a certain period and duty cycle in PIC18F4550? Let's try a 5 kHz, 25% duty cycle wave.
- PIC18F4550 has a **CCP** (Capture Compare) module which comes with PWM capability. The PWM output comes out at RC2.
- For PWM, the CCP module uses two **Timer2** registers to specify the **period**:

PWM period = (PR2 + 1) x 4 x N x Tosc

where PR2 is Timer2's 8-bit "Period register"
N = Timer2's pre-scale value of 1, 4 or 16, as set in T2CON (Timer2 Control) register (* see box on next page)
Tosc = 1 / Fosc, where Fosc = 48 MHz

Some Timer2 registers

PR2 (Period Register)

D7	D6	D5	D4	D3	D2	D1	D0

T2CON (Timer2 Control Register)

D7	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0

D7 – not used

TOUTPS3:TOUTPS0	D6 D5 D4 D3	Timer2 output **post-scaler selector** 0 0 0 0 = 1:1 Post-scale value 0 0 0 1 = 1:2 Post-scale value 0 0 1 0 = 1:3 Post-scale value … 1 1 1 1 = 1:16 Post-scale value
TMR2ON	D2	Timer2 **ON** and **OFF control** bit 1 = Enable (start) Timer2 0 = Stop Timer2
T2CKPS1:T2CKPS0	D1 D0	Timer2 clock **pre-scaler selector** 0 0 = 1:1 Pre-scale value 0 1 = 1:4 Pre-scale value 1 X = 1:16 Pre-scale value

Q4. What is the PR2 value to generate a 5 kHz wave, assuming a pre-scaler of 16?

Your answer: _____

☐ Solution

Frequency = 5 k ⇨ Period = 1 / 5k = 0.2 m

Tosc = 1 / 48M Pre-scaler, N = 16

Substituting into the formula,

PWM period = (PR2 + 1) x 4 x N x Tosc

1 / 5k = (PR2 + 1) x 4 x 16 x (1 / 48M)

=> PR2 = 149

- The "**High Time**" (or "On Time") is specified using another register called **CCPR1L**, as follows:

  High Time = 25% of Period

  25% x 149 = 37.25 = 37 (ignoring the decimal portion)

  => CCPR1L = 37

- The bottom 4 bits of the CCP1 Control Register (**CCP1CON**) should be set to 1100 for PWM operation.

CCP1CON (CCP1 Control Register)

D7	D6	DC1B1	DC1B0	CCP1M3	CCP1M2	CCP1M1	CCP1M0

1 1 0 0 for PWM mode

Other combinations for Capture / Compare modes not studied in this module

- Let's put everything together to program the PIC18F4550 to generate a 5 kHz, 25% wave.

- The complete program (in outline form) is given below:

```
TRISC=0x00;   // RC2 is connect to motor and should be made an output
T2CON=0b0 0000 1 11;  // Timer2 is On, Prescaler is 16
CCP1CON=0b00 00 1100;    // Turn on PWM
PR2 = 149;                // Load PWM period of 0.2 ms or 5 kHz
CCPR1L = 37;              // Load PWM on time ( i.e. 25% x 149 )
```

- With this, you should be able to figure out the TimerPWM.c used in the second part of the experiment to control DC motor speed using PWM.

Activites:

Before you begin, ensure that the Micro-controller Board is connected to the General IO Board.

Blinking an LED bar at 1 second interval, using a delay created using Timer0

1. Launch the *MPLABX IDE* and create a new project called *Lab6*.

2. Add the file *TimerDelay.c* to the *Lab6* project *Source File* folder. Make sure *Copy* is ticked before you click *Select*. If you have forgotten the steps, you will need to refer to the previous lab sheet.

3. Study the code and describe what this program will do:

Blink LED bar at 1 sec intervals. ⇨

_____

4. Note that the main function configures the *Timer0* but does not turn it ON. The *Timer0* is only turned on in the *Delay1sec* function.

5. Build, download and execute the program. Observe the result and see if it is as expected.

Change start count value to change delay.

6. Modify the code so that the delay is 0.5 second (instead of 1 second). Keep pre-scaler of 256.

> Hint:
>
> - Since Fosc remains at 48 MHz and pre-scaler remains at 256, the effective clock frequency of Timer0 remains unchanged at 48 MHz / 4 / 256 = 46875 Hz.
> - Each count = 1 / (46875 Hz) = 21.333 us, the same as before.
> - A delay of 0.5 sec = a count of 0.5 sec / 21.333 us = 23438 counts.
> - So the Timer0 should start from 65536–23438 = 42098 or hex A472.

7. Build, download and execute the program to verify your coding. The LED bar now blinks at a faster rate.

Change pre-scaler to change delay.

8. Without changing the start count value in 6 above, modify the code to use a pre-scaler of 64 (instead of 256). [Hint: T0CON = 0b00000___;] What effect do you think this will have on the rate of blinking?

   Your answer: The LED bar will blink at a _____ (faster/slower) rate.

9. Build, download and execute the program to verify your answer above.

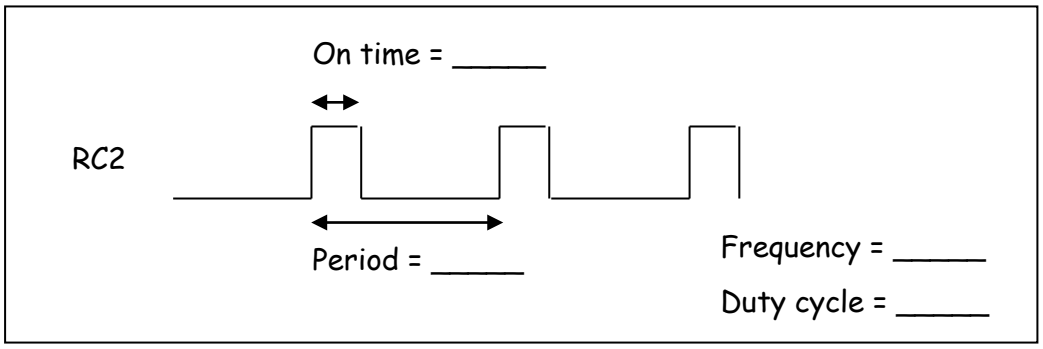
Controlling DC motor speed, using PWM created using Timer2

10. Replace *TimerDelay.c* with *TimerPWM.c*.

PWM 5kHz 25% duty cycle

11. The TimerPWM.c code is to produce a 5 kHz, 25% duty cycle wave at *RC2* using PWM.

12. Build, download and execute the program. Use the oscilloscope connected to *RC2* (/ DC motor) to see if the period & duty cycle are correct. Record your observations below:



13. Note that *PR2* = 149, *CCPR1L* = 37 in this case.

14. Note also the speed of the DC motor at 25%.

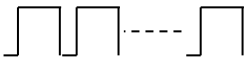
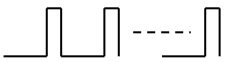
PWM 5kHz 25%, 50% or 75% duty cycle depending on dip switches

15. The value of *CCPR1L* to get a 5 kHz, 50% duty cycle wave is simply

    50% x 149 = 74

    Likewise, to get a 5 kHz, 75% duty cycle wave, *CCPR1L* = 75% x 149 = 112

16. Modify the code so that the duty cycle produced depends on the settings of the (active low) dip switches (on the General IO Board) connected to *RA4* and *RA3*, as follows:

RA4	RA3	Duty Cycle
Closed i.e. == 0	don't care	75 %  (high speed)
Open i.e. == 1	Closed i.e. == 0	50 %  (medium speed)
Open i.e. == 1	Open i.e. == 1	25 %  (low speed)

17. Build, download and execute the program to verify your coding. Debug until the program can work.

// TimerDelay.c

```
/* TimerDelay.c Program containing a 1 sec delay function
*  Use Timer0
*  Frequency of OSC = 48 MHz, Prescaler = 256
*  TMR0H:TMR0L contain the starting count value
*  Monitor TMR0IF flag. When TMR0IF = 1, one sec is over
*/
#include <xc.h>

void Delay1sec(void);                   // Function to provide 1 sec delay using Timer0
void Delay1sec(void)
{
   TMR0H=0X48;                          // Starting count value
   TMR0L=0XE5;

   INTCONbits.TMR0IF=0;                 // Clear flag first
   T0CONbits.TMR0ON=1;                  // Turn on Timer 0

   while(INTCONbits.TMR0IF==0);         // Wait for time is up when TMR0IF=1
   T0CONbits.TMR0ON=0;                  // Turn off Timer 0 to stop counting
}

void main(void)
{
   TRISD=0x00;                          // PortD connected to 8 LEDs
   T0CON=0b00000111;                    // Off Timer0, 16-bits mode, prescaler to 256

   while(1)                             // Repeatedly
   {
      PORTD=0x00;                       // Off all LEDs
      Delay1sec();

      PORTD=0xFF;                       // On all LEDs
      Delay1sec();

   }
}
```

// TimerPWM.c

```
/* TimerPWM.c Program to generate PWM at RC2
*  Use Timer2
*  Frequency of OSC = 48 MHz, Prescaler = 16
*  PR2 register set the frequency of waveform
*  CCPR1L with CP1CONbits.DC1B0, CCP1CONbits.DC1B1 set the On-Time
*  Use Timer0 for the one second delay function
*/
#include <xc.h>

void Delay1sec(void);                 // Function to provide 1 sec delay using Timer0
void Delay1sec(void)
{
   TMR0H=0X48;                        // Starting count value
   TMR0L=0XE5;

   INTCONbits.TMR0IF=0;               // Clear flag first
   T0CONbits.TMR0ON=1;                // Turn on Timer 0

   while(INTCONbits.TMR0IF==0);       // Wait for time is up when TMR0IF=1
   T0CONbits.TMR0ON=0;                // Turn off Timer 0 to stop counting
}

void main(void)
{
// Do not remove these as well=============
   ADCON1 = 0x0F;
   CMCON = 0x07;
// ========================================
// Your MAIN program Starts here: =========

   TRISC=0x00;                        // PortC RC2 connects to motor
   TRISD=0x00;                        // PortD connected to 8 LEDs
   T0CON=0b00000111;                  // Off Timer0, 16-bits mode, prescaler to 256

   T2CON=0b00000111;                  // Timer2 is On, Prescaler is 16

   CCP1CON=0b00001100;                // Turn on PWM on CCP1, output at RC2
   PR2 = 149;                         // Load period of PWM 0.2msec for 5KHz

   while(1)                           // Repeatedly
   {
      CCPR1L = 37;                    // Duty cycle 25%, 149 x 25% = 37
   }
}
```

Lab 7 – Interrupt programming

Objectives

- ☐ To learn to use PIC18F4550 microcontroller's INT0 external hardware interrupt & Timer0 interrupt.
- ☐ To learn to the sequence of code execution in an interrupt event.

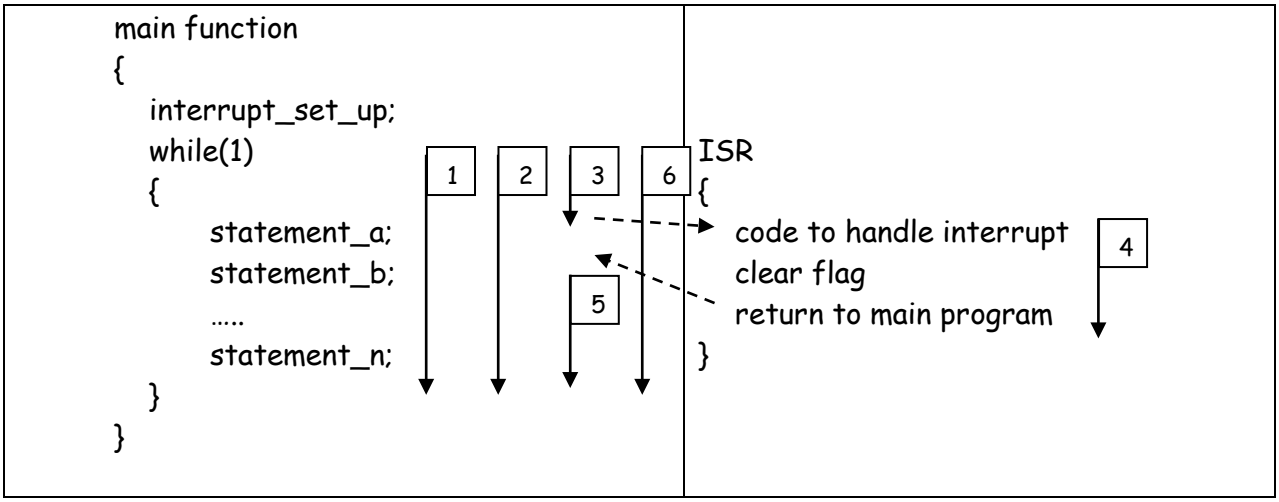
Introduction / Briefing

What is interrupt?

- ☐ A microcontroller can use the interrupt method to respond to an event.
- ☐ In this method, when a peripheral (e.g. an I/O pin or a timer) needs something to be done, it notifies the micro-controller, which stops whatever it is doing and "serves" the peripheral. After that, the micro-controller goes back to continue what it was doing.
- ☐ As an analogy, you could be reading newspaper. When there is a buzz tone on your hand phone, you are "interrupted" – you stop reading and reply an SMS. After that, you continue reading your newspaper where you left off.
- ☐ The program associated with the interrupt is aptly called the interrupt service routine or ISR.
- ☐ In this experiment, you will learn the basics of interrupt, focusing on INT0 external hardware interrupt and Timer0 interrupt, (though there are many other interrupt sources in the PIC).

Sequence of code execution

☐ The diagram below shows the sequence of code execution in an interrupt event:



☐ After interrupt has been set up, the while loop in the main function is executed over and over again (1, 2…).

☐ Let's say an interrupt event occurs when statement_a is being executed (3).

☐ The microcontroller will complete the execution of this statement. Then it will go to a specific location (*) called the "interrupt vector" to look for the ISR (interrupt service routine).

☐ In the ISR, the codes to handle the interrupt will get executed (4).

☐ After that, the microcontroller will return to the main function to continue with statement_b (5), and the while loop will get executed over and over again (6…).

(*) In the lab, a "boot-loader" is used in the PIC18F4550. This program downloads a user program from a PC via the USB port. The "boot-loader" changes the high and low-priority interrupt-vectors to 0x001008 and 0x001018, respectively.

INT0 external hardware interrupt

☐ The PIC18F4550 has 3 external hardware interrupts: INT0, INT1 and INT2 which use pins RB0, RB1 and RB2 respectively. We will discuss INT0 (and INT1 and INT2 are similar in terms of operation).

☐ The INT0 interrupt responds to a change of voltage i.e. a transition at RB0.

☐ To enable the INT0 interrupt, set both the GIE (Global Interrupt Enable) and the INT0IE (INT0 Interrupt Enable) bits in the INTCON register.

INTCON (Interrupt Control Register)

GIE			INT0IE			INT0IF	
1			1				

Q1. Give the C-code to enable the INT0 interrupt.

INTCONbits._____ = _____;
_____;

☐ The INTEDG0 bit of the INTCON2 register is used to specify whether interrupt is to occur on a falling (i.e. a high to low transition) or a rising (i.e. a low to high transition) edge at RB0:

INTCON2 (Interrupt Control Register 2)

	INTEDG0						
	0						

INTEDG0 = 0: INT0 interrupt on falling edge at RB0
INTEDGO = 1: INT0 interrupt on rising edge at RB0 (power-on reset default)

Q2. Give the C-code to select falling edge triggering.

INTCON2bits._____ = _____;

☐ Note that falling edge triggering has been chosen because on the microcontroller board, the push button switch at RB0 has been connected as "active low".

☐ When interrupt has been enabled and there is a falling edge at RB0, the flag INT0IF (external hardware INTerrupt 0 Interrupt Flag) in the INTCON register will be set. To clear this flag, so that future interrupt can be noticed, use the code INTCONbits.INT0IF = 0;

☐ With these, you should be able to understand the code in Int_INT0_b.c.

Interrupt priority (D.I.Y.)

☐ By default, all interrupts are "high priority".

☐ It is also possible to make some interrupts "high priority" and others "low priority". This is done by setting the **IPEN** (Interrupt Priority ENable) bit in the RCON register.

☐ When interrupt priority is enabled, we must classify each interrupt source as high priority or low priority. This is done by putting 0 (for low priority) or 1 (for high priority) in the **IP** (interrupt priority) bit of each interrupt source.

☐ The IP bits for the different interrupt sources are spread across several registers – INTCON, INTCON2, INTCON3, IPR1 and IPR2. We will not go into the details of all these.

☐ A higher priority interrupt can interrupt a low priority interrupt but NOT vice-versa.

Timer0 interrupt

☐ In the last experiment, you used Timer0 to introduce a delay of 1 second. The C code was:

```
T0CON=0b00000111;                    // Off Timer0, 16-bits mode, Fosc/4, prescaler to 256

TMR0H=0X48;                          // Starting count value
TMR0L=0XE5;

INTCONbits.TMR0IF=0;                 // Clear flag first
T0CONbits.TMR0ON=1;                  // Turn on Timer 0

while(INTCONbits.TMR0IF==0);         // Wait for time is up when TMR0IF=1
T0CONbits.TMR0ON=0;                  // Turn off Timer 0 to stop counting
```

First, Timer0 is configured but turned OFF. Then, the starting count value is written to TMR0H, followed by TMR0L. Next, the flag is cleared and Timer0 turned ON. After that, the while loop is used to wait for 1 second to elapse i.e. for the TMR0IF interrupt flag to be set. Finally, Timer0 is turned OFF.

☐ Instead of "polling" for the timer overflow (from FFFF to 0000) using

```
while (INTCONbits.TMR0IF == 0);
```

the "interrupt" method can be used. The advantages of the interrupt method over the polling method are discussed in the lecture.

☐ The Timer0 interrupt responds to Timer0 overflow.

☐ To enable the Timer0 interrupt, set both the GIE (Global Interrupt Enable) and the Timer0IE (Timer0 Interrupt Enable) bits in the INTCON register.

INTCON (Interrupt Control Register)

GIE		TMR0IE			TMR0IF		
1		1					

Q3. Give the C-code to enable the Timer0 interrupt.

```
INTCONbits._______ = _______;
_____________________________;
```

☐ When interrupt has been enabled and there is a Timer0 overflow, the flag Timer0IF (TiMeR0 overflow Interrupt Flag) in the INTCON register will be set. To clear this flag, so that future interrupt can be noticed, use the code INTCONbits.TMR0IF = 0;

☐ With these, you should be able to understand the Timer0 Interrupt code outlined as follows. (This is similar to, but not exactly the same as, the Int_TMR0.c used in the experiment.)

main function	
INTCONbits.GIEH =1;	Global Interrupt Enable
INTCONbits.TMR0IE = 1;	Timer0 Interrupt Enable
T0CON = 0b00000111;	Stop Timer0, 16-bit, Fosc/4, pre-scaler 256
TMR0H = 0x48; TMR0L = 0xE5;	Starting count value for a 1 second delay
INTCONbits.TMR0IF = 0;	Clear Flag
T0CONbits.TMR0ON = 1;	Turn on Timer0

while (1)
{
…
}
	PIC free to do other things in while loop e.g. use slide switch to turn motor on/off

ISR	
if (INTCONbits.TMR0IF) {	Check that it is really Timer0 overflow causing the interrupt
TMR0H = 0x48; TMR0L = 0xE5;	Reload Timer0 with starting count value - for the next round
….	
INTCONbits.TMR0IF = 0;	Clear flag, to get ready for the next interrupt
}	

Enable interrupt

Configure Timer0 for 1 sec delay

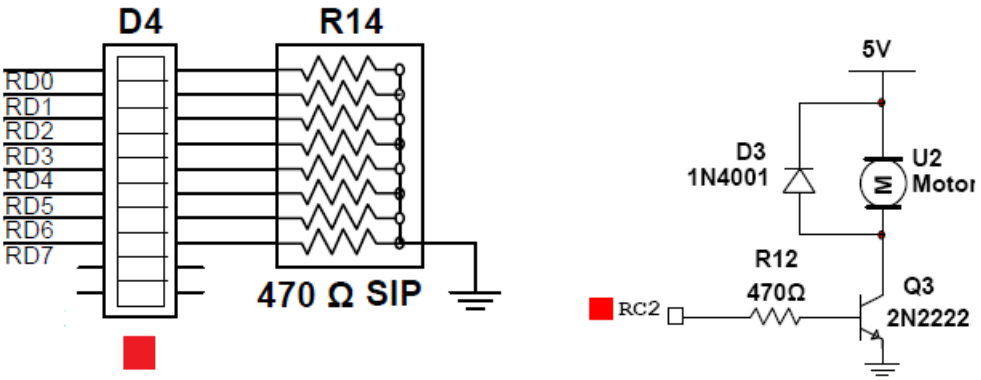
Respond to interrupt and reload for next interrupt.

Activites:

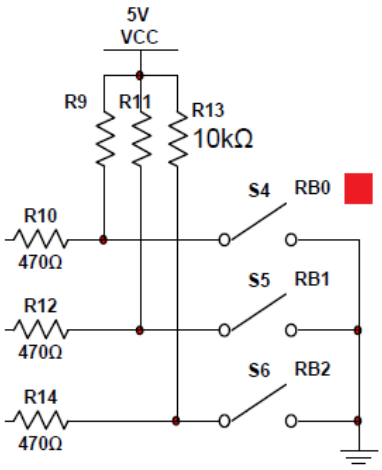
Before you begin, ensure that the Micro-controller Board is connected to the General IO Board.

External hardware interrupt

1. This part of the experiment uses the *LED bar* connected to *Port D* and the *motor* connected to *RC2*. Both are found on the *General IO Board*.



2. This part of the experiment also uses the *push button* connected to *RB0*. This is found on the *Micro-controller Board*.



3. Launch the *MPLABX IDE* and create a new project called *Lab7*.

4. Add the file *Int_INT0_a.c* to the *Lab7* project *Source File* folder. Make sure *Copy* is ticked before you click *Select*. If you have forgotten the steps, you will need to refer to the previous lab sheet.

Polling an active low switch at RB0 ⇨

5. Study the code. In the main program the function "*LED_RD7_RD0*" is first called to light up the LED's in the bar, one by one, from left to right.

Then the switch at *RB0* is checked. The first time it is pressed (i.e. becomes 0), the motor at *RC2* is turned on. The next time it is pressed, the motor is turned off.

No interrupt is used and the *RB0* switch is responded to only after the LED bar sequence has completed.

6. Build, download and execute the program. Press the switch at *RB0* to control the motor on/off. Does the motor respond promptly to the switch?

   Your answer: _____

Using interrupt for the active low switch at RB0

7. Replace *Int_INT0_a.c* with *Int_INT0_b.c* .

8. Study the code. This time, *INT0* interrupt is used.

   Can you find the main function and the ISR?

   In the main function, can you find the codes that enable the interrupt, select falling edge triggering and clear the flag?

9. In the while loop in the main function, the function "*LED_RD7_RD0*" is called to light up the LED's in the bar, one by one, from left to right.

10. Pressing the switch at *RB0* causes an interrupt event to occur. The interrupt service routine for the "active low" button at *RB0* is called *ISR_PortB0_low*.

    The ISR first checks that the *INT0IF* is set.

    The first time the switch at *RB0* is pressed (i.e. becomes 0), the motor at *RC2* is turned on. The next time it is pressed, the motor is turned off.

    The ISR clears the flag at the end so that the next interrupt event can be recognised and responded to.

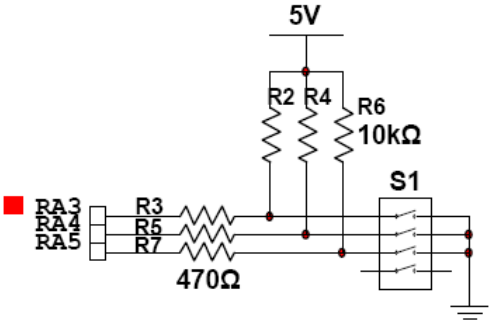
    {You may have noticed that there is no line of code "checking" if *RB0* has been pressed in the form of "*if (PORTBbits.RB0 == 0)*". This is because interrupt is used, instead of polling.}

11. Build, download and execute the program. Press the switch at *RB0* to control the motor on/off. Does the motor respond promptly to the switch?

    Your answer: _____

Timer0 interrupt

12. This part of the experiment also uses the on/off *switch* connected to *RA3*. This is found on the *General IO Board*.



Using interrupt for Timer0 overflow

13. Replace *Int_INT0_b.c* with *Int_TMR0.c*.

14. Study the code. This time, *Timer0* interrupt is used.

    Can you find the main function and the ISR?

    In the main function, can you find the codes that enable the interrupt, configure the *Timer0*, clear the flag and turn on the *Timer0*?

15. In the while loop in the main function, the switch at *RA3* is used to turn the motor on/off.

16. *Timer0* overflow causes an interrupt event to occur. The interrupt service routine for the *Timer0* overflow is called *ISR_Timer0_Int*.

    The ISR first checks if *TMR0IF* is set.

    If so, the timer is reloaded with the starting count value (to get ready for the next round). A variable *j* is incremented and then displayed on the LED bar connected *Port D*.

    At the end, the *TMR0IF* flag is cleared, so that the next interrupt event can be recognised and responded to.

17. Build, download and execute the program. Turn the switch at *RA3* on/off to see if the motor can be turned on/off.

    Your answer: _____

18. Next, look at the LED bar and record your observation below:

    Your answer: _____

19. There are two processes – main and ISR - in the program but the micro-controller can only run one process at a time. Do you feel that the motor is responding well to the switching at *RA3* when the LED's are counting?

Your answer: _____

Beeping every second

20. Finally, modify the program so that the buzzer will beep every second.

Your answer: _____

21. Build, download and execute the program to verify your coding. Debug until the program can work.

// Int_**INT0_a.c** Polling based program

```
#include <xc.h>
#include "delays.h"

unsigned char j;
unsigned char press;

void LED_RD7_RD0(void)// The function to shift a set-bit from left to right
{
       j = 0x80;            //  Initialise  j  with  B1000
                            0000  //  ie  the  leftmost  bit
                            (or MSB)
       while(j!=0x01)       // Check that the bit has not been shifted
                            // to the right-most bit (LSB) ie B00000001
       {
         PORTD= j;          // Display j at PORTD
         delay_ms(250);     // Calling a delay function from delays.h
          j = j>>1;         // Making use of LOGICAL-RIGHT-SHIFT bit-wise
       }                    // operator to shift data to the right

       PORTD = j;           // Display j at PORTD
}                           // Stop at B00000001


void main(void)             // Main Function
{
       ADCON1 = 0x0F;
       CMCON = 0x07;

       TRISBbits.TRISB0 = 1;      // RB0 is the push button switch for INT0
       TRISCbits.TRISC2 = 0;      // RC2 connects to a DC motor
       TRISD = 0x00;        // PortD connects to a bar LEDs

       PORTD = 0x00;        // LEDs all off
       press = 0;           // Not pressing yet

       while(1)             // Main Process
       {
           LED_RD7_RD0();  // Move Port D LEDs from bit7 to bit0

           // polling the switch at RB0
           if (PORTBbits.RB0 == 0)
           {
             press++;       // To track first or second time pressing RB0 switch
```

```
            if (press == 1)                 // First press
            {
               PORTCbits.RC2 = 1;           // Turn On Motor
            }
            else
            if (press == 2)                 // Second press
            {
              PORTCbits.RC2 = 0;            // Else turn Off Motor
               press = 0;                   // Reset the pressing counter
            }
          }
       }
}
```

// Int_INT0_b.c

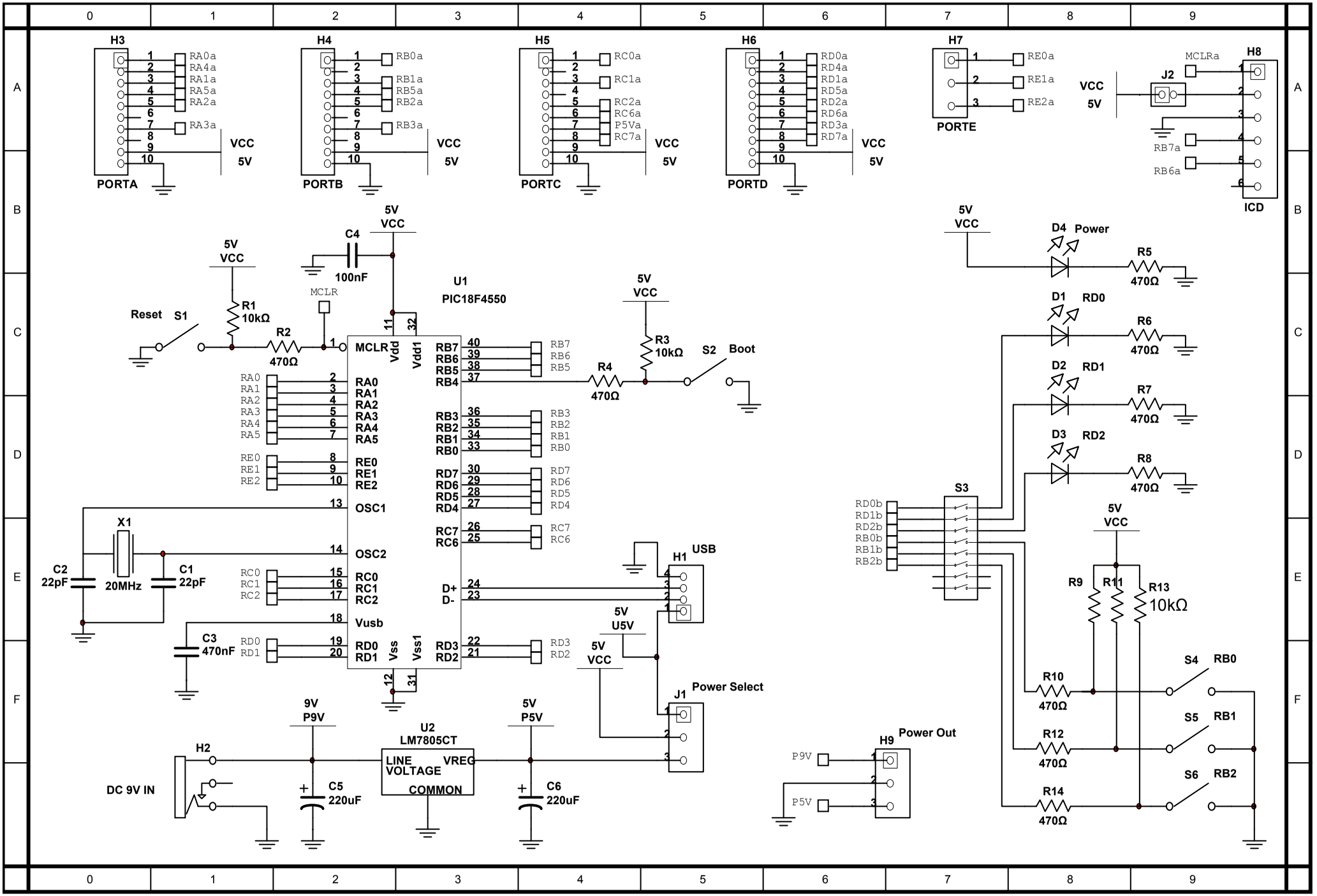
```
// Int_INT0_b.c Interrupt based program
// ISR activated by INT0 from an active low switch from RB0

#include <xc.h>
#include"delays.h"

unsigned char i, j;
unsigned char press, a, b;

void interrupt ISR_PortB0_low(void)     // Interrupt Service Routine for INT0
{
   if (INTCONbits.INT0IF)// External Interrupt Flag Bit = 1 when interrupt occurs
   {
       press++;             // To track first or second time pressing RB0 switch

       if (press == 1)              // First press
       {
          PORTCbits.RC2 = 1;        // Turn On Motor
       }
       else
       if (press == 2)              // Second press
       {
          PORTCbits.RC2 = 0;        // Else turn Off Motor
          press = 0;                // Reset the pressing counter
       }

    INTCONbits.INT0IF = 0;       //Clearing the flag at the end of the ISR
  }
}


void LED_RD7_RD0(void)// The function to shift a set-bit from the MSB to LSB
{
       j = 0x80;            // Initialise j with B1000 0000
                            // ie the leftmost bit (or MSB)

       while(j!=0x01)       // Check that the bit has not been shifted
                            // to the right-most bit (LSB) ie B00000001
       {
         PORTD= j;          // Display j at PORTD
         delay_ms(250);     // Calling a delay function from delays.h
         j = j>>1;          // Making use of LOGICAL-RIGHT-SHIFT bit-wise
       }                    // operator to shift data to the right

       PORTD = j;           // Display j at PORTD
}                           // Stop at B00000001
```

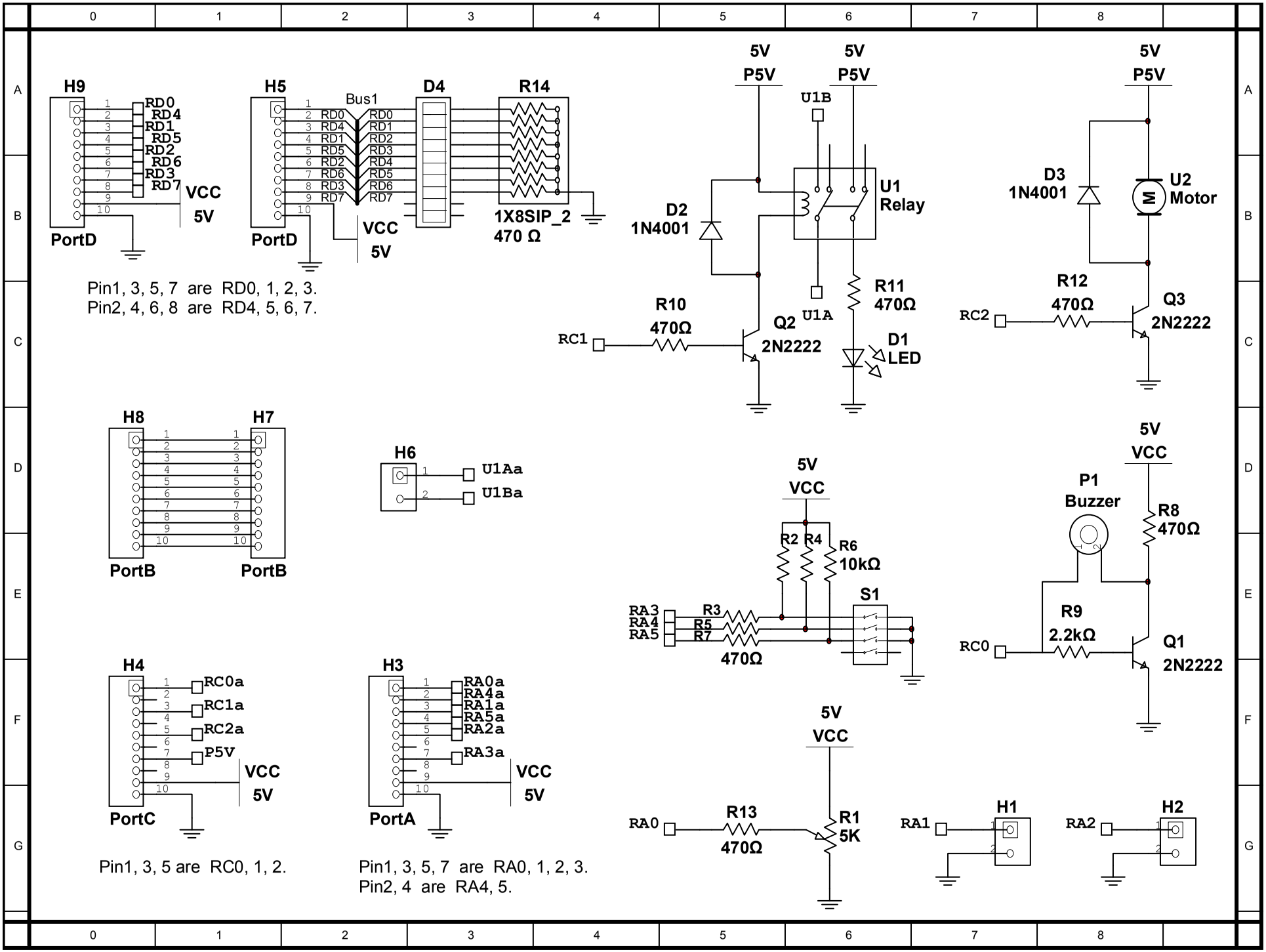
```
void main(void)             // Main Function
{

        ADCON1 = 0x0F;
        CMCON = 0x07;

        TRISBbits.TRISB0 = 1; // RB0 is the push button switch for INT0
        TRISCbits.TRISC2 = 0; // RC2 connects to a DC motor
        TRISD = 0x00;         // PortD connects to a bar LEDs

        PORTD = 0x00;         // LEDs all off
        press = 0;            // Not pressing yet
        j = 0;

        RCONbits.IPEN =1;     // Bit7 Interrupt Priority Enable Bit
                              // 1 Enable priority levels on interrupts
                              // 0 Disable priority levels on interrupts

        INTCONbits.GIEH =1; // Bit7 Global Interrupt Enable bit
                              // When IPEN = 1
                              // 1 Enable all high priority interrupts
                              // 0 Disable all high priority interrupts

        INTCON2bits.INTEDG0 = 0;// Bit4 External Interrupt2 Edge Select Bit
                              // 1 Interrupt on rising edge
                              // 0 Interrupt on falling edge

        INTCONbits.INT0IE = 1;  // Bit4 INT0 External Interrupt Enable bit
                              // 1 Enable the INT0 external interrupt
                              // 0 Disable the INT0 external interrupt

        INTCONbits.INT0IF = 0;  // Clearing the flag

        while(1)              // Main Process
        {
            LED_RD7_RD0();  // Move Port D LEDs from bit7 to bit0
        }
}
```

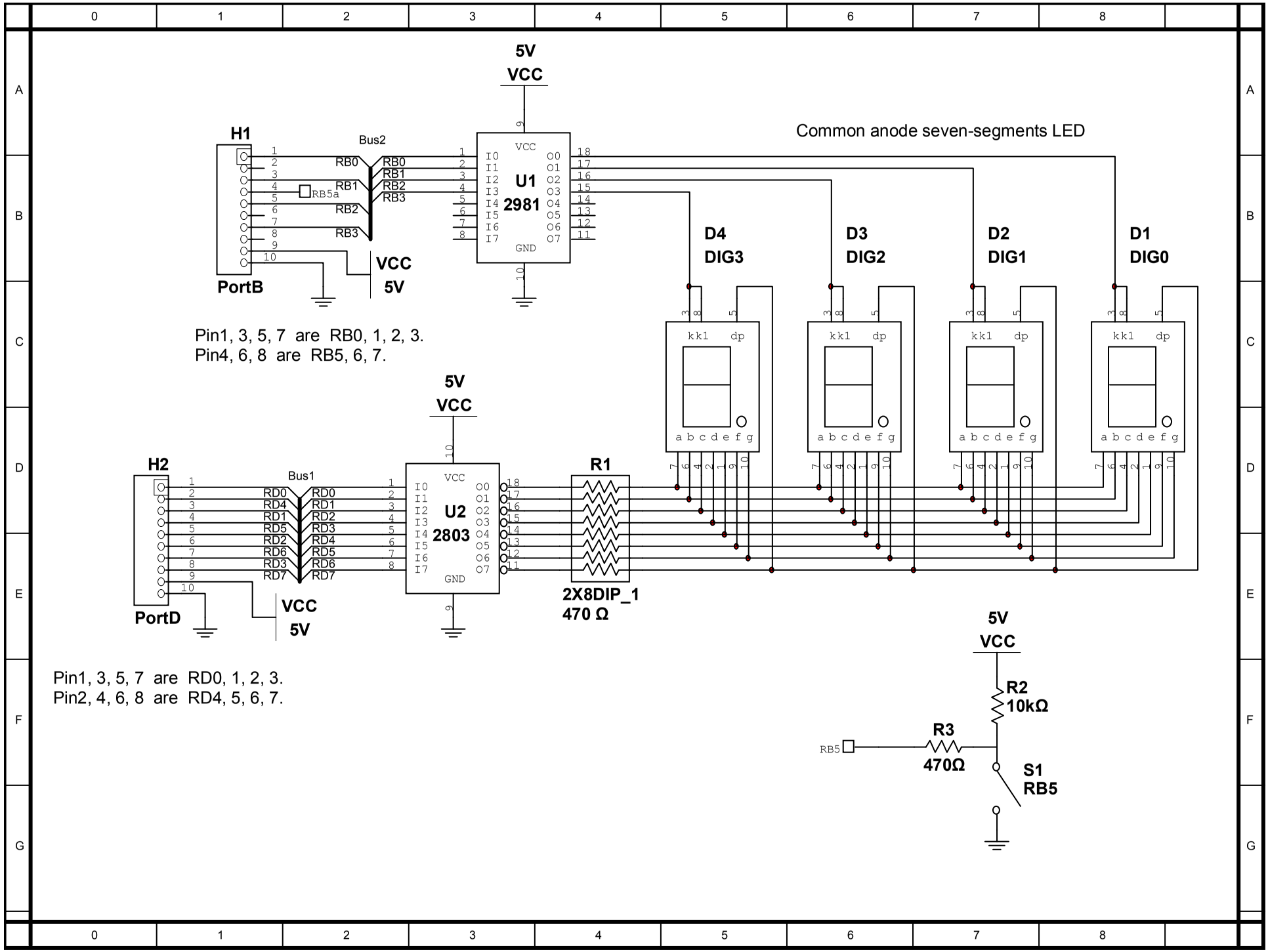
// Int_TMR0.c

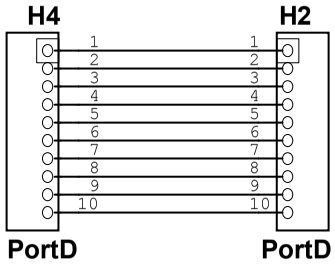
```
/* Int_TMR0.c Timer Interrupt based program
 * ISR activated by Timer0 interrupt when it over-flow
 * Set up a Timer0 interrupt-driven program to count up the LEDs at PORTD at
 * 1 sec interval
 *
 * Timer0 is configured for 16 bit Timer Mode operation.
 * The TMR0 interrupt is generated when the TMRO register
 * overflows from FFFFh to 0000h.
 * This overflow sets the TMR0IF bit.
 * The TMR0IF bit must be cleared in software by the Timer0 module
 * ISR before re-enabling this interrupt.
 *
 * Timer0 starting value is set by writing to TMR0H and TMR0L.
 * For 1 sec, the starting value is 0x48E5
 * Main Process - configure external interrupt and use RA3 switch to control
 ^ motor at RC2
*/

#include <xc.h>

unsigned char j;

void interrupt ISR_Timer0_Int()   // Timer0 Interrupt Service Routine (ISR)
{
  if (INTCONbits.TMR0IF)          // TMR0IF:- Timer0 Overflow Interrupt Flag Bit
                                  // 1 = TMR0 reg has overflowed
                                  // 0 = TMR0 reg has not overflowed
  {
        TMR0H = 0x48; // Timer0 start value = 0x48E5 for 1 second
        TMR0L = 0xE5;

        j++; // Increase count by 1
        PORTD = j; // Show count value at Port D Leds

       INTCONbits.TMR0IF = 0; // Reset TMR0IF to "0" since the end of
                              // the interrupt function has been reached
  }
}


void main(void)                   // Main Function
{

       ADCON1 = 0x0F;
       CMCON = 0x07;

       TRISAbits.TRISA3 = 1;      // RA3 is the On/Off switch
       TRISCbits.TRISC2 = 0;      // RC2 connects to a DC motor
       TRISD = 0x00;              // PortD connects to a bar LEDs

       PORTD = 0x00;              // LEDs all off
       j = 0;                     // Start count from 0

       RCONbits.IPEN =1;          // Bit7 Interrupt Priority Enable Bit
                                  // 1 Enable priority levels on interrupts
                                  // 0 Disable priority levels on interrupts


       INTCONbits.GIEH =1;        // Bit7 Global Interrupt Enable bit
                                  // When IPEN = 1
                                  // 1 Enable all high priority interrupts
                                  // 0 Disable all high priority interrupts

       T0CON = 0b00000111;        // bit7:0 Stop Timer0
                                  // bit6:0 Timer0 as 16 bit timer
                                  // bit5:0 Clock source is internal
                                  // bit4:0 Increment on lo to hi transition on TOCKI
```

```
                                // bit3:0 Prescaler output of Timer0
                                // bit2-bit0:111 1:256 prescaler


        INTCON2 = 0b10000100;   // bit7 :PORTB Pull-Up Enable bit
                                //     1 All PORTB pull-ups are disabled
                                // bit2 :TMR0 Overflow Int Priority Bit
                                //   1 High Priority

        TMR0H = 0x48;           // Initialising TMR0H
        TMR0L = 0xE5;           // Initialising TMR0L for 1 second interrupt

        T0CONbits.TMR0ON = 1;   // Turn on timer
        INTCONbits.TMR0IE = 1;  // bit5 TMR0 Overflow Int Enable bit
                                //0 Disable the TMR0 overflow int

        INTCONbits.TMR0IF = 0;  // bit2 TMR0 Overflow Int Flag bit
                                // 0 TMR0 register did not overflow

       while (1) // Main Process
       {
         if (PORTAbits.RA3 == 0)  // If RA3 switch is ON

             PORTCbits.RC2 = 1;    // Turn On Motor

          else

             PORTCbits.RC2 = 0;    // Else turn Off Motor

       }

}
```

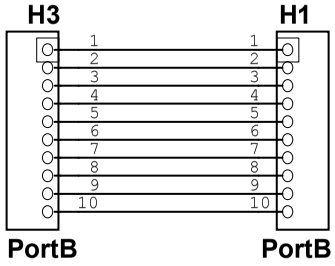

H3
PORTA
RA0a
RA4a
RA1a
RA5a
RA2a
RA3a
VCC
5V
H4
PORTB
RB0a
RB1a
RB5a
RB2a
RB3a
H5
PORTC
RC0a
RC1a
RC2a
RC6a
P5Va
RC7a
H6
PORTD
RD0a
RD4a
RD1a
RD5a
RD2a
RD6a
RD3a
RD7a
H7
PORTE
RE0a
RE1a
RE2a
H8
ICD
J2
MCLRa
RB7a
RB6a
U1
PIC18F4550
C4
100nF
R1
10kΩ
R2
470Ω
Reset
S1
MCLR
R3
10kΩ
R4
470Ω
S2
Boot
X1
20MHz
C2
22pF
C1
22pF
C3
470nF
Vusb
OSC1
OSC2
H1
USB
D+
D-
U5V
J1
Power Select
H2
DC 9V IN
9V
P9V
U2
LM7805CT
LINE VOLTAGE
VREG
COMMON
5V
P5V
C5
220uF
C6
220uF
H9
Power Out
D4
Power
R5
470Ω
D1
RD0
R6
470Ω
D2
RD1
R7
470Ω
D3
RD2
R8
470Ω
S3
R9
R11
R13
10kΩ
S4
RB0
S5
RB1
S6
RB2
R10
470Ω
R12
470Ω
R14
470Ω


H9
RD0
RD4
RD1
RD5
RD2
RD6
RD3
RD7
VCC
5V
PortD
H5
Bus1
D4
R14
1X8SIP_2
470 Ω
Pin1, 3, 5, 7 are RD0, 1, 2, 3.
Pin2, 4, 6, 8 are RD4, 5, 6, 7.
5V
P5V
U1B
U1
Relay
D2
1N4001
R10
470Ω
Q2
2N2222
RC1
U1A
R11
470Ω
D1
LED
D3
1N4001
U2
Motor
R12
470Ω
Q3
2N2222
RC2
H8
H7
PortB
H6
U1Aa
U1Ba
5V
VCC
R2
R4
R6
10kΩ
S1
RA3
RA4
RA5
R3
R5
R7
470Ω
P1
Buzzer
R8
470Ω
R9
2.2kΩ
Q1
2N2222
RC0
H4
RC0a
RC1a
RC2a
P5V
PortC
H3
RA0a
RA4a
RA1a
RA5a
RA2a
RA3a
PortA
Pin1, 3, 5 are RC0, 1, 2.
Pin1, 3, 5, 7 are RA0, 1, 2, 3.
Pin2, 4 are RA4, 5.
RA0
R13
470Ω
R1
5K
RA1
H1
RA2
H2

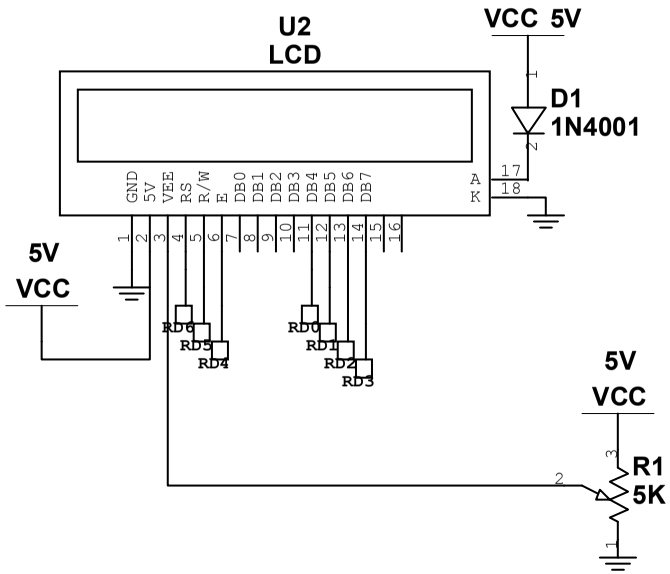

5V
VCC
H1
Bus2
RB0
RB1
RB2
RB3
RB5a
PortB
VCC
5V
U1
2981
Common anode seven-segments LED
D4
DIG3
D3
DIG2
D2
DIG1
D1
DIG0
kk1
dp
a b c d e f g
Pin1, 3, 5, 7 are RB0, 1, 2, 3.
Pin4, 6, 8 are RB5, 6, 7.
H2
Bus1
RD0
RD4
RD1
RD5
RD2
RD6
RD3
RD7
PortD
U2
2803
R1
2X8DIP_1
470 Ω
Pin1, 3, 5, 7 are RD0, 1, 2, 3.
Pin2, 4, 6, 8 are RD4, 5, 6, 7.
R2
10kΩ
R3
470Ω
RB5
S1
RB5

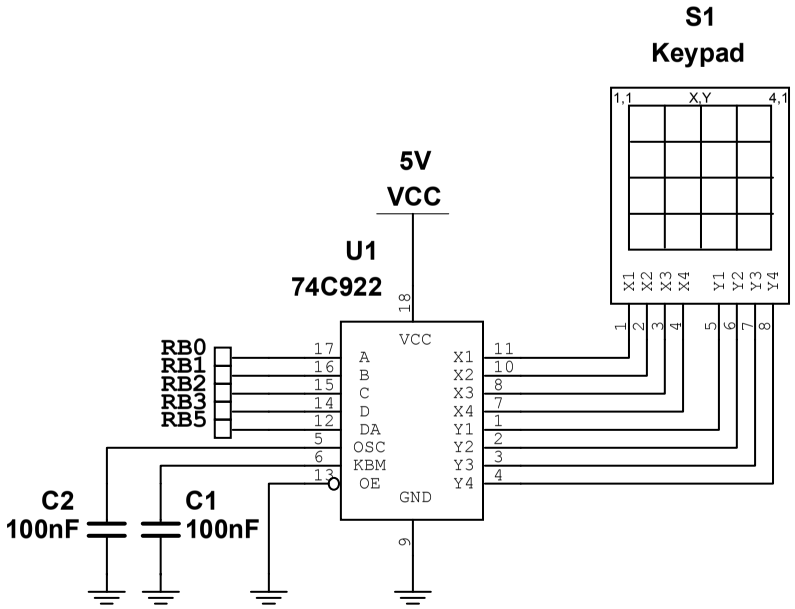


Pin1, 3, 5, 7 are RD0, 1, 2, 3.
Pin2, 4, 6, 8 are RD4, 5, 6, 7.



Pin1, 3, 5, 7 are RB0, 1, 2, 3.
Pin4, 6, 8 are RB5, 6, 7.





Microcontroller Applications

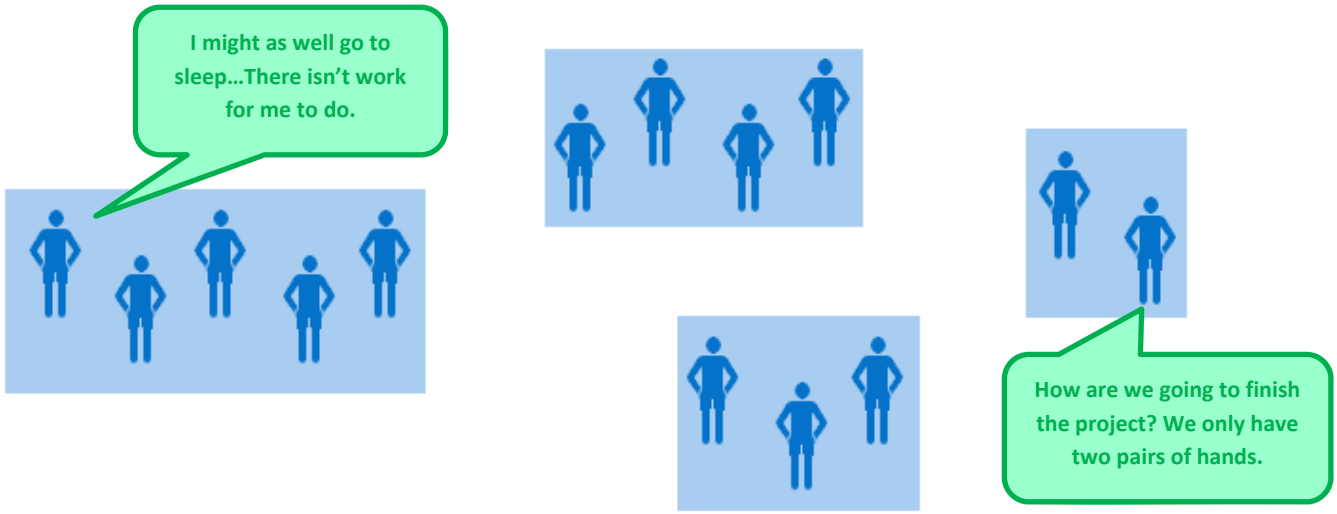
C-D-I-O Project Specifications

1. C-D-I-O Project

- All students taking the MAPP module are required to complete a C (conceive) – D (design) – I (implement) – O (operate) project, as specified in this document.

2. Project Teams

- At the beginning of the semester, the students in each class will be grouped into teams of 3 (min) or 4 (max). The lecturer taking the class can decide on how the grouping is done.



3. Basic Requirement

- The project must use a microcontroller.

MAPP project that does not use a microcontroller

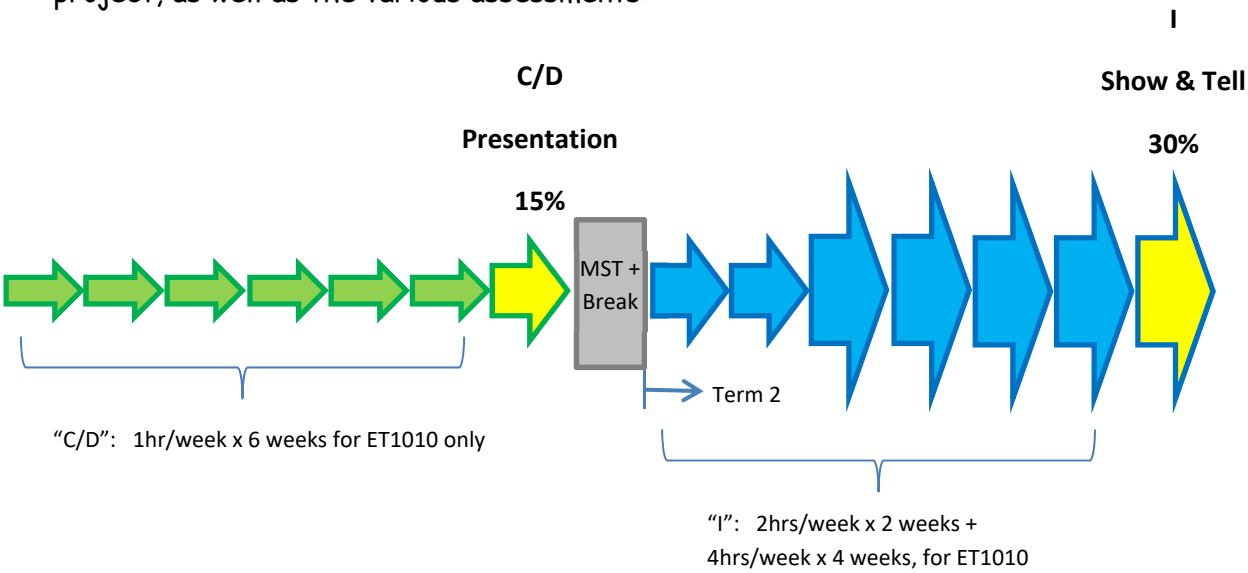
=

Cheese burger without cheese

4. Time Available & Module Assessments

- The project is to be done during the scheduled practical sessions (32 hours for MAPP, more than 4 weeks for ET0884) and contribute to 45% of the module assessment.

- The Schedule/Assessments in the Appendix shows the amount of time available for project, as well as the various assessments:



5. Conceive / Design (15% of module)

- Each project team will propose an interesting, useful (in solving a real problem) and original project idea. The project should also be feasible (i.e. doable by the students within the given time & cost constraints) and cost-effective.

Make sure your project does not end up like this...

Your project is "very interesting"...

This is a "perfect place" for your project...

Your project is "very original"...

- The lecturer taking the class may set a "project theme" e.g. "microcontroller applications that help the elderly".

- The theme can also be based on the course of study e.g. "Clean Energy applications for students taking DEEE", "Aerospace applications for students taking DASE" etc.

6. "Design Thinking" Tools for C/D

- Various tools can be used in the Conceive stage:

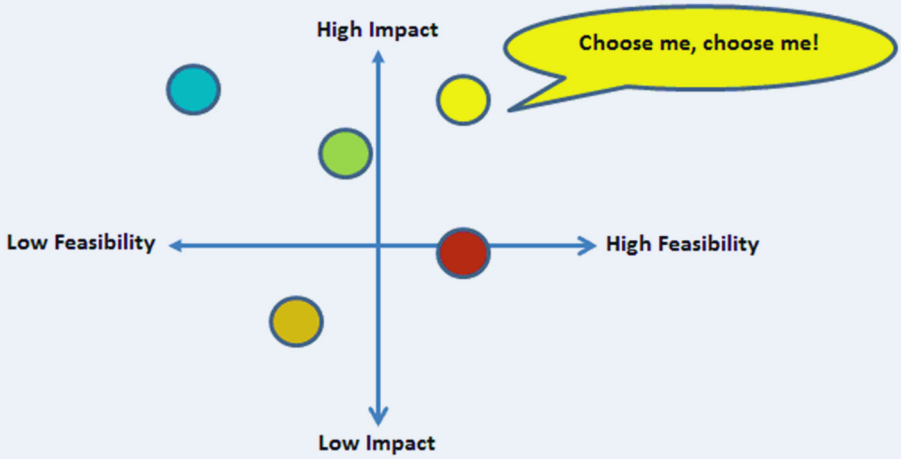
Journalist ("Interview") – Imagine you are a newspaper reporter. Interview the target users to find out what they need.

A-E-I-O-U ("Observation") – Observe the target users and take note of the activities, environment, interactions, objects and users.

- Other tools can be used in the Design stage:

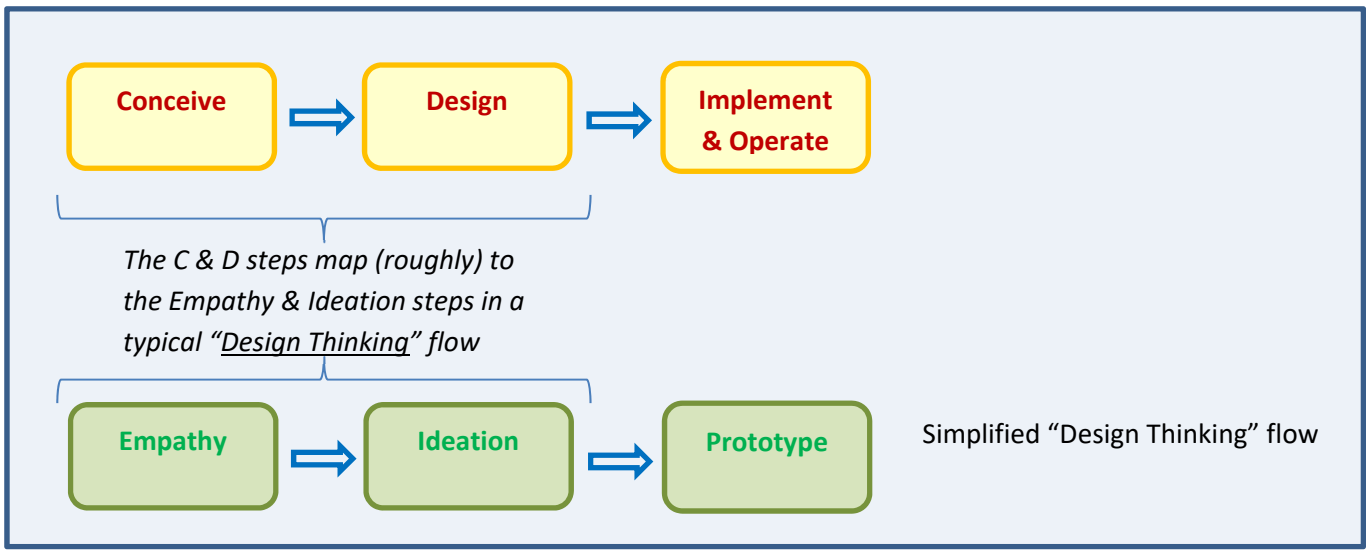
Brainstorming – Come up with as many ideas as possible, based on your interviews / observation. Draw simple sketches to capture the ideas.

Brainstorming



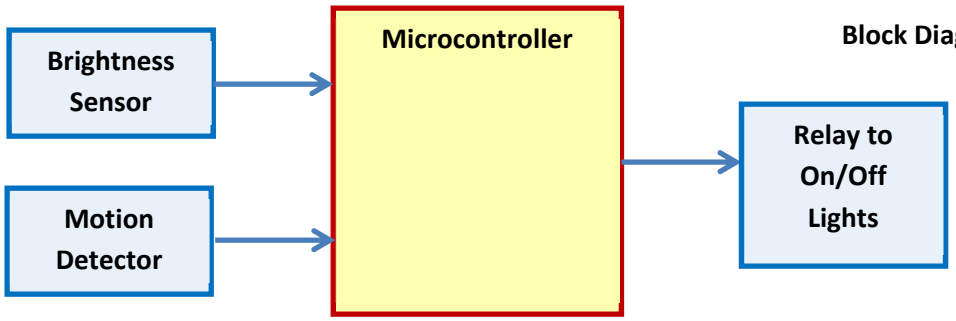
Matrix

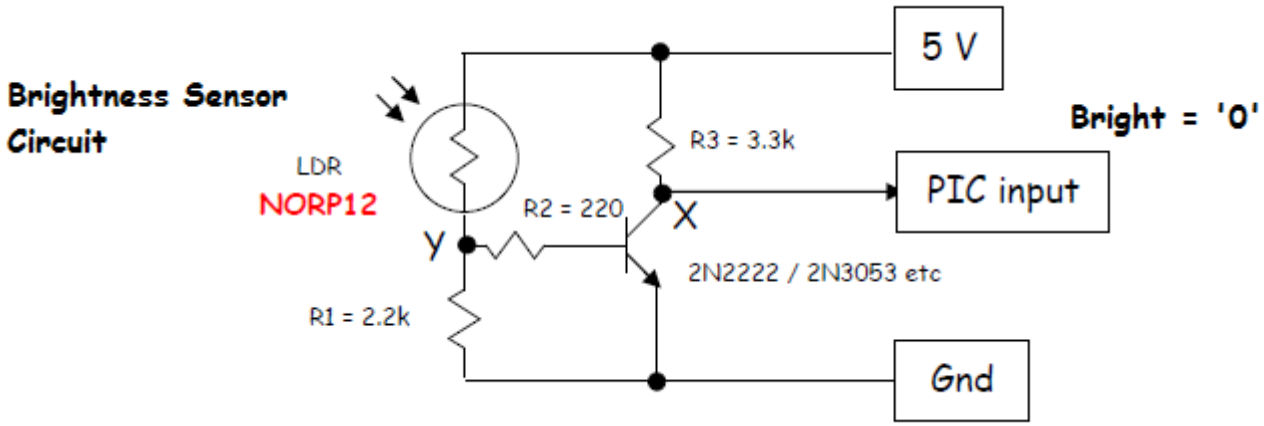
Matrix for Generation – Sort the ideas into 4 quadrants. Focus on high impact, high feasibility ideas.



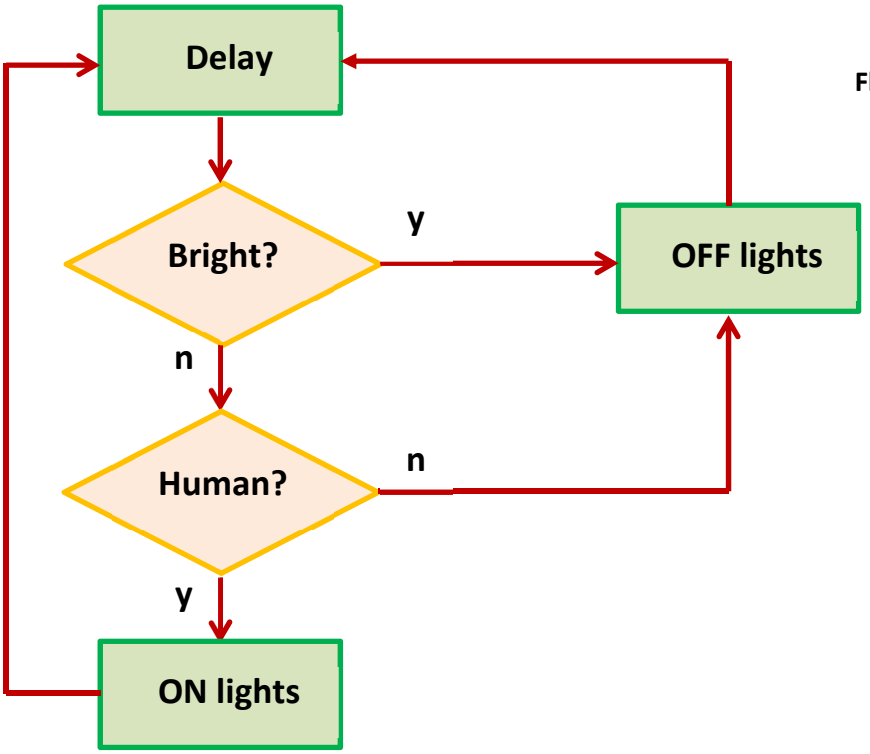
7. Block Diagram / Circuit Diagram / Flowchart

- After a project idea is selected, students will draw block diagram & circuit diagrams (for the hardware design) & flowchart (for the software design) and to plan an implementation schedule .

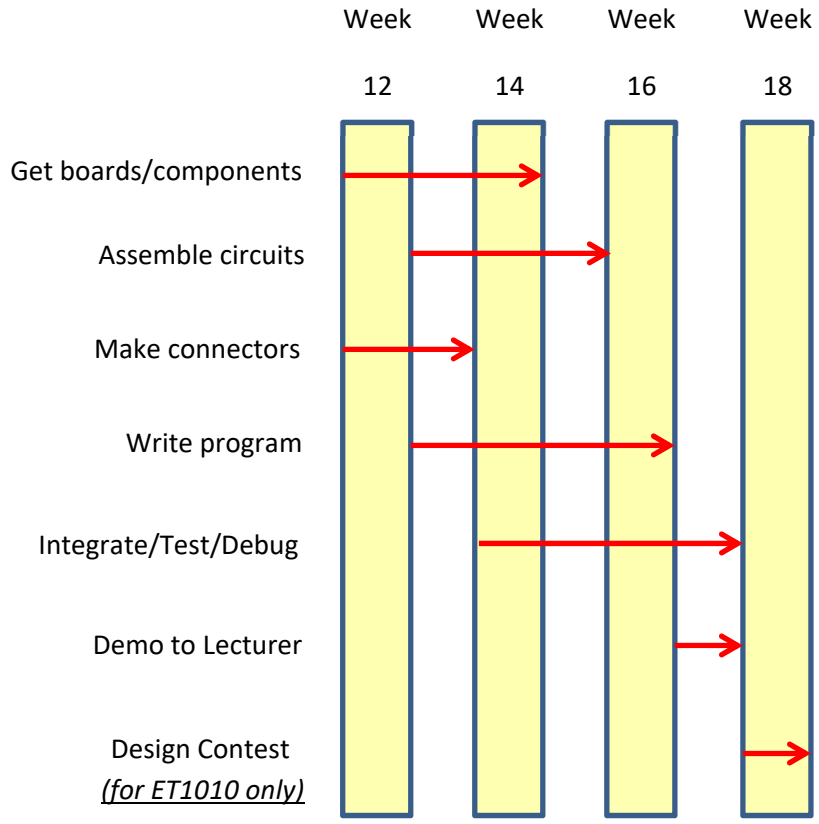




Flowchart



Implementation Plan



If you find it impossible to complete the project within the limited time given, you will need to scale down your project scope.

8. Presenting the Project Idea

- At the end of the "C/D" stages, students will make a 10-15 minute presentation of their work.

- In the presentation, describe the research/analysis of user needs done, describe how the application works, outline its key features, show the block diagram/circuit diagrams, flowchart, implementation plan etc. Use sketches, photos, video clips, models to demonstrate the project idea where appropriate.

9. "C/D" Assessment Criteria

Has the team done good research & analysis to understand user needs? __ / 25

Has the team come up with a creative idea / solution to the user's problem? __ / 25

Has the team shown that the project is feasible with the block diagram, circuit diagrams, flow-chart & implementation plan? __ / 25

Is the presentation lively & engaging? __ / 25

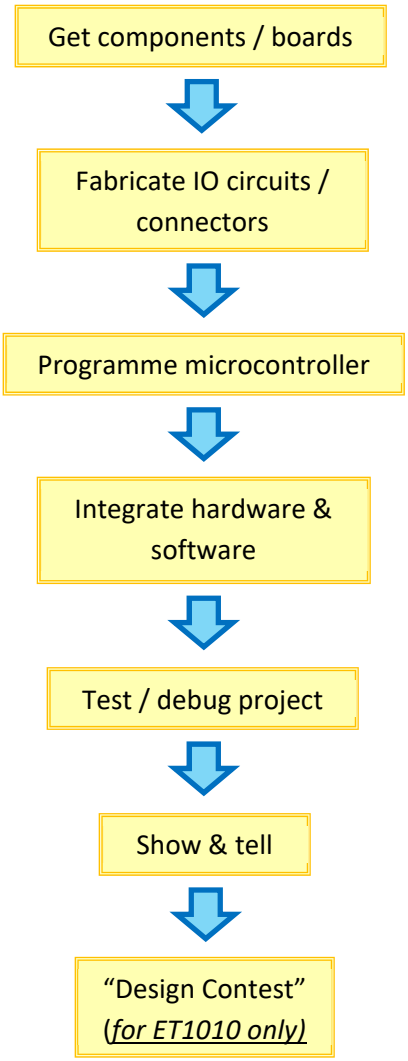
Although group marks can be awarded, individual students can score higher/lower, depending on his/her contribution.

Selected groups can be asked to present to a panel of industrial/academic judges (at the beginning of Term 2) and bonus marks will be awarded.

10. Implement (30% of module)

- In the second term, students implement their project ideas and then do a "show & tell" (i.e. a demo).

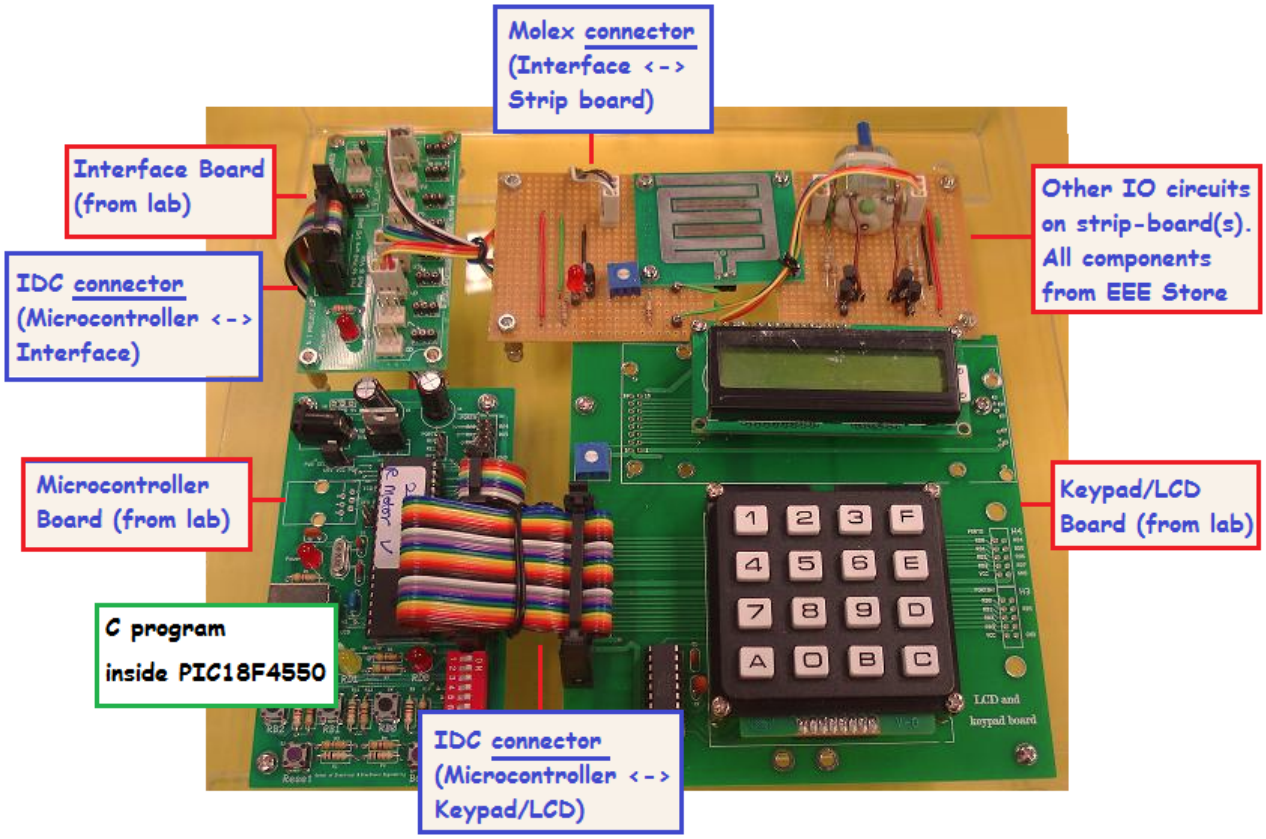
- The steps are outlined below:



11. Project resources

- The PIC18F4550 Microcontroller Board and the Keypad/LCD Board can be signed out from the lab. Likewise, the Interface Board (which allows other IO circuits to tap power from the Microcontroller Board) can be signed out from the lab.

- Students have to fabricate other IO circuits on strip boards, with components from the MAPP Lab. Write-ups on many IO devices (IR, LDR, Motion Sensor, Tilt Switch, DC Motor, Servo Motor Solenoid, Relay, Distance Measurement, Temperature Sensor, Moisture Sensor, Real-Time-Clock, Voice Recording/Playback) are in the Blackboard.



- Students have to provide their own notebook (with software installed), soldering tools & multi-meters etc.

- Students are also free to use other suitable I/O devices not available from the lab. However, they have to source & pay for these themselves.

- Students should take good care of items on loan, as these must be returned at the end of the semester. Students will have to replace any lost/damaged items.

12. Demonstrating the Project

- At the end of the "I" stage, students will do a 10-15 minute show & tell of their work.
- In the show & tell, outline the objective of the project – what is it supposed to do? Do a demo run, briefly described the technical difficulties involved.

13. "I" Assessment Criteria

Did the team put in effort in implementation? __ / 35

Is the project working i.e. a success? __ / 35

Is the show & tell lively & engaging? Can students answer questions asked? __ / 30

Although group marks can be awarded, individual students can score higher/lower, depending on his/her contribution.

Selected groups can be nominated to enter a "**Design Contest**" (at the end of Term 2). **Bonus marks** will be given to ALL participants. Certificates + Prizes will be given to the teams that come up with great idea, can do and can "sell".
-- for ET1010 only

CERTIFICATE OF PARTICIPATION
D&IP* "Design Contest" 2010
is presented to
Mediana
For her excellent achievement in
the Conceive-Design-Implement-Operate project

This is to show to amount of time available for the CDIO project.
Please refer to Blackboard for the actual schedule for this semester.

Appendix -CDIO schedule / assessments – MAPP

DTL / Tutorial	Practical / Lab	C-D-I-O Project & Design Thinking	Assessment

Wk	DTL / Tutorial	Practical / Project			
ET1010	1 hr DTL + 1 hr Tutorial	1st hour	2nd hour	3rd hour	4th hour
ET0884	2 hour Tutorial	Part I			Part II
1	C1 – **Intro**duction to micro-controller	Exp 1: Intro to PIC18F4550 Board, MPLAB-IDE, C-compiler, USB downloader [installation of software tools on students' notebooks]			Empathy/Ideation,**form team** ("Conceive/Design") [part 1:select **area of interest**]
2	C2 – Microchip's PIC18F**4550** – an **overview**	Exp 2: Interfacing to switches + LEDs			Empathy/Ideation: **observe** ("Conceive/Design") [part 2: prepare for **interview**]
3	C3 – PIC18F4550's **I/O ports** & **device interfacing**	Exp 3: Interfacing to 7-segment display + buzzer			Empathy/Ideation ("Conceive/Design") [part 3: **brainstorm**]
4	C3 cont.	Exp 4: Interfacing to keypad + LCD			Empathy/Ideation, ("Conceive/Design") [part 4: ***draft design & plan***]
5	C4 – PIC18F4550's **analogue to digital converter**	Exp 5: ADC (Analogue to Digital Converter) & interfacing to high power devices			**Work on** Conceive/Design **presentation slides**, Fill in **1st half of SDL form**
6	C5 – A brief "MPLAB-C18 **Compiler"** user's guide	Make up or revision			**Improve** Conceive/Design **presentation slides** Fill in **1st half of SDL form**
7	Presenting the project idea conceived **(Conceive/Design, CA3 -- 15%)** Make up or revision	**Lab Test (LAB 1 -- 15%)**		Presenting the project idea conceived **(Conceive / Design, CA3 -- 15%)**	
8-11	**MST(for ET1010) / Class Test 1(for ET0884) (20%) /** VACATION				
12	C6 – PIC18F4550's programmable **timer / counter**	Exp 6: Programmable timer & PWM (Pulse width modulation)		Students **request for components** from Lab. **sign out other resources/PCBs** etc.	
13	C6 cont.	Exp 7: Interrupt programming		Students start **implementation** of project (fabrication of I/O circuits on strip board, interfacing MCT board to I/O circuits. PIC programming, troubleshooting etc)	
14	C7 – PIC18F4550's **interrupt**	Students continue **implementation** of project			
15	C7 cont.	Students continue **implementation** of project			
16	C8 – PIC18F4550's **serial port (a brief intro)**	Students continue **implementation** of project. Students **fill in the 2nd half of page 1 SDL form**			
17	Revision	Students continue **implementation** of project, Student do **peer evaluation on page 2 of SDL form. Submit SDL form to lecturer**. Assessment of completed project. Short-listed projects will participate in the DesignContest (with bonus marks) -- for ET1010			
18	Make up	Assessment of the rest of the projects. **(Implement, CA4 -- 30%)** **CA1 (GP) and Design Contest (Wed 2—5pm), for ET1010 only** **Class Test 2, for ET0884 only**			

Quiz

Name: _____ **Adm No**: _____ C**lass:** D____/ 2_____

My Learning (Challenges and Strategies/ Resources to Be Used):

Challenges:

-
-

Strategies/ Resources to be used to overcome the above challenges:

-
-

Feedback Received From (your lecturer and peers) :

-
-

Student Signature:_____ Date: _____

(By the week before MST)

Follow-up Actions Done: (Key Challenges and Strategies/ Resources Used):

Key Challenge(s):

-

Key Strategies / Resources used helps overcome the above challenge(s) most:

-

Evidence / Result:

Student Signature:_____ Date: _____

(By the week of project interview)

Peer Evaluation for ---- Name: _____ Adm No: _____ C**lass:** D____/ 2_____

Peer evaluation from my team mates	**By Mid Semester**		**By End Semester**
	One thing I appreciate of this team member …	One thing I would like to request of this team member …	This team member has (✓ one of the options)
Team member 1:			☐ Improved greatly ☐ Improved slightly ☐ Not improved
Team member 2:			☐ Improved greatly ☐ Improved slightly ☐ Not improved
Team member 3:			☐ Improved greatly ☐ Improved slightly ☐ Not improved

ENGINEERING @ SP

The School of Electrical & Electronic Engineering at Singapore Polytechnic offers the following full-time courses.

1. Diploma in Aerospace Electronics (DASE)
The Diploma in Aerospace Electronics course aims to provide students with a broad-based engineering curriculum to effectively support a wide spectrum of aircraft maintenance repair and overhaul work in the aerospace industry and also to prepare them for further studies with advanced standing in local and overseas universities.

2. Diploma in Computer Engineering (DCPE)
This diploma aims to train technologists who can design, develop, setup and maintain computer systems; and develop software solutions. Students can choose to specialise in two areas of Computer Engineering & Infocomm Technology, which include Computer Applications, Smart City Technologies (IoT, Data Analytics), Cyber Security, and Cloud Computing.

3. Diploma in Electrical & Electronic Engineering (DEEE)
This diploma offers a full range of modules in the electrical and electronic engineering spectrum. Students from 2019/20 Year 1 intake can choose one of the six available specialisations (Biomedical, Communication, Microelectronics, Power, Rapid Transit Technology and Robotics & Control) for their final year. Students from earlier intakes and direct-entry 2nd year students can choose one of the seven available double-specialisation tracks (Aerospace + Communication, Biomedical + Robotics & Control, Computer + Communication, Microelectronics + Nanoelectronics, Microelectronics + Robotics & Control, Power + Control and Rapid Transit Technology + Communication) for their final year.

4. Diploma in Energy Systems & Management (DESM)*
The Diploma in Energy Systems & Management course aims to equip students with the knowledge and expertise in three specialisations: clean energy, power engineering and energy management, so as to design clean and energy efficient systems that will contribute to an economically and environmentally sustainable future.

5. Diploma in Engineering Systems (DES)*
The Diploma in Engineering Systems course aims to provide students with a broad-based engineering education to support activities and future challenges requiring interdisciplinary engineering systems capabilities. The course leverages on the experience and expertise of two schools, namely the School of Electrical & Electronic Engineering and the School of Mechanical & Aeronautical Engineering.

6. Diploma in Engineering with Business (DEB)
Diploma in Engineering with Business provides students with the requisite knowledge and skills in engineering principles, technologies, and business fundamentals, supported by a strong grounding in mathematics and communication skills, which is greatly valued in the rapidly changing industrial and commercial environment.

7. Common Engineering Program (DCEP)
In Common Engineering Program, students will get a flavour of electrical, electronics and mechanical engineering in the first semester of their study. They will then choose one of the 7 engineering courses specially selected from the Schools of Electrical & Electronic Engineering and Mechanical & Aeronautical Engineering.

*Course is applicable only for AY2018 intake and earlier

School of Electrical & Electronic Engineering

More than **60** Years of solid foundation	**8** Tech Hubs	Unique **PTN** Scheme	**SP-NUS** **SP-SUTD** Programmes	More than **35,000+** Alumni

SINGAPORE POLYTECHNIC SP

Electives offered by SP | SCHOOL OF ELECTRICAL & ELECTRONIC ENGINEERING

All SP students, including EEE students are free to choose electives offered by ANY SP schools, subject to meeting the eligibility criteria.

Like all schools, School of Electrical and Electronic Engineering offers electives for:

- EEE students only
- *and* for all SP students

EEE students are required to complete 3 electives, starting from Year 2 to Year 3 (one elective per semester).

Electives Choices for All SP students

Mod Code	Module Title
EP0400	Unmanned Aircraft Flying and Drone Technologies
EP0401	Python Programming for IoT*
EP0402	Fundamentals of IoT*
EP0403	Creating an IoT Project*
EP0404	AWS Cloud Foundations
EP0405	AWS Cloud Computing Architecture
EP0406	Fundamentals of Intelligent Digital Solutions

Certificate in IoT (Internet of Things)

* A certificate in IoT would be awarded if a student completes the 3 modules: EP0401, EP0402 and EP0403

Electives Choices for EEE students

Mod Code	Module Title
EM0400	Commercial Pilot Theory
EM0401	Autonomous Electric Vehicle Design
EM0402	Artificial Intelligence for Driverless Cars
EM0403	Autonomous Mobile Robots
EM0404	Smart Sensors and Actuators
EM0405	Digital Manufacturing Technology
EM0406	Linux Essential
EM0407	Advanced Linux
EM0408	Linux System Administration
EM0409	Rapid Transit System
EM0410	Rapid Transit Signalling System
EM0412	Data Analytics
EM0413	Mobile App Development
EM0414	Client-Server App Development
EM0415	Machine Learning & Artificial Intelligence
EM0416	Solar Photovoltaic System Design
EM0417	Energy Management and Auditing
EM0418	Integrated Building Energy Management System
EM0419	Digital Solutioning Skills
EM0422	Technology to Business
EM0423	Independent Study 1