

Chapter 3: Further Microcontrollers

Learning Objectives:

At the end of this topic you will be able to:

- recall and describe the structure of microcontrollers as programmable assemblies of:
 - memory;
 - input ports;
 - output ports;
 - CPU;
 - interrupts.
- recall and explain the use of interrupts to allow an external device to be serviced on request;
- recall and describe applications of a microcontroller;
- design and analyse microcontroller-based circuits;
- programme microcontrollers using the following assembler language instructions:

addlw;	andlw;
bcf;	bsf;
btfs;	btfs;
call;	clrf;
comf;	decfsz;
goto;	incf;
iorlw;	movf;
movlw;	movwf;
nop;	retfie;
return;	sublw.

Introduction to microcontrollers

Designing programs using flowcharts was covered in the AS course. This chapter extends that by looking at how to program microcontrollers using assembler language.

Investigations

It is expected that the student construct and test each of the sample programs, using any available assembler platform and hardware.

Both software and hardware packages are available from suppliers such as **Microchip**, **Matrix TSL** and **PICAXE**.

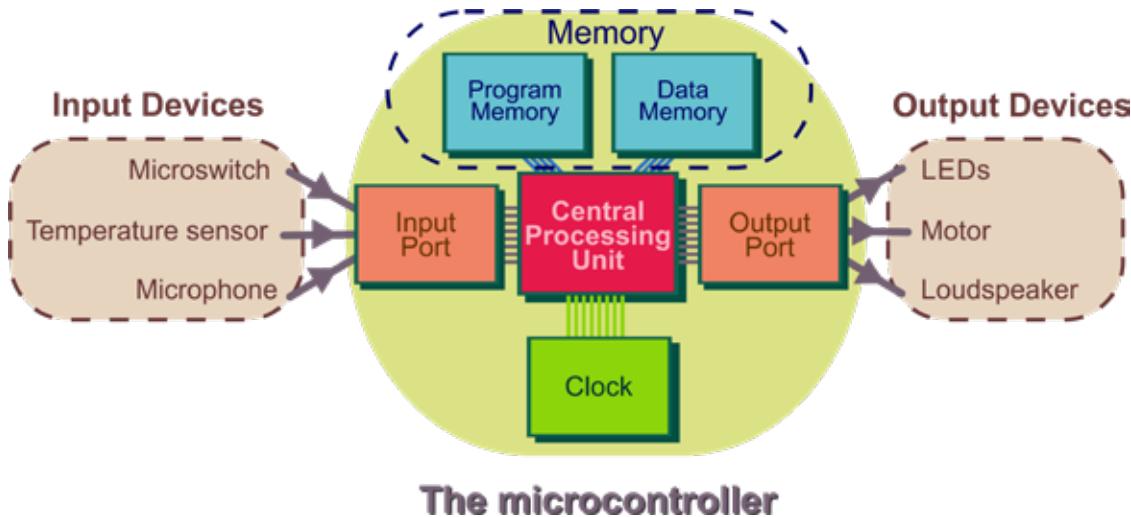
Note:

- Wherever possible, sample programs use port **A** as inputs and port **B** as outputs, allowing maximum flexibility with hardware choice.
- The **PICAXE-18M2** chip is emulating a 16F88 chip and so the chip does not operate at quite the same speed as a 16F88 programmed in raw assembler code. The pinout for port **B** is identical to the 16F88. There are some differences in the pinout for port **A**, so **PICAXE** users will need to be aware of this.

Structure of a microcontroller

A microcontroller is a digital integrated circuit, consisting of:

- central processing unit (CPU);
- memory;
- input and output ports.



The CPU processes the digital signals, does calculations and logic operations, creates time delays, sets up sequences of signals etc., following a program of instructions, stored in part of its electronic memory called the program memory.

Microcontrollers do exactly what they are told to do by the program, and nothing else! A program is a list of instructions, along with any data needed to carry them out. Its activities are synchronised by the clock, which sends a stream of voltage pulses into the CPU to control the execution of program instructions and the movement of data.

To talk to the outside world, the microcontroller has 'ports' that input or output data in the form of binary numbers. Each port has a number of connections - often referred to as 'bits'. An 8-bit port handles an 8-bit (or one byte) number.

Information from sensors is fed into the system through the input port(s). The microcontroller processes this data and uses it to control devices that are connected to the output port(s). The ports themselves are complex electronic circuits - not simply a bunch of terminals to hang components on.

Some microcontrollers incorporate analogue-to-digital converters that allow analogue sensors to be connected directly to an input.

Outputs can be interfaced to a microcontroller with a transistor or MOSFET, so even high-powered output devices can be accommodated easily.

Programming languages

In the world of microprocessors, there are a number of programming languages. Like all languages, these consist of a vocabulary (the mnemonics) and syntax (the 'grammar' used to link the mnemonics and data together to build a program of instructions). A program will work only if the vocabulary and syntax are totally error-free.

PIC microcontrollers are an extensive and important group of microcontrollers, present in a wide range of devices from DVD players to engine management systems. There are a number of ‘dialects’ of their programming language. In particular, MPASM is a common assembler language used to write programs for PICs.

Programs and instructions

A microprocessor processes digital data, following a sequence of instructions given in a program. The program is an instruction/data sandwich.

The basic structure is:

Instruction – do ‘this’ with the following data
Data
Instruction – now do ‘this’ with the next item of data
Data
and so on...

The microprocessor processes data in the form of binary numbers. It ‘understands’ a limited number of instructions, which it recognises because each is given a unique binary number.

The same idea could be used in some restaurants to order a meal:

“I’ll have a number 45, followed by a number 19 with a number 68 to finish, please.”

This works efficiently, because the chef has a recipe to follow for each of the numbered items on the menu. In the same way, the programmer tells the microprocessor to carry out a numbered instruction, such as 1010011_2 . The built-in instruction decoder lists the tasks to carry out to complete this instruction.

The collection of commands which can be used with a microprocessor is known as the instruction set, and it differs from one type of microprocessor to another, (in the same way as the numbering of items on a menu differs from restaurant to restaurant.)

In reality, programmers rarely write the program as a list of binary numbered instructions and data. Instead they use a mnemonic for each instruction – a word or abbreviation that suggests what the instruction does, for example *goto*, *clrf* (clear file) etc.

Ports

Microprocessors control real world devices such as motors, heaters and lights. They monitor external conditions like temperature, speed and light intensity. They must have the means to input data from, and output data to, the outside world. This is done through electronic sub-systems called ports.

A subsystem which imports data from the outside world to the microprocessor is called an input port. An output port sends data from the microprocessor to the outside world.

The PIC 16F88 microcontroller

This is one of the 18 pin PIC microcontroller range.
Its pinout is shown opposite.
(It does not include all functions of the pins.)

There are two ports.

- Port **A** has eight bits (**RA0/AN0** to **RA7**)
- Port **B** has eight bits (**RB0** to **RB7**)

The remaining two bits are:

- V_{ss} and V_{DD} , the power supply connections.

(The IC runs on a power supply voltage between 2V and 5.5V.)

Some port bits can be configured for alternative functions.

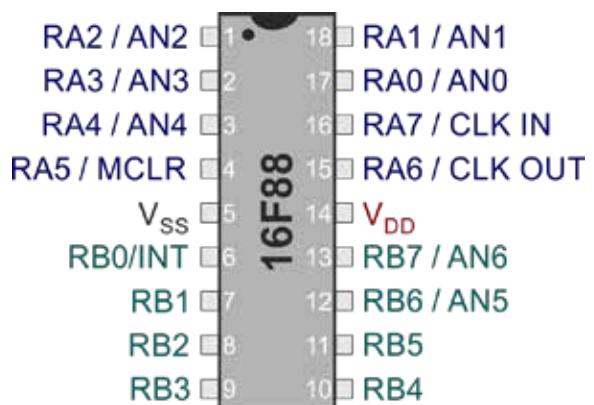
For example:

- Pin 4 (**RA5/MCLR**) can be used as an input or a master reset pin. It is ‘active low’, meaning that to reset the microcontroller, it is necessary to pull this pin down to 0V. Usually, this pin is connected to the positive supply rail, V_{DD} , through a resistor.
- Pins 15 and 16 (**RA6** and **RA7**) can be used to set up one of six external oscillator modes to provide clock pulses for the microcontroller.
- Pin 6 can be used to trigger an external hardware interrupt.(We will look at this in more detail later in this chapter).
- Several pins can be configured as analogue inputs, (**AN0**, **AN1** etc.) each with its own ADC

We make use of the following bits only as digital inputs/outputs

- Port **A** bits **RA0** to **RA2**;
- Port **B** bits **RB0** to **RB7**.

Later, we consider how each of these bits can be configured individually as inputs or outputs.



Investigation 3.1

Write a program called program 1, to output data to a single output.

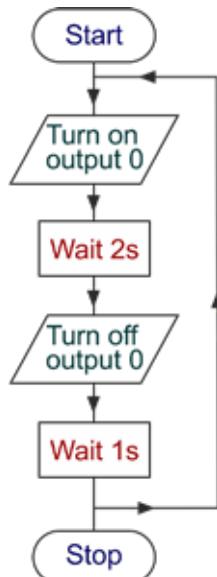
The program will:

- switch on a LED, attached to port **B** bit **0**, for two seconds;
- switch it off for one second;
- then repeat the process, over and over again.

Program 1:

(Assume that a one second delay subroutine called ‘wait1000ms’ is available.)

The flowchart:



The assembler code is:

```

        clrf    PORTB          ;make sure port B outputs are low at the start of
                                ;program
begin   bsf     PORTB,0      ;set bit 0 (the LSB) of port B (use label 'begin'
                                ; in PICAXE)
        call    wait1000ms    ;call the subroutine named 'wait1000ms' twice
        call    wait1000ms
        bcf    PORTB,0        ;clear bit 0 (the LSB) of port B
        call    wait1000ms    ;call the subroutine named 'wait1000ms' once
        goto   begin          ;branch to the part of the program labelled 'begin'
  
```

NB Do not set up and test program 1 until you have read pages 59-64, and then come back to answer the question below.

You can now test program 1 by entering the program (incorporated into the template) and assembling it using the assembler software package.

Comment on its performance in the space below:

.....

.....

.....

Analysing program 1:

The program is set out in four columns.

- The first is used for *labels*, such as ‘begin’.
- The second contains the instruction *mnemonic* (abbreviation), such as ‘*goto*’.
- The third contains the *data* relating to the instruction, such as ‘PORTB’.
- The fourth contains comments, that explain what the instruction does, such as ‘;clear bit 0 (the LSB) of port B’. Notice that a semicolon (;) marks the start of the comment. The comments written after the semicolons are ignored by the microcontroller.

The instructions:

clrf - (**clear file**) - clears (makes logic 0) all eight bit in the specified file register.

bsf - (**bit set file**) - sets (makes logic 1) the particular bit in the specified file register.

For example – **bsf PORTB,4** sets bit **4** of port **B**.

bcf - (**bit clear file**) This clears the particular bit in the file register specified.

For example - **bcf PORTA,2** clears bit **2** of port **A**.

call - causes the program to jump to a named *subroutine*, (a separate program section identified by a label.)

For example - **call tensconds** causes the program to jump to a subroutine labelled ‘tensconds’, which causes a delay of ten seconds before returning to execute the next line of the program.

goto - does what its name suggests - sends the processor to the address (or label) which follows.

For example - **goto test** causes the program to branch to the part of the program labelled ‘test’.

Although program 1 looks simple, there is a great deal of extra information that has to be provided before the microcontroller can execute it.

For example:

- the type of PIC being used;
- register names and addresses;
- standard subroutine listings;
- port configuration information.

This information can be provided on a program template.

Program 1 and related additional information is incorporated in the template given on the next four pages and is shown in bold red type.

```
;*****  
;  
;                                     TEMPLATE PROVIDED BY CENTRE  
;  
;          TITLE: Program 1  
;  
;          AUTHOR: #####YOUR NAME#####  
;  
;          DATE:    #####CURRENT DATE#####  
;  
;  
; Program description  
;  
;  
;The program flashes an LED on and off continuously  
  
;*****  
;  
;          DEFINITIONS  
;  
;*****  
list      p=16F88           ; tells the assembler which PIC chip to program for  
radix    dec                ; set default number radix to decimal  
;radix    hex                ; uncomment this to set radix to hex  
_config h'2007', 0x3F50   ; internal oscillator, RA5 as i/o, wdt off  
_config h'2008', 0x3FFF  
errorlevel -302           ; hide page warnings  
  
W        EQU h'00'           ; pointer to Working register  
F        EQU h'01'           ; pointer to file  
  
;***** REGISTER USAGE *****  
  
;For PIC16F88, user RAM starts at h'20'. The following definitions  
;will be found useful in many programs.  
  
; Register page 1  
TRISA      EQU h'85'       ; data direction registers  
TRISB      EQU h'86'  
OSCCON     EQU h'8F'       ; internal oscillator speed  
ANSEL      EQU h'9B'       ; ADC port enable bits  
  
; Register page 0  
STATUS     EQU h'03'       ; status  
PORTA     EQU h'05'       ; input / output ports  
PORTB     EQU h'06'  
INTCON     EQU h'0B'       ; interrupt control  
ADRESH    EQU h'1E'       ; ADC result  
ADCON0    EQU h'1F'       ; ADC control  
  
B0        EQU h'20'       ; general use byte registers B0 to B27  
B1        EQU h'21'  
B2        EQU h'22'  
B3        EQU h'23'  
B4        EQU h'24'  
B5        EQU h'25'  
B6        EQU h'26'  
B7        EQU h'27'  
B8        EQU h'28'  
B9        EQU h'29'  
B10       EQU h'2A'
```

```

B11          EQU h'2B'
B12          EQU h'2C'
B13          EQU h'2D'
B14          EQU h'2E'
B15          EQU h'2F'
B16          EQU h'30'
B17          EQU h'31'
B18          EQU h'32'
B19          EQU h'33'
B20          EQU h'34' ; used in interrupt routine
B21          EQU h'35' ; used in interrupt routine
B22          EQU h'36'
B23          EQU h'37'
B24          EQU h'38'
B25          EQU h'39'
B26          EQU h'3A'
B27          EQU h'3B'

WAIT1        EQU h'3C' ; counters used in wait delays
WAIT10       EQU h'3D'
WAIT100      EQU h'3E'
WAIT1000     EQU h'3F'
ADCTEMP      EQU h'40' ; adc loop counter

;***** REGISTER BITS *****

C            EQU h'00' ; carry flag
Z            EQU h'02' ; zero flag
RP0          EQU h'05' ; register page bit
INT0IF       EQU h'01' ; interrupt 0 flag
INT0IE       EQU h'04' ; interrupt 0 enable
GIE          EQU h'07' ; global interrupt enable

;***** VECTORS *****
;

;The PIC16F88 reset vectors

ORG  h'00'           ; reset vector address
    goto start         ; goes to first instruction on reset/power-up
ORG  h'04'           ; interrupt vector address
    goto interrupt

;

;***** SUBROUTINES *****
;

; Predefined wait subroutines - wait1ms, wait10ms, wait100ms, wait1000ms

wait1ms          ; (199 x 5) + 5 instructions = 1000us = 1ms @ 4MHz resonator
    movlw  d'199'      ; 1
    movwf  WAIT1       ; 1
loop5ns

```

```
    clrwdt          ; 1 this loop 1+1+1+2 = 5 instructions
    nop             ; 1
    decfsz WAIT1,F ; 1
    goto  loop5ns   ; 2
    nop             ; 1
    return          ; 2

wait10ms
    movlw  d'10'      ; 10 x 1ms = 10ms
    movwf  WAIT10

loop10ms
    call   wait1ms
    decfsz WAIT10,F
    goto  loop10ms
    return

wait100ms
    movlw  d'100'     ; 100 x 1ms = 100ms
    movwf  WAIT100

loop100ms
    call   wait1ms
    decfsz WAIT100,F
    goto  loop100ms
    return

wait1000ms
    movlw  d'10'      ; 10 x 100ms = 1000ms
    movwf  WAIT1000

loop1000ms
    call   wait100ms
    decfsz WAIT1000,F
    goto  loop1000ms
    return

; Predefined ADC subroutines – readadc0, readadc1, readadc2

readadc0
    movlw  b'00000001' ; setup mask for pin A.0
    call   readadc     ; do the adc conversion
    movwf  B0          ; save result in B0
    return

readadc1
    movlw  b'00000010' ; setup mask for pin A.1
    call   readadc     ; do the adc conversion
    movwf  B1          ; save result in B1
    return

readadc2
    movlw  b'00000100' ; setup mask for pin A.2
    call   readadc     ; do the adc conversion
    movwf  B2          ; save result in B2
    return
```

```

readadc
; generic sub routine to read ADC 0, 1 or 2 (pass appropriate mask in W)
; to start conversion we need mask (001, 010, 100) in ANSEL bits 0-2
; but the actual channel number (0, 1, 2) in ADCON0 channel select bits
; then set the ADCON0, GO bit to start the conversion

    bsf    STATUS,RP0      ; select register page 1
    movwf  ANSEL          ; move mask value 001,010,100 into ANSEL
    bcf    STATUS,RP0      ; select register page 0
    movwf  ADCTEMP        ; 00000??? get mask value
    rlf    ADCTEMP,F      ; 000???x rotate twice
    rlf    ADCTEMP,W      ; 000??xx
    andlw b'00011000'     ; 000??000 mask off the unwanted bits
    iorlw b'00000001'     ; 000??001 set the 'ADC on' bit
    movwf  ADCON0         ; move working into ADCON0
    movlw  d'10'           ; 10 x 3 = 30us acquisition time
    movwf  ADCTEMP        ; re-use ADC1 register as a counter

loopacq
    decfsz ADCTEMP,F    ; loop around to create short delay
    goto   loopacq        ; each loop is 1+2 = 3 instructions = 3us @ 4MHz
    bsf    ADCON0,2       ; now start the conversion

loopadc
    clrwdt              ; pat the watchdog
    btfsc   ADCON0,2     ; is conversion finished?
    goto   loopadc        ; no, so wait a bit more
    movf   ADRESH,W      ; move result into W
    return               ; return with result in W

;NOTE for PICAXE users: the following five subroutines and two instructions are not supported
;by PICAXE compiler
readtemp1:
readtemp2:
readtemp3:
debug:
lcd:
    clrw                 ; instruction not supported by this template
    return               ; instruction not supported by this template

;***** MAIN PROGRAM *****
;***** INITIALISATION *****

start
    bsf    STATUS,RP0      ; select register page 1
    movlw  b'01100000'     ; set to 4MHz internal operation
    movwf  OSCCON
    clrf   ANSEL          ; disable ADC (enabled at power-up)
    bcf    STATUS,RP0      ; select register page 0

;the data direction registers TRISA and TRISB live in the special register set. A '1' in
;these registers sets the corresponding port line to an Input, and a
;'0' makes the corresponding line an output.

```

```

Init
    clrf  PORTA          ; make sure PORTA output latches are low
    clrf  PORTB          ; make sure PORTB output latches are low
    bsf   STATUS,RP0       ; select register page 1
    movlw b'11111111'      ; set port A data direction (0 = output bit, 1 =
                           ; input bit)
    movwf TRISA           ;
    movlw b'11111110'      ; set port B data direction (0 = output bit, 1 =
                           ; input bit)
    movwf TRISB           ;
    bcf   STATUS,RP0       ; select register page 0

;***** PROGRAM *****
;***** remove semicolons from next two lines to enable interrupt routine*****
;     bsf      INTCON,INT0IE; set external interrupt enable
;     bsf      INTCON,GIE   ; enable all interrupts

main
;*****



        clrf  PORTB          ; make sure PORTB outputs are low at the start of
                           ; program
begin   bsf   PORTB,0        ; set bit 0 (the LSB) of PORTB
        call  wait1000ms      ; call the subroutine named 'wait1000ms' twice
        call  wait1000ms
        bcf   PORTB,0        ; clear bit 0 (the LSB) of PORTB
        call  wait1000ms      ; call the subroutine named 'wait1000ms' once
        goto begin            ; branch (unconditionally) to the part of the
                           ; program labelled 'begin'

;*****



;           INTERRUPT SERVICE ROUTINE
;*****



W_SAVE    EQU B2           ; backup registers used in interrupts

interrupt
    movwf W_SAVE           ; Copy W to save register

    btfss INTCON,INT0IF    ; check correct interrupt has occurred
    retfie                  ; no, so return and re-enable GIE

;*****The interrupt service routine (if required) goes here*****


    bcf   INTCON,INT0IF    ; clear interrupt flag
    movf  W_SAVE,W          ; restore W
    retfie                  ; return and re-set GIE bit

    END                     ; all programs must end with this

```

This template is about four pages long. We have added/modified only eleven lines. The template is so comprehensive so that it can be used for all programs. Some need only parts; more advanced programs require all of it.

At the moment, you might not understand its content, but that won't stop you writing programs.

There is a convention for entering numerical data in a program. Using this template, the system accepts numbers in decimal notation, by default.

- To enter a binary number, such as 11110000_2 , use the notation b '11110000'.
- To enter a hexadecimal number, such as $D9_h$, use the notation h'D9'.

Template structure

The template is broken down into a number of sections:

Title:

Contains the name of the programmer, the program title and a brief description of what the program does.

Definitions:

Control the assembler, telling it how to treat mnemonics and data, for example.

The **list** directive tells the assembler which PIC is being used, and which number system to accept as default, as the **radix** of the program.

Register usage:

Allows you to name general purpose registers, or even individual bits within registers, that you are going to use in the program.

The 16F88 PIC chip has fifty two special purpose registers and three hundred and sixty eight general-purpose registers. The PIC chip recognizes them by the number allocated to them.

Humans work better with names than with numbers, especially if the name reflects the purpose of the register. In a program, it is easier for the programmer to use the name '**PORTA**' than to remember that the hexadecimal number '05'.

Naming is done using the **equ** (equate) directive.

The line: PORTA EQU h'05'

tells the assembler to replace the name 'PORTA', wherever it appears in the program, with the number 05_h .

Register bits

Register bits allow you to use the ‘official’ names of certain bits detailed in the Special Function Registers, (SFRs) section of the data sheet.

The line: Z EQU h'02'

tells the assembler to replace the name ‘Z’, wherever it appears in the program with the number 05_h . In this case, ‘Z’ is the bit which contains the zero flag in the Status Register.

Similarly the line: RP0 EQU h'05'

refers to the register page bit, RP0.

Vectors

These tell the processor what to do when specific events happen.

For example, when the processor is switched on, or when it is reset, it needs to find the program it is supposed to run. To do this, it jumps to address 0000_h , (the **reset vector address**) and executes the first instruction it finds there. Usually, this is a *goto* command followed by the start address of the program. *Goto* does what its name suggests and sends the processor to the address (or label) which follows.

Similarly, if an interrupt occurs, (when, for example, the interrupt pin on the PIC chip receives the appropriate signal,) the processor jumps to address $0004h$ (the interrupt vector address.) There a *goto* command followed by the start address of the interrupt program sends it to the interrupt routine.

Subroutines

These are short sections of the program, used several times during the main program.

Rather than write the section out each time that it is used, it is written only once in an area of memory outside the main program. When needed, the processor is made to jump to the subroutine by the command *call*. The end of a subroutine is indicated by the command *return*, which makes the processor returns to the main program.

Standard subroutines such as delays or A to D conversion are included in this section of the template.

Main Program

The general structure of a program line is:

Label *Command* *Operand* ;*comments*

A *label* is used to mark a significant point in a program. You can instruct the processor to jump to a particular label if you want to repeat the section of the program that follows.

Note: In the PICAXE editor, all labels must be followed by a colon (:).

In MPASM, the editor recognises labels with or without a colon after it.

Command is the mnemonic for the particular instruction you want, and the *operand* is the specific data to be used with the instruction.

The *comment*, proceeded by the ‘;’ is the explanation of what you are doing at this point in the program.

The main program is split into two parts

i) Initialisation

sets up the input and output port configuration by writing the appropriate bit pattern into the data direction registers, known as **TRISA** and **TRISB**.

ii) Main

is where the program is inserted

Interrupt service routine

The section, which follows the main program, sets out the interrupt service routine, the program to be carried out if an interrupt is caused.

The start address of this routine is labelled **interrupt**, as used in the interrupt vector address.

The end of the interrupt service routine is marked by the instruction *retfie*, (return from interrupt.)

Configuring the ports:

Data is inputted and outputted through registers, called ports. The number of ports varies from PIC to PIC, but there are usually at least two, called port **A** and port **B**.

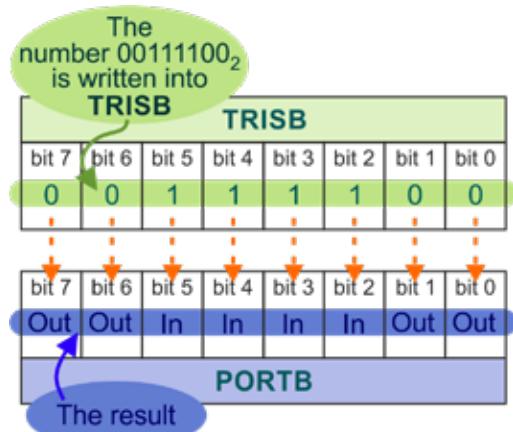
These are bi-directional, meaning that they can be set up either to input data or to output it. Control registers, called **TRISA**, for port **A** and **TRISB** for port **B**, determine whether the port is an input or an output, (or is a mixture of both.) Writing logic 1 into a bit of **TRISA** causes the corresponding bit of port **A** to input data. Logic 0 makes the corresponding bit output data. **TRISB** controls port **B** in the same way.

The next diagram illustrates this relationship.

Usually, ports are configured at the beginning of the main program by writing appropriate binary numbers into registers **TRISA** and **TRISB**.

Thereafter, the ports are left that way for the rest of the program. In other words, the only time you use **TRISA** and **TRISB** is at the beginning of the program.

In the PIC 16F88 microcontroller, **PORTA**, **TRISA**, **PORTB** and **TRISB** are all 8-bit registers.



Data memory is partitioned into four memory banks:

- **PORTA** and **PORTB** are in Bank 0;
- **TRISA** and **TRISB** are in Bank 1, in locations that mirror **PORTA** and **PORTB** in Bank 0.

Normally, programs use memory located in Bank 0. To move to Bank 1, a bit, called **RP0**, in the **STATUS** register is set (has logic 1 written into it.) To move back to Bank 0, that bit is cleared (has logic 0 written into it.) Memory banks are often referred to as *pages*.

Example

Write code to configure:

- all bits of port **A** as inputs;
- bits **0** to **3** of port **B** bits as inputs and bits **4** to **7** as output bits.

Use the following code:

```
bsf    STATUS,RP0      ; the following instructions will refer to Memory Bank1, and  
                      ; so affect the TRIS ;registers rather than the ports  
                      ; themselves  
  
movlw  b'11111111'    ; Set port A data direction to 'all I/P'  
  
movwf  TRISA          ; store number 11111111 in TRISA  
  
movlw  b'00001111'    ; 0 = output bit, 1 = input bit  
  
movwf  TRISB          ; store number 00001111 in TRISB  
  
bcf    STATUS, RP0     ; the rest of the program will refer to Memory Bank0, and so  
                      ; operate on the ports ;themselves rather than the TRIS  
                      ; registers.
```

At the beginning of a program, it is good practice to clear ports **A** and **B**. This was done in the template shown earlier but not in this segment of code, in case it leads to a misunderstanding.

Analysing this code:

The instructions *bsf* and *bcf* were considered earlier.

The first new instruction is:

- *movlw b'11111111'*

The mnemonic *movlw* means **move literal to working register**. Here, ‘literal’ means ‘number’. The working register is where most operations are carried out. In this case it is used as a temporary store for the number. The instruction loads it with the number 11111111_2 .

The second one is:

- *movwf TRISA*

Another instruction that moves data between file registers. The instruction *movwf* **moves** (copies) the content of the **working register** into the specified **file register**, (here **TRISA**.)

These two instructions together load the **TRISA** data direction register with 11111111_2 , to set up the all port **A** bits as inputs. (No **single** instruction loads a number directly into a file register, other than into the working register.) Instructions *movlw* and *movwf* are useful for transferring different binary patterns to the output port - usually quicker than repeated use of *bsf* and *bcf*.

Note:

Program 2 includes the instructions needed to configure the ports. For all future programs in this chapter, you need to configure ports **A** and **B** as appropriate.

Assembling a program

Once you have entered your program into the template on the assembler software package, it will be saved as ‘.asm’ file, which is simply a text file. This is often referred to as the source file.

The next step is to compile the assembly-language program. The compiler ‘reads’ the source file and lists any syntax errors, telling you the type of error and line number it occurs on. When free from syntax errors, the compiler generates a message that the compilation is successful.

Compilation converts the .asm file into binary instructions. For brevity, these are stored as a series of hexadecimal numbers in what is called a ‘.hex’ file. For a human, these values do not really mean anything, but this is the language that the microcontroller understands.

When a program is uploaded to a microcontroller, all labels and names are converted to corresponding hexadecimal values. If you attempt to upload a program without compiling it first, a message saying ‘Cannot find hex file’ will appear.

The hex file downloaded to the PIC for Program 1 is shown below:

Program1.hex

```
:020000040000FA
:020000000528D1
:0800080005288501860183161D
:100010009B018501860183125A28C730C000640005
:100020000000C00B0F28000008000A30C1000D209E
:10003000C10B172808006430C2000D20C20B1D2818
:1000400008000A30C3001B20C30B2328080001301E
:100050003320A000080002303320A1000800043043
:100060003320A20008008312C40083169B00831271
:10007000C40D440D183901389F000A30C400C40B68
:100080003F281F1564001F1942281E0883169B0174
:10009000831208004730CD001330CE000230CF006D
:1000A000CD0B5328CE0B5528CF0B50285728582856
:1000B00059280800850186018316FF30850000302D
:1000C000860083120B168B170614212021200610A0
:1000D00021206428B400030E8301B500350E83008F
:0600E000B40E340E09000D
:02400E00723FFF
:02401000FC3F73
:00000001FF
```

Investigation 3.2

Write a program called program 2, to output data to a multiple outputs.

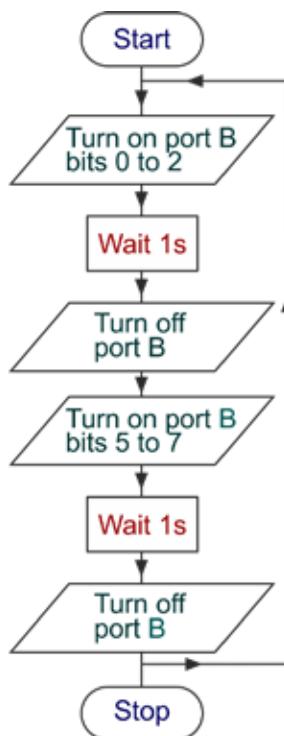
The program will:

- switch on three LEDs, attached to the three **least** significant bits of port **B**, for one second;
- switch off these LEDs and switch on three LEDs attached to the three **most** significant bits of port **B**, for one second;
- then repeat the process, over and over again.

Program 2:

(Again assume that a one-second delay subroutine called ‘wait1000ms’ is available.)

The flowchart:



Port **B** bits **0,1,2 5,6,7** are set up as outputs.

The port configuration for program 2 is written in the section of the template labelled **init**:

```

clrf PORTB      ; make sure port B output latches are low
bsf STATUS,RP0   ; select register page 1
movlw b'11111111' ; set port A data direction to all inputs
movwf TRISA
movlw b'00000000' ; set port B data direction to all outputs
movwf TRISB
bcf STATUS,RP0   ; select register page 0
  
```

The assembler code for program 2 is written in the section of the template labelled **main**:

```
begin    clrf  PORTB      ; make sure port B outputs are low at the start of program
        movlw b'00000111'   ; set up first binary pattern for port B (saves using bsf 3
                            ; times)
        movwf PORTB       ; store binary pattern in port B
        call  wait1000ms   ; call the subroutine named 'wait1000ms'
        movlw b'11100000'   ; set up second binary pattern for port B (saves using bsf 5
                            ; times)
        movwf PORTB       ; store binary pattern in port B
        call  wait1000ms   ; call the subroutine named 'wait1000ms'
        goto begin         ; branch (unconditionally) to the part of the program labelled
                            ; 'begin'
```

Test program 2 and comment on its performance below:

.....
.....
.....
.....
.....

The Instruction Set

The complete instruction set for the PIC16F88 microcontroller comprises 35 instructions. For the examination, you should be able to use and interpret the subset given in the following table:

Table 1

Mnemonic	Operand(s)	Description	Zero flag
<i>addlw</i>	k	Add working register to literal k	*
<i>andlw</i>	k	AND working register with literal k	*
<i>bcf</i>	f,b	Clear bit b of file register f	
<i>bsf</i>	f,b	Set bit b of file register f	
<i>btfsc</i>	f,b	Bit test bit b of file register f, - skip if clear	
<i>btfss</i>	f,b	Bit test bit b of file register f, - skip if set	
<i>call</i>	label	Call subroutine at label	
<i>clrf</i>	f	Clear file register f	*
<i>comf</i>	f,d	Complement file register f	*
<i>decfsz</i>	f,d	Decrement file register f - skip if zero	
<i>goto</i>	label	Unconditional Branch to label	
<i>incf</i>	f,d	Increment file register f	*
<i>iorlw</i>	k	Inclusive OR working register with literal	*
<i>movf</i>	f,d	Move file register f	*
<i>movlw</i>	k	Move literal to working register	
<i>movwf</i>	f	Move working register to file register f	
<i>nop</i>	-	No Operation	
<i>retfie</i>	-	Return from interrupt service routine and set global interrupt enable bit GIE	
<i>return</i>	-	Return from subroutine	
<i>sublw</i>	k	Subtract working register from literal k	*

* indicates where an instruction affects the zero flag (in the STATUS register). A non-zero result clears the zero flag to 0, and a zero result sets it to 1.

An instruction consists of a mnemonic and an operand.

- For instructions like *call* and *goto*, the operand is a label, (a marker inside the program.)

For example:

```
goto    restart      ; branch (unconditionally) to the part of the program labelled
                  ; 'restart'
call    second       ; call the subroutine named 'second'
```

- For instructions like *addlw* and *movlw* the operand is a number.

For example:

```
addlw  b'10000111'   ; add the binary number 10000111 to the number in the working
                      ; register
```

- For instructions like *clrf* and *movwf*, the operand is the name (or number) of a file register, either a Special Function Register (SFR) or a General Purpose Registers (GPR).

For example:

```
    movwf PORTB          ; move the contents of the Working register into the file  
                        ; register called port B  
    clrf  UNT           ; write logic 0 into all bits of the file register called COUNT
```

- Instructions such as, *bcf*, *bsf* and *btfs*, require the operand to specify both a file register name (or number) and the specific bit of that register that is affected.

For example:

```
    bsf    PORTA,0        ; set bit 0 (the LSB) of port A
```

- The *nop* instruction - (no operation) - does nothing, but takes one clock cycle to do so.

For example:

In the ‘wait1ms’ delay subroutine in the template, *nop* is used to trim the delay to exactly one millisecond.

For instructions like *movf*,(move the contents following file) the operand gives the name (or number) of the file register and the destination, (working register or file register,) for the result of the instruction.

For example:

```
    movf    PORTA,0        ; move the contents of port A into the working register
```

or alternatively:

```
    movf    counter,1        ; move the contents of the file called counter into itself.  
                            ; Although apparently pointless, this instruction will test  
                            ; whether the file ‘counter’ contains zero. If it does, this  
                            ; instruction will set the zero flag in the STATUS register.  
                            ; Otherwise, the zero flag will not be set
```

Note:

The **equ** (equate) directives **W EQU h'00** and **F EQU h'01** provided in the template, allow:

movf PORTA,0 to be written as **movf PORTA,W**

and **movf counter,1** to be written as **movf counter,W**.

You need to be familiar with both versions, but the sample programs will use *W* and *F* rather than *0* and *1* as they are easier to interpret.

The instruction *btfs* (bit test file, skip if set) checks the particular bit of the specified file, and if it is set (logic1), the processor jumps over the next instruction. If the bit is cleared (logic 0) then the next instruction is carried out.

For example:

```
wait      btfss  PORTA,0      ; If bit 0 of port A is clear (logic 0), then the "goto wait"
          ; instruction
          goto    wait      ; is carried out and the program continually loops back until
          ; bit 0 is set bcf PORTB,3
          ; If bit 0 of port A is set, then the "goto wait" instruction
          ; is skipped and the subsequent
          ; instruction is carried out – in this case, bit 3 of port B
          ; is cleared)
```

The instruction **btfsC** (**b**it **t**est **f**ile, **s**kip if **c**lear) is very similar to **btfsS** except that the processor jumps over the next instruction if the chosen bit of the specified file is **cleared**

- The instruction **comf** (complement file) complements (inverts) the contents of a file register.

For example:

```
comf    PORTB,F      ; inverts the output pattern in port B
```

Suppose that port **B** contains the binary number 10101010_2 .

After the **comf** instruction, it contains 01010101_2 .

```
comf    PORTB,W      ; the complement of the contents of port B is stored in the
                      ; working register
```

Suppose that port **B** contains 10101010_2 before the **comf** instruction.

After the instruction is executed, it still contains 10101010_2 but the working register will contain 01010101_2 .

Investigation 3.3

Write program 3 that waits for a switch to be pressed.

The program:

- waits until a switch connected to the input port, is closed;
- then repeats the process described in Program 1 until the switch is released.

Connections:

- the switch is connected to bit **1** of the input port;
- the LED is connected to bit **0** of the output port.

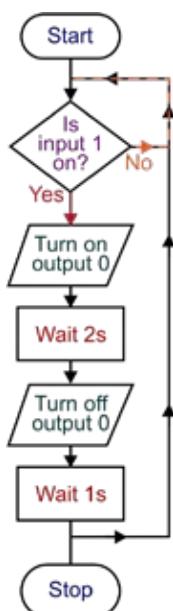
Note:

If you are using the PICAXE trainer board, it might be easier to test either bit **6** or **7** of port **A**.

Program 3:

(Again assume that a one-second delay subroutine called 'wait1000ms' is available.)

The flowchart:



The assembler code is:

```

      clrf  PORTA          ; make sure port A bits are low at the start of program
      clrf  PORTB          ; make sure port B outputs are low at the start of program
begin   btfss  PORTA,1    ; test bit 1 of port A
        goto  begin         ; if result equal to 0 program goes to this line then loops
                                ; back
        bsf   PORTB,0       ; if result equal to 1 program goes to this line
                                ; and of Port B sets bit 0 (the LSB)
        call  wait1000ms    ; call the subroutine named 'wait1000ms' twice
        call  wait1000ms
        bcf   PORTB,0       ; clear bit 0 (the LSB) of port B
        call  wait1000ms    ; call the subroutine named 'wait1000ms' once
        goto  begin         ; branch to the part of the program labelled 'begin'
  
```

Test program 3 and comment on its performance:

.....
.....
.....
.....
.....
.....
.....

What happens if the switch connected to bit 1 is held down?

.....
.....
.....
.....
.....
.....
.....

Investigation 3.4

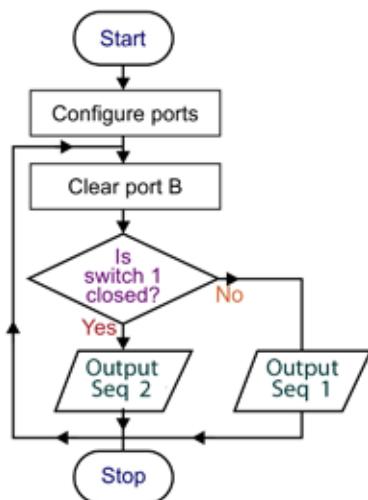
Write program 4 that changes the output sequence when a switch is pressed.

The program modifies program 3 to provide two sequences:

- The sequence described in Program 3 is generated on power-up;
- When the switch is pressed:
 - LEDs connected to the four most significant bits of port **B** light for 0.2 seconds;
 - then LEDs connected to the four least significant bits light for 1 second;
 - the process then repeats, over and over again.

Program 4:

The flowchart



The assembler code is:

```

begin    clrf  PORTA          ; make sure port A bits are low at the start of program
        clrf  PORTB          ; make sure port B outputs are low at the start of program
        btfss PORTA,1         ; test bit 1 of port A
        goto  seq1           ; if result equal to 0 program goes to seq1
        goto  seq2           ; if result equal to 1 program goes to seq2
seq1     bsf   PORTB,0        ; sets bit 0 (the LSB) of port B
        call  wait1000ms     ; call the subroutine named 'wait1000ms' twice
        call  wait1000ms
        bcf   PORTB,0        ; clear bit 0 (the LSB) of port B
        call  wait1000ms     ; call the subroutine named 'wait1000ms' once
        goto  begin          ; branch to the part of the program labelled 'begin'
seq2     movlw b'11110000'    ; set up pattern for seq2
        movwf PORTB          ; send pattern to port B
        call  wait100ms       ; call the subroutine named 'wait100ms' twice
        call  wait100ms
        comf  PORTB,F        ; invert output pattern
        call  wait1000ms     ; call the subroutine named 'wait1000ms'
        goto  begin          ; branch to the part of the program labelled 'begin'
  
```

Test program 4 and comment on its performance.

Suggest how it could be improved.

Exercise 3.1:

1. Write code to:

- select Bank 1;
 - configure bits **0, 1** and **2** of port **A**, and bits **0, 1, 2, 3** and **4** of port **B** as input bits, and all other bits as output bits.
 - select Bank 0.

2. Observing correct syntax, write the instructions needed to:

(a) set bit 3 of port A;

(b) clear bit 1 of port B;

(c) clear the file register called testfile;

(d) move the number 11011_2 into the working register;

(e) move the contents of the working register into a file register called cost;

(f) move the contents of the file register called *cost* into the working register;

(g) branch unconditionally to a point in the program identified by the label repeat;

(h) test bit 2 of the file register called input, and skip the next instruction if the bit is set.

Include File.

Using an *include* directive at the beginning of a program removes most common sub-routines and equate statement register usage information from the project template.

As long as this file is contained in the same folder as your programs, each program can access it to obtain the relevant data as required. The include file is provided in appendix 1. The instruction #include "Header.inc" must be added to the template.

The result is a template which is considerably smaller than the original version:
Program 1 has been included in red as with the full version of the template.

```
;*****  
;  
; TEMPLATE PROVIDED BY CENTRE FOR USE WITH HEADER FILE  
;  
; TITLE: Program 1  
;  
; AUTHOR: #####YOUR NAME#####  
;  
; DATE: #####CURRENT DATE#####  
;  
;  
; Program description  
;  
;  
;The program flashes an LED on and off continuously  
  
;*****  
;  
; DEFINITIONS  
;  
;*****  
;The header file can be used with either the PIC16F88 or PIC16F877A  
;  
#define UseRC ; UseRC, UseCrystal or UseInternal - defaults to UseRC  
;  
#define UseInternal ; UseRC, UseCrystal or UseInternal - defaults to UseRC  
;  
#define UseInterrupt ;  
;  
#define Watchdog ; The Watchdog timer are off by default unless this is uncommented  
  
#include "Header.inc"  
radix dec ; set default number radix to decimal  
;  
;radix hex ;uncomment to change default to hex  
  
;*****  
;  
; VARIABLES  
;  
;*****  
;  
; ##### Create any variables here using the EQU statement  
;  
; ##### Locations H'20' to H'3F' are available for your own use  
;  
; ##### e.g. MYVAR EQU H'20'  
  
;*****  
;  
; MAIN PROGRAM  
;  
;*****  
  
;***** INITIALISATION *****  
init  
    clrf PORTA ; make sure port A output latches are low  
    clrf PORTB ; make sure port B output latches are low  
    bsf STATUS,RP0 ; select register page 1  
    movlw b'11111111' ; set port A data direction (0 = output bit, 1 = input bit)  
    movwf TRISA ;  
    movlw b'11111110' ; set port B data direction (0 = output bit, 1 = input bit)
```

```
    movwf  TRISB          ;  
    bcf    STATUS,RP0      ; select register page 0  
  
;***** PROGRAM *****  
  
;***** remove semicolons from next two lines to use interrupt routine*****  
;bsf     INTCON,INT0IE; set external interrupt enable  
;bsf     INTCON,GIE   ; enable all interrupts  
  
main  
    clrf  PORTB          ; make sure PORTB outputs are low at the start of program  
begin  bsf   PORTB,0       ; set bit 0 (the LSB) of PORTB  
        call  wait1000ms    ; call the subroutine named 'wait1000ms' twice  
        call  wait1000ms  
        bcf   PORTB,0       ; clear bit 0 (the LSB) of PORTB  
        call  wait1000ms    ; call the subroutine named 'wait1000ms' once  
        goto begin          ; branch (unconditionally) to the part of the program labelled  
                           ; 'begin'  
*****  
;  
;           INTERRUPT SERVICE ROUTINE  
*****  
W_SAVE EQU B20      ; backup registers used in interrupts  
  
interrupt  
    movwf W_SAVE          ; Copy W to save register  
    btfss INTCON,INT0IF   ; check correct interrupt has occurred  
    retfie                ; no, so return and re-enable GIE  
  
;*****The interrupt service routine (if required) goes here*****  
  
    bcf   INTCON,INT0IF   ; clear interrupt flag  
    movf  W_SAVE,W        ; restore W  
    retfie                ; return and re-set GIE bit  
  
    END                  ; all programs must end with this
```

Investigation 3.5

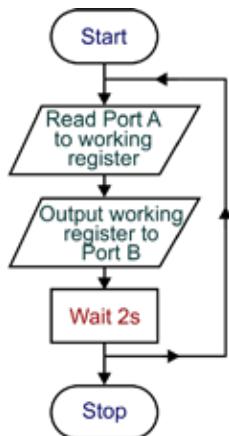
Write program 5 that uses data from the inputs to control the output sequence.

The program:

- monitors three emergency exits in a theatre by checking switches attached to each;
- identifies the opened door by lighting the appropriate LED on a display;
- then repeats the process over and over again;
- updates the display every two seconds.

Program 5:

The flowchart:



The assembler code is:

```

      clrf  PORTA           ; make sure port A bits are low at the start of program
      clrf  PORTB           ; make sure port B outputs are low at the start of program
TEST   movf  PORTA,W       ; move the contents of port A into the working register
        movwf PORTB          ; move the contents of the Working register into port B
        call   wait1000ms     ; wait 2s
        call   wait1000ms
        goto  TEST            ;branch to label 'TEST' to retest inputs
  
```

Test program 5 and comment on its performance. Suggest how it could be improved.

.....

You may find that some of the **unused** port B output LEDs come on and stay on. Obviously the constructed circuit for this program would use only three LEDs so there would be no problem. Sometimes, however, this would be a major issue. Program 7 includes a possible solution to this problem.

For now, try configuring the unused bits in port **A** as outputs and the unused bits in port **B** as inputs. Comment on how well this temporary ‘fix’ worked.

Investigation 3.6

Write program 6 that modifies program 5 by using a longer update time, 8 seconds.

Information

To increase the delay in program 5 to eight seconds, the `call wait1000ms` instruction could be written eight times, but that would be tedious.

Where programs include several long delays, this approach would be very cumbersome. A better approach is to use ‘loop control’ with the instruction `decfsz` (**d**e**c**rement **f**ile, **s**kip if **z**ero).

This command decrements (reduces by 1) the number stored in the specified file register. If the result of the subtraction is zero, it skips over the next instruction. If the result is not zero, the next instruction is not skipped, but is carried out.

For example:

```
temp      EQU B1      ; use reg B1 to store the count 'temp'
movlw     5          ; move decimal 5 into temp register
movwf     temp       ; this controls the number of times program loops around
                    ; "test" (5 times)
again    call      wait100ms ; wait 100ms
        decfsz  temp,F   ; subtract 1 from temp and store result in temp is result
                    ; zero?
        goto     test     ; no, so keep checking
        nop      ; yes, so continue to next instruction
```

The result is a 500ms delay

Program 6:

The assembler code is:

```
COUNT    EQU B2      ; use reg B2 to store the count

TEST     clrf      PORTB    ; make sure port B outputs are low at the start of program
        movf      PORTA,W ; move the contents of port A into the working register
        movwf     PORTB    ; move the contents of the Working register into PORTB
        movlw     8          ; move decimal 8 into COUNT register
        movwf     COUNT    ; this controls the number of times program loops around
                    ; "again" (8 times)
again    call      wait1000ms ; wait 1000ms
        decfsz  COUNT,F   ; subtract 1 from COUNT and store result in COUNT
        goto     again     ; result not zero so keep checking
        goto     TEST      ; result zero, so continue to next instruction
```

Test program 6 and comment on its performance.

It is identical to program 5 except that a loop control is used to increase the display update time to eight seconds, so that it takes up to eight seconds to respond to a door opening or closing.

Suggest how it could be improved.

The STATUS register

This is one of the Special Function Registers, SFRs, discussed earlier. It comprises a number of selection bits, and ‘flags’. A flag is a marker to show that some event, or condition, has occurred. For example, the zero flag is used to spot when the processor, in subtracting one number from another, produces a result of zero.

The STATUS register has three main jobs:

- It allows us to select the memory bank used in the instructions that follow using bit five, the selection bit called **RP0**;
- It shows the results of arithmetic and logic operations, using the **zero flag (Z)**, the **carry flag (C)** and the **digit carry flag (DC)**.
- It identifies the cause of a device reset, on the **PD** and **TO** flags.

The structure is shown below:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
IRP	RP1	RP0	TO	PD	Z	DC	C

On power-up, bits **4** to **7** are all at logic 0 and so bank 0 is automatically selected.
In this course we will only make use of bits **2** and **5** (shown unshaded in the table).

RP0 - Memory Bank Select bits. (Each bank contains 128 bytes of memory.)

RP0 = 0: Bank0 (00h – 7Fh) selected.

RP0 = 1: Bank1 (80h – FFh) selected.

Z - Zero flag

Z = 1 means that the result of an arithmetic or logic operation was zero.

Z = 0 means that the result was not zero.

Look at the last column of table 1, to see the eight instructions that can affect the zero flag.

We use the zero flag in the status register to check if the contents of a file register have been reduced to zero.

Arithmetic Instructions

incf - (**increment file**) - the number stored in the named file register is increased by one.

```
movlw d'100'          ; The W register contains the decimal value '100'
movwf count            ; The count register contains the decimal value '100'
incf count,F          ; increment count by 1 so the count register contains decimal
                      ; value '101'
```

If the result is required in the working register **F** is replaced with **W**:

```
incf count,F          ; increment count by 1 so the W register contains decimal
                      ; value '101'
```

Adding 1 to a count of 255 causes ‘overflow’ and results in a stored value of zero. When this happens, the zero flag in the status register is set.

```
    movlw  d'255'          ; The W register contains the decimal value '255'  
    movwf  count            ; The count register contains the decimal value '255'  
    incf   count,F          ; increment count by 1 resulting in decimal value '0' and the  
                           ; zero flag is set
```

addlw - (**add** the **literal** to the content of **working register**) adds a literal eight-bit value to the contents of the working register and stores the result in the working register.

This is useful when you need to increase a count by more than one. For example, a cricket scoreboard could have three switches, to increase the recorded score by one, four and six runs respectively.

For example:

```
    movlw  d'103'          ; The W register contains the decimal value '103'  
    addlw  d'6'              ; The decimal value '6' is added to the W register.  
                           ; Now the W register contains the decimal value '109'
```

When count increases from decimal 255 to decimal 0 the zero flag in the status register is set.

```
    movlw  d'100'          ; The W register contains the decimal value '100'  
    addlw  d'156'          ; The decimal value '156' is added to the W register.  
                           ; Now the W register contains the decimal value '0'  
                           ; and the zero flag is set
```

sublw - **subtracts W from the literal** - **does NOT subtract the literal from W**.

The zero bit in the STATUS register is set by the subtract operation as follows:

Examples:

```
    movlw  4  
    sublw  5          ; 5 - 4 = 1 so Z=0  
    movlw  5  
    sublw  5          ; 5 - 5 = 0 so Z=1
```

We can test if the zero flag is set as follows:

```
test    sublw  5  
       btfsc STATUS,Z      ; check the zero flag in the STATUS register, is it set?  
       goto   test           ; no, so keep testing  
       nop                 ; yes, so skip to here
```

Investigation 3.7

Write program 7 to control a rugby union scoreboard.

The program:

- adds 3 points to the score when a penalty or a drop goal has been scored;
- adds 5 points to the score when a try but no conversion has been scored;
- adds 7 points to the score when a try plus a conversion has been scored;
- can reset the score (to zero) at the end of the game.

Program 7:

The assembler code is:

```

score    EQU B10          ; use reg B10 to store the score
        movlw b'00000000'      ; clear working register
reset    clrf  score       ; clear score register
        clrf  PORTB          ; make sure port B outputs are low at the start of
                                ; program
pen     btfsc PORTA,3      ; test bit 3 of port A to test for reset
        goto  reset           ; game over
        btfss PORTA,0          ; test bit 0 of port A to test for a penalty/drop goal
        goto  try              ; no penalty/drop goal so test for a try
        movf  score,W          ; penalty/drop goal so get ready to add to score
        addlw 3                 ; add 3 points for a penalty or a drop goal
        movwf score             ; add 3 points to score
        movwf PORTB            ; display score
        call  wait1000ms        ; wait 1 second to debounce switch 0
        goto  pen               ; loop back to check for next score
try    btfsc PORTA,3        ; test bit 3 of port A to test for reset
        goto  reset             ; game over
        btfss PORTA,1          ; test bit 1 of port A to test for an unconverted try
        goto  contry            ; no try so test for a try plus conversion
        movf  score,W          ; unconverted try so get ready to add to score
        addlw 5                 ; add 5 points for a try
        movwf score             ; add 5 points to score
        movwf PORTB            ; display score
        call  wait1000ms        ; wait 1 second to debounce switch 1
        goto  pen               ; loop back to check for next score
contry btfsc PORTA,3        ; test bit 3 of port A to test for reset
        goto  reset             ; game over
        btfss PORTA,2          ; test bit of port A to test for a try plus conversion
        goto  pen               ; no conversion so test for a penalty/drop goal
        movf  score,W          ; try plus conversion so get ready to add to score
        addlw 7                 ; add 7 points for a try plus conversion
        movwf score             ; add 2 points to score
        movwf PORTB            ; display score
        call  wait1000ms        ; wait 1 second to debounce switch 2
        goto  pen               ; loop back to check for next score

```

Use the program specification and assembler code to complete the flowchart for program 7.



Test program 7 and comment on its performance.

.....
.....
.....
.....
.....
.....
.....
.....

Suggest how it could be improved.

.....
.....
.....
.....
.....

Investigation 3.8

Write program 8 to control part of the security system for a building by testing the value of the data inputted as an entry code.

The program:

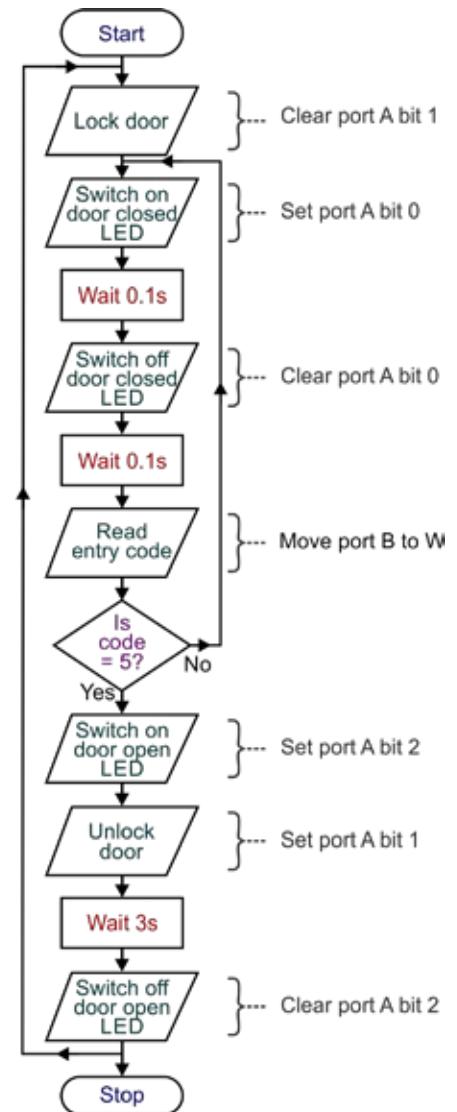
- reads an eight-digit entry code, entered on a keypad;
- releases the lock and lights a LED for three seconds when the correct code 000101_2 (decimal 5), is entered;
- flashes a second LED to indicate when the door is locked.

Connections:

- Keypad - connected to port **B**;
 - ‘Door locked’ flashing LED - connected to port **A** bit **0**;
 - Solenoid - connected to port **A** bit **1** (and keeps the door locked while bit **1** is low);
 - ‘Door open’ LED - connected to port **A** bit **2**;
- (Due to the sensors connected to port **A** of the PICAXE Trainer, PICAXE users may be able only to simulate this program)

Program 8:

The flowchart is shown opposite:



The assembler code is:

```
        clrf    PORTB
        clrf    PORTA      ; turn off all LEDs and lock door
        movlw   b'00000001' ; set up initial output conditions
        movwf   PORTA      ; 'door locked' LED on, 'door open' LED off, door locked
        call    wait100ms   ; 'door locked' LED on for 0.1s
        bcf    PORTA,0      ; 'door locked' LED off .....
        call    wait100ms   ; for 0.1s
        bcf    STATUS,Z     ; reset the zero flag in the STATUS register
        movf    PORTB,W     ; move the contents of Port B into the working register
        sublw   5            ; subtract the contents of the working reg from decimal 5
        btfss  STATUS,Z     ; check the zero flag in the STATUS register
        goto   test          ; if not set, keep testing
        bsf    PORTA,2      ; switch on 'door open' LED
        bsf    PORTA,1      ; open lock
        call    wait1000ms   ; wait 3 seconds
        call    wait1000ms
        call    wait1000ms
        goto   test          ; start again
```

Test program 8 and comment on its performance.

Investigation 3.9

Microcontrollers are often used to count events, like counting objects breaking a light beam on a conveyor belt.

There are complications in this seemingly simple task. In particular, care must be taken to ensure that the system counts each event only once. This may mean adding a short delay before the program loops back. Once a specific number of objects have passed through the beam and are placed in the box, another action, such as closing the box with the objects inside it, is carried out. The process then repeats. Write program 9 to count and pack objects on a conveyor belt.

Some of the actions taking place in this program, like closing the box and replacing it, are indicated by LEDs.

The program:

- reduces the count by one when the beam is broken;
- displays the number of objects needed to fill the box;
- closes the box when twelve objects have been placed in it;
- lights LED (1) to indicate that the box is full and about to be closed;
- waits three seconds;
- switches off LED (1);
- lights LED (2) to indicate that a replacement box is needed;
- then resets the counter (to twelve) to repeat the process.

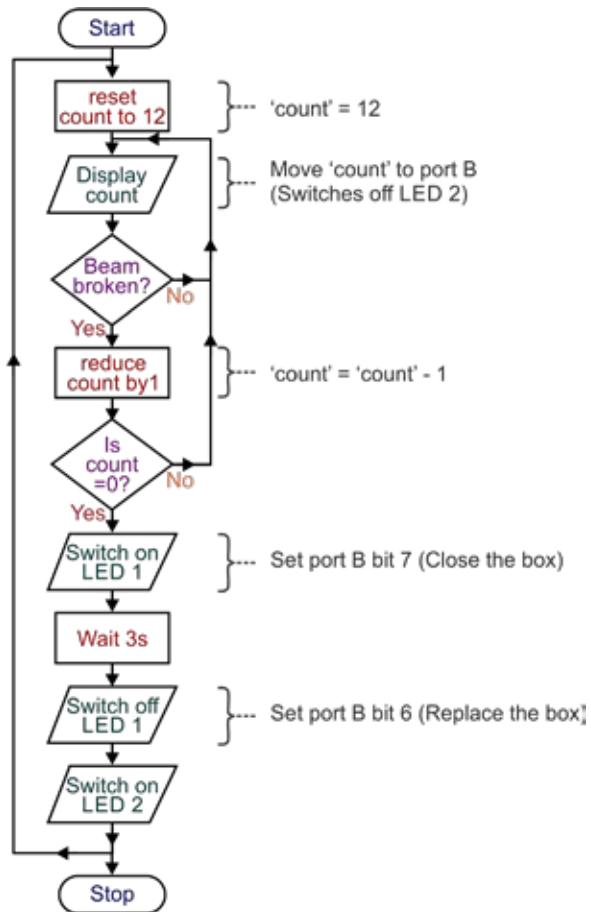
(When the counter contents are then displayed on Port B, LED (2) is turned off in the process.)

Connections:

- The number of objects needed to fill the box is displayed on port **B** bits **0** to **3**.
- The ‘Close box’ LED (1) is attached to port **B** bit **6**.
- The ‘Replace box’ LED (2) is attached to port **B** bit **7**.

Program 9:

The flowchart:



The assembler code is:

```

newbox    clrf    PORTA      ; clear port A
          clrf    PORTB      ; clear port B
          movlw   12          ; move the count into the working register
          movwf   PORTB      ; display count in port B
test     btfss   PORTA,0    ; has object been detected
          goto    test        ; no, so keep testing
          call    wait1000ms  ; yes, so wait 2s for object to pass(debounce)
          call    wait1000ms
          decfsz  PORTB,F    ; subtract one from count and check if count is zero
          goto    test        ; no, so keep testing
          bsf    PORTB,7      ; yes so close box
          call    wait1000ms  ; wait 3s for box to be closed
          call    wait1000ms
          call    wait1000ms
          goto    newbox       ; start again with new box
  
```

Test program 9 and comment on its performance.

Logic Instructions

The *andlw* instruction:

andlw - **AND**'s the content of the **literal** given in the instruction with the **working register**.

One use for the *andlw* operation is to **clear** certain bits in a register. For example, suppose that the working register contains the number 11001010, and you want to clear the lowest four bits, the instruction *andlw b'11110000'* would do this.

Working register contains	11001010
AND with	<u>11110000</u>
Result	11000000

The lowest four bits must be cleared by this operation because anything **AND** 0 = 0.

Masking:

Place '0's in the literal to clear the corresponding bits in the working register. The resulting literal is sometimes called a *mask*.

In the same way, the *andlw* operation can isolate the input from a sensor (or switch) attached to one bit of an input port from the other bits. Earlier, program 5 highlighted the need for isolation.

Suppose that a sensor is attached to bit 3 of port **A**. The instructions *movf PORTA,W* followed by *andlw b'00001000'* leaves all other bits cleared, and the state of bit 3 unaffected.

To see this, suppose we represent the contents of port **A** as **xxxxSxx** where 'x' is a *don't care* value (0 or 1) and **S** is the output of the sensor.

This is moved into the **W** register where it is **AND**'d with the mask *b'11110000'*:

Working register	xxxxSxxx
AND with	<u>00001000</u>
Result	0000S000

This process also tests the state of the sensor bit, because if that bit is zero, then the zero flag is set, otherwise it is not.

The *iorlw* instruction:

iorlw - **OR**'s the content of the **working register** with the **literal** given in the instruction

One use for the *iorlw* operation is to **set** bits in a register. (The *bsf* instruction can set only one bit at a time.) The instruction *iorlw b'11110000'* will set the highest four bits of the working register, leaving the four lowest bits unaffected.

For example:

Working register	01000010
OR with	<u>11110000</u>
Result	11110010

The uppermost four bits must be set because anything **OR** 1 = 1. Any '1's in the literal number set the corresponding bits in the working register.

Writing programs using logic operators

Take care:

To display the effect of logic operators such as *andlw* and *iorlw*, the number placed in the working register would have to be displayed before and after using the logic operator.

A delay is needed before and after the logic operation is executed in order to give time to see the number displayed. However, in doing this, the display subroutine places a different, unconnected number in the working register, giving an unexpected result.

The solution is to protect the contents of the working register in a different file before the delay subroutine is executed and then later return it to the working register.

Investigation 3.10

Write program 10 to demonstrate the effect of *iorlw* and the need to protect the working register.

The program:

- clears port **A** and port **B**;
 - moves the number 01100000_2 into the working register;
 - displays it on port **B**;
 - protects the contents of the working register in the file called ‘temp’;
 - adds a delay to give time for the display to be read;
 - displays the contents of the working register again to display the value corrupted by the delay subroutine;
 - recovers the original contents of the working register;
 - displays it on port **B** to show that it has worked;
 - applies the *iorlw* operator to that number;
 - displays the result on port **B**;
 - adds a delay to give time for the display to be read;
 - repeats the process.

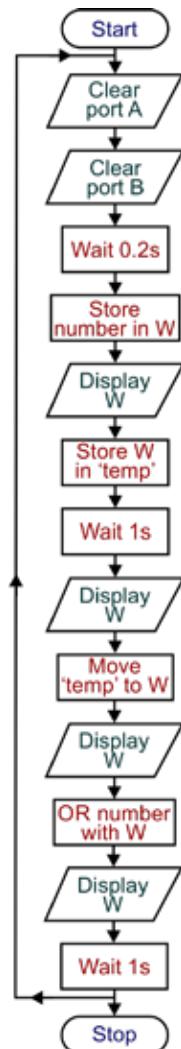
Program 10

The assembler code is:

```

temp      EQU     B10          ; rename B10 as temp
          clrf    PORTA        ; clear port A
again     clrf    PORTB        ; clear port B
          call    wait100ms    ; delay100 milliseconds
          call    wait100ms    ; delay100 milliseconds
          movlw   b'01100000'  ; load number into W
          movwf   PORTB        ; display it on port B
          movwf   temp         ; protect contents of W in temp
          call    wait1000ms   ; delay 1000 milliseconds
          movwf   PORTB        ; send pattern to port B
          movf    temp,W       ; recover pattern to W reg
          movwf   PORTB        ; send pattern to port B
          iorlw  b'00000110'  ; W register OR'ed with b'01100000
          movwf   PORTB        ; result in port B
          call    wait1000ms   ; delay1000 milliseconds
          goto   again         ; loop forever

```



Test program 10 and comment on its performance.

Investigation 3.11

Write program 11 to create a menu using *andlw*.

The program:

- clears port **B** (turning off the LEDs);
- reads the menu choice into the working register;
- when switch **X** is pressed, sequence 1 is selected.
This lights the first LED for two seconds and then the second LED lights for one second.
This sequence is complete and so the program returns to the beginning;
- when switch **Y** is pressed, sequence 2 is selected.
This pulses the buzzer four times - one second on and then one second off each time.
Then the program returns to the beginning.

Pressing different switches changes the number stored, as the table shows.

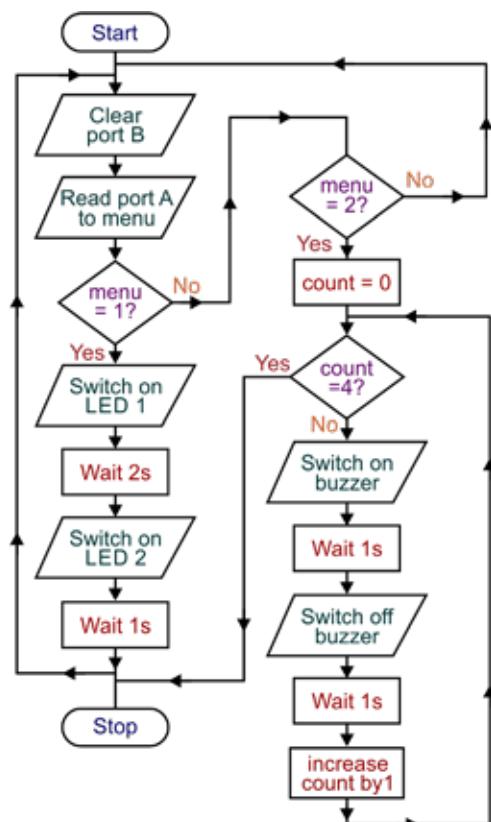
Switch pressed	Number stored
X	1
Y	2

Connections:

- Switch X, connected to port A bit 0, selects task 1.
- Switch Y, connected to port A bit 1, selects task 2.
- LEDs are connected to port B, bits 1 and 2.
- Buzzer is connected to port B, bit 3.

Program 11

The flowchart:



The assembler code is:

count	EQU B10	; rename B10 as count
begin	clrf PORTB movf PORTA,W andlw 1 btfs STATUS,Z goto task1 movf PORTA,W andlw 2 btfs STATUS,Z goto task2 goto begin	; turn off outputs at the start of program ; move the task choice into working register ; isolate bit 1 and then test its state ; check the zero flag in the STATUS register ; not set so task 1 is selected. ; set so move the task choice into working register again ; isolate bit 2 and then test its state ; check the zero flag in the STATUS register ; not set so task 2 selected ; set so no task selected - loop back to beginning
task1	bsf PORTB,0 call wait1000ms call wait1000ms bsf PORTB,1 call wait1000ms goto begin	; turn first LED on ; wait1s ; wait1s ; turn second LED on ; wait1s ; task1 complete so go back to begin
task2	clrf count test	; set count register to zero ; move the current value of count into the working register
	sublw 4 btfs STATUS,Z goto begin bsf PORTB,2 call wait1000ms bcf PORTB,2 call wait1000ms incf count,F goto test	; subtract the contents of the working reg from decimal 4 ; does this set the zero flag in the STATUS register? ; yes - task2 is complete so go back to beginning ; no - turn buzzer on ; wait1s ; turn buzzer off ; wait1s ; add 1 to count ; go back and check how many flashes have occurred

Test program 11 and comment on its performance.

Interrupts

An *interrupt* is an event that forces the processor to jump from its current activity to a specific point in the program to carry out a special program called the *Interrupt Service Routine* (ISR). Interrupts are designed to be special events whose arrival cannot be predicted precisely.

A non-PIC example:

You are holding a party. You are not sure when the guests will arrive. One option is to open the front door every couple of minutes to see if anyone is there. This is very time-consuming, and means that you cannot really get on with other jobs. In programming, this approach is called *polling*. Alternatively, you can wait for the doorbell to ring. Then you can stop whatever you are doing to open the front door. This is like using an interrupt.

Interrupts allow the microcontroller to respond immediately to an external event, such as a switch being pressed. These events are called hardware interrupts.

For example, an environmental control system for a building may have a number of functions.

It could:

- control air conditioning, heating and ventilation (HVAC);
- detect fires in the building.

To control HVAC, the program polls a number of sensors throughout the building, periodically, switching heating equipment and fans on and off accordingly. However, if a fire breaks out, an instant response is needed. If the control program polled the smoke sensors, it may take time for the program to check that particular sensor. Connecting the smoke sensors to cause an interrupt ensures speedy response when there is a fire.

Hardware Interrupt Circuit

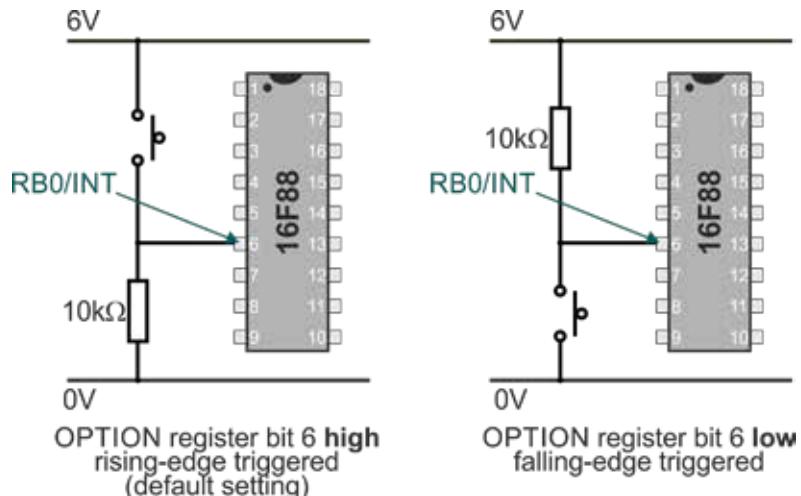
The interrupt can be set to occur on either the rising or the falling edge of the signal pulse from the RB0/INT pin, depending on the state of bit **6** of the OPTION register.

OPTION register bit **6**:

- logic 1 - interrupt occurs on the **rising** edge of the RB0/INT pulse;
- logic 0 - interrupt occurs on the **falling** edge of the RB0/INT pulse.

On power-up, the OPTION register contains ‘1111 1111’, and so, by default, interrupts occur on the rising edge of the RB0/INT pulse.

The circuit diagrams show how an interrupt is triggered in each case.



Note: If the right hand circuit was used with Option bit 6 high the interrupt would occur when the switch was released rather than pressed.

Interrupt Service Routine

When an interrupt is detected, the processor completes the current instruction and then jumps to the Interrupt Service Routine.

This takes place in stages. The processor:

- completes the instruction it is currently executing;
- stores the contents of the *program counter* on the *stack*, an area of memory used to store return addresses;
- loads the address 004h, the Interrupt Vector Address;
- executes the instruction found there.

You can write the ISR starting at memory location 4.

More commonly, you write a *goto* instruction in memory location 4, redirecting the processor to the location of the ISR. Usually you mark the start of the ISR with a label such as *inter*.

Once the processor has completed the ISR, it uses the stack to return to the point in the main program where it left off when the interrupt was called.

The INTCON register

There are four types of events that can trigger interrupts:

- completion of writing data to the EEPROM memory;
- overflow of the TMR0 timer (changing from FFh to 00h);
- change of state of any of the pins RB4, RB5, RB6 or RB7;
- change of state of pin RB0/INT to logic 0.

This course requires consideration of only the last one, a hardware interrupt triggered by use of the RB0/INT pin on the PIC chip.

Interrupts are controlled by a Special Function Register, called INTCON, which allows users to make the CPU respond to any type of interrupt event and indicate which type has taken place.

Its structure is shown below.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
GIE	PEIE	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF

Key: In this course we will only make use of bits **1, 4 and 7** (shown unshaded in the table)

G/E Global Interrupt Enable bit

- logic 1 enables all interrupt sources;
- logic 0 disables them.

INT0IE - RB0/INT External Interrupt Enable bit

- logic 1 enables;
- logic 0 disables it.

INT0IF - RB0/INT External Interrupt Flag

- logic 1 – RB0/INT interrupt occurred;
- logic 0 – no interrupt occurred.

- The Global Interrupt Enable bit must be set to make the processor respond to any type of interrupt.

You can clear this bit if you don't want the processor to be interrupted in a particularly important part of the program.

- On power-up, the INTCON register contains the binary number 0000 000x₂, where x is unknown. This means that all interrupts are disabled by default.
- The ISR must clear bit 1, the INT0F bit, which flags up that a RB0/INT interrupt happened so that the program returns to normal operation. Otherwise the ISR will run repeatedly.
- The GIE bit is cleared automatically when an interrupt occurs - “an interrupt can't be interrupted”. On returning to the main program, the GIE bit must be set once more to make the processor respond to further interrupts. The *retfie* command, (return from interrupt and enable {GIE}), does this automatically but the *return* command does not.

The following code enables RB0/INT interrupts:

```
movlw b'10010000'      ; enables GIE and INTE bits only  
movwf INTCON           ; and clears all interrupt flags.
```

As the template includes the following equates:

```
INTCON EQU h'0B'        ; interrupt control  
INT0IF EQU h'01'        ; interrupt 0 enable  
GIE    EQU h'07'        ; global interrupt enable
```

It uses this equivalent code to enable interrupts:

```
; bsf INTCON,INT0IE ; set external interrupt enable  
; bsf INTCON,GIE    ; enable all interrupts
```

The following code clears the INT0IF flag:

```
bcf   INTCON,1          ; clears bit 1 (INT0IF) of the INTCON register.
```

Using the equate statements described above, the template equivalent is:

```
bcf   INTCON,INT0IF    ; clear interrupt flag
```

Protecting registers during interrupts

When an interrupt occurs, the program counter contents are automatically stored (on the stack). Once the ISR is completed, the processor retrieves these from the stack allowing it to continue with the main program from where it left off.

No other registers are protected in this way. This causes problems when the contents of a register are changed during execution of the ISR. On return to the main program, the changed contents of the register may cause undesirable effects.

The working register is particularly vulnerable in this respect. It is good practice to save its contents at the beginning of the ISR, and restore them at the end. To do so, in the Register Usage section of the program, nominate a file register to use to store the contents of the working register. This is done by using another equate statement.

The code in this case is:

```
W_temp equ 10h            ; the temporary storage place is now called W_temp, and is file  
                           ; register number 10h
```

At the beginning of the ISR, labelled *inter* in the following code, store the contents of the working register in the *W_temp* file register:

```
inter movwf W_temp
```

At the end of the ISR, before returning to the main program, restore the original contents of the working register:

```
    movf  W_temp,W      ; move the file W_temp into the working register
    retfie             ; return to the main program
```

Remember

Before writing a program with interrupts you will need to “uncomment” the two lines starting with bsf in the template at the point shown below.

```
;*****MAIN PROGRAM *****
;*****remove semicolons from next two lines to use interrupt routine;*****
; bsf      INTCON,INT0IE   ; set external interrupt enable
; bsf      INTCON,GIE     ; enable all interrupts
```

Putting all this together produces the complete interrupt service routine section of the template:

```
;*****
;           INTERRUPT SERVICE ROUTINE
;*****
W_SAVE EQU B20          ; backup registers used in interrupts

interrupt
    movwf W_SAVE          ; copy W to save register
    btfss INTCON,INT0IF   ; check correct interrupt has occurred
    retfie                ; no, so return and re-enable GIE

;*****The interrupt service routine (if required) goes here*****

    bcf  INTCON,INT0IF   ; bit clear so clear interrupt flag
    movf W_SAVE,W         ; restore W
    retfie                ; return and re-set GIE bit

    END                  ; all programs must end with this
```

Note: The above interrupt service routine section is the only one you need for this course. It has a limitation in that it does not protect any status bits being used in the main program and could cause problem on returning from the interrupt to the main. This issue can be overcome by using the more sophisticated interrupt service routine shown below (for reference only.)

```
;*****
;           INTERRUPT SERVICE ROUTINE
;*****
W_SAVE EQU B20          ; backup registers used in interrupts
S_SAVE EQU B21

interrupt
    movwf W_SAVE          ; copy W to save register
    swapf STATUS, W        ; swap status to be saved into W
    clrf  STATUS           ; clear status to reset to page 0
    movwf S_SAVE           ; save status save register
```

```
btfs INTCON,INT0IF ; check correct interrupt has occurred  
retfie ; no, so return and re-enable GIE  
  
*****The interrupt service routine (if required) goes here*****  
  
bcf INTCON,INT0IF ; bit clear so clear interrupt flag  
swapf S_SAVE, W ; swap status save register into W  
movwf STATUS ; move W into STATUS register  
swapf W_SAVE, F ; swapf w save into F, then again into W  
swapf W_SAVE, W ; (swapf is used because it does not alter any  
; STATUS bits)  
retfie ; return and re-set GIE bit  
  
END ; all programs must end with this
```

Investigation 3.12

Write program 12 to demonstrate the use of an interrupt service routine.

The program:

- continuously complements the output displayed on port **B**;
- jumps to the interrupt service routine when interrupted;
- within the ISR, flashes a LED connected to port **B** bit **2** while the interrupt pin is held high.

The assembler code is:

```

init
    clrf PORTB          ; make sure port B output bits are low
    bsf STATUS,RP0       ; select register page 1
    movlw b'11111111'   ; set port A data direction (NB – For PICAXE 18M2,
                         ; set bit 7 to 1)
    movwf TRISA
    movlw b'00000001'   ; set port B data direction (NB – For 16F88, set
                         ; bit 0 to 1)
    movwf TRISB
    bcf STATUS,RP0       ; select register page 0

;***** MAIN PROGRAM *****
;***** remove semicolons from next two lines to enable interrupt routine*****
    bsf      INTCON,INT0IE ; set external interrupt enable
    bsf      INTCON,GIE   ; enable all interrupts

main
seq1    clrf PORTB
        movlw b'10101010' ; set up initial output conditions
        movwf PORTB         ; send pattern to port B
        call   wait1000ms   ; delay 1000 milliseconds
        comf  PORTB,F       ; invert output pattern
        call   wait1000ms   ; delay1000 milliseconds
        goto  seq1           ; loop seq1 forever

;***** INTERRUPT SERVICE ROUTINE *****
;***** W_SAVE EQU B20 ; backup the working register used in interrupts

interrupt:
        movwf W_SAVE          ; copy W to save register
        btfss INTCON,INT0IF  ; check correct interrupt has occurred
        retfie                ; no, so return and re-enable GIE

;*****The interrupt service routine (if required) goes here*****


seq2    clrf PORTB
        bsf   PORTB,2        ; output B.2 high
        call  wait100ms      ; delay 100 milliseconds
        bcf   PORTB,2        ; output B.2 low

```

```
call    wait100ms      ; delay 100 milliseconds
btfsC  PORTB,0        ; ***** NB- change to PORTA,7, for PICAXE 18M2*****
goto   seq2            ; no so loop again
bcf    INTCON,INT0IF  ; clear interrupt flag
movf   W_SAVE,W       ; restore W
retfie                         ; return and re-set GIE bit

END      ; all programs must end with this
```

Test program 12.

In doing so, interrupt the main program ‘seq 1’ several times when bits **1**, **3**, **5** and **7** are high. Describe what you observe when the program returns to ‘seq 1’.

Investigation 3.13

Write program 13 to allow an ISR to latch on.

The program:

- requires a modification of the ISR of program 12 so that the interrupt sequence is latched on even if the interrupt pin is released.

The ISR is unlatched by taking port **A** bit **0** high. Only one change to the ISR is needed.

;*****The interrupt service routine (if required) goes here*****

```
seq2
    clrf    PORTB
    bsf     PORTB,2      ; output B.2 high
    call    wait100ms    ; delay 100 milliseconds
    bcf     PORTB,2      ; output B.2 low
    call    wait100ms    ; delay 100 milliseconds
    ;btfsc PORTB,0       ; replace this line with a different instruction in space
    ; below

    goto   seq2          ; bit not clear so loop again
    bcf    INTCON,INT0IF ; bit clear so clear interrupt flag
    movf   W_SAVE,W      ; restore W
    retfie
    END               ; all programs must end with this
```

Test program 13 and comment on its performance.

Investigation 3.14

Write program 14 to create a light meter.

The program:

- displays the light level as a 8-bit binary number on port **B**;
- the light level display updates every 3 seconds;
- is controlled by a light sensor, connected to port **A** bit **0**;
- causes a warning LED, connected to port **A** bit **3**, to light for 6 seconds when the light level is greater than 01111112 (decimal 127).

Features

- The program makes use of the *readadc0* subroutine, contained in the template. This provides an example of using port **A** bit **0** as an analogue port. (This technique could be of use in programs for the NEA (non-exam assessment)).
- It also uses the idea of loop control introduced in Program 6 to create a 3 second delay subroutine.

```

count  equ B10          ; rename B10 as count
call    readadc0        ; readadc A.0 into register B0
movf   B0,W            ; get ADC on pin A.0 result
movwf  PORTB           ; display as binary on 8 LEDs
call    wait3sec        ; delay 3 seconds
btfs  PORTB,7          ; is light level more than 127
goto   main             ; no so update light level display
bsf   PORTA,3           ; yes so display warning LED
call    wait3sec        ; delay 6 seconds
call    wait3sec
bcf   PORTA,3           ; result zero, so continue to next instruction
goto   main             ; next reading

; *****3 second delay subroutine*****
wait3sec movlw 3          ; move decimal 3 into COUNT register
           movwf COUNT         ; controls the number of times S/R loops around "again"
                           ; (3 times)
again   call    wait1000ms   ; wait 1000ms
           decfsz COUNT,F      ; subtract 1 from COUNT and store result in COUNT
           goto   again          ; result not zero so keep checking
           return

```

Test program 14 and comment on its performance.

.....
.....
.....
.....
.....
.....
.....
.....

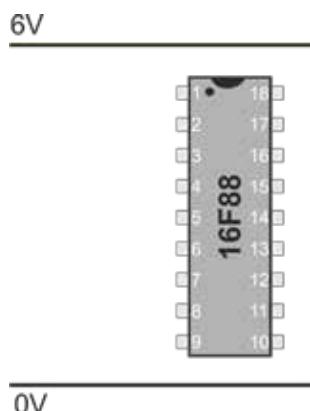
Exercise 3.2

1. (a) Write a statement to associate the name *workstore* with the register at address 0Fh.

(b) Write an Interrupt Service routine that:

- is identified by the label *panic*;
 - protects the working register by transferring its contents to the file register *workstore* at the beginning of the ISR;
 - clears the INT0IF bit;
 - outputs logic 1 to four sirens connected to bits **4**, **5**, **6** and **7** of port **B**;
 - calls the subroutine called *thirtysec*;
 - recovers the contents of the working register;
 - returns to the main program and resets the global interrupt enable bit as it does so.

2. The PIC 16F88 microcontroller in the diagram is configured so that the RB0/INT pin is active low.
- (a) Complete the circuit diagram to show how a push switch and other necessary components, are connected so that as the switch is pressed, the microcontroller jumps to the interrupt service routine.



- (b) Complete the instructions to configure the INTCON register so that:
- pressing the switch triggers an interrupt;
 - all other interrupt sources are disabled.

```
movlw .....  
.....
```

- (c) The interrupt service routine is given below:

```
inter      movlw      b'11110000'  
          movwf      PORTB  
          call       tensec      ;call the ten second delay subroutine.  
          retfie
```

The interrupt vector address is 04.

Write the instruction that must be included at that address to jump to the interrupt service routine.

```
04 .....
```

3. Complete the instructions to configure:

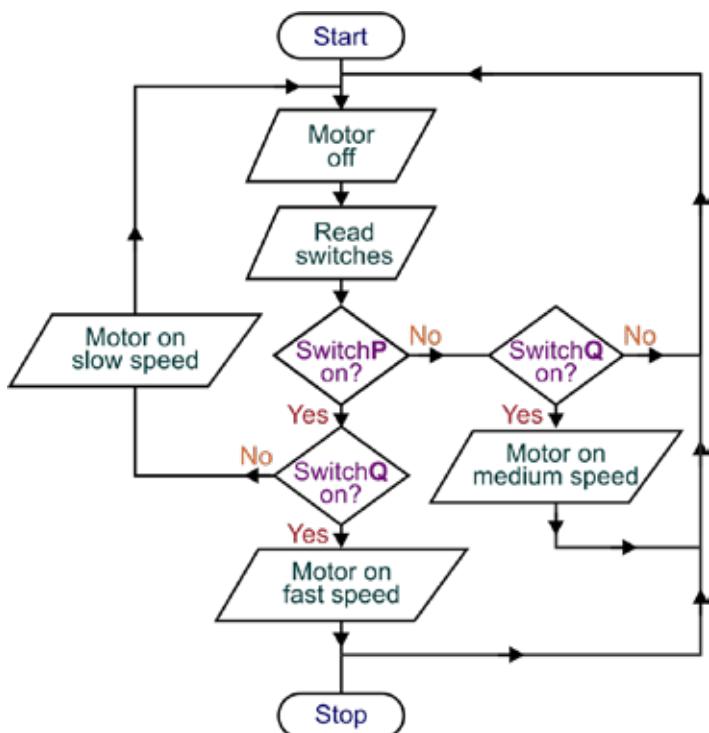
- all bits of port **A** and the four lowest significant pins of port **B**, as inputs;
- the four highest significant pins of port **B** as outputs.

bsf	STATUS, RP0
movlw	b'.....'
movwf
movlw	b'.....'
movwf
bcf	STATUS, RP0

4. An electric motor is used in the extractor fan in a cooker hood to remove fumes from a cooker in the kitchen. The speed of the motor is controlled by a microprocessor. Two switches, **P** and **Q**, on the cooker hood select the speed. The flowchart specifies the way in which the switches control the motor speed.

Connections:

- switch **P** - connected to port A, bit 0;
- switch **Q** - connected to port A, bit 1;
- the motor - connected to port B bit 7.



When either switch is pressed, it outputs a logic 1 signal.

- (a) Analyse the flowchart to establish how the setting of the switches determines the motor speed and complete the following:

Switch **P** and **Q** both off

.....

Switch **P** on (alone)

.....

Switch **Q** on (alone)

.....

Switch **P** and **Q** both on

.....

- (b) The following code section checks the state of switches **P** and **Q**, and uses it to set the speed of the motor, using the subroutines ‘*slow_speed*’, ‘*medium_speed*’ and ‘*fast_speed*’, defined elsewhere, to control the speed of the motor.

```

110 begin      clrf      port B
111           clrf      port A
112           bcf       STATUS,Z
113           movf      port A,W
114           sublw     1
115           btfsc    STATUS,Z
116           goto     slow
117           movf      port A,W ;the next section checks whether switch Q,
                           ;alone, is pressed
118           sublw     .....
119           btfsc    STATUS,Z
120           goto     medium
121           movf      port A,W ;the next section checks whether both
                           ;switches are pressed
122           sublw     3
123           btfsc    STATUS,Z
124           goto     .....
125           goto     begin

126 slow       call      slow_speed
127           goto     begin
128 medium    call      medium_speed
129           goto     begin
130 fast      call      fast_speed
131           goto     begin

```

- (i) What is the purpose of the instruction in line 113?
-
-

- (ii) Explain what is happening in lines 113 to 116.
-
-
-

- (iii) Complete line 118. This part of the program checks the state of switch Q and act upon it.
- (iv) Complete lines 124. This makes the program jump to a new location when the zero flag is set.

Appendix 1

```

;***** INCLUDE FILE PROVIDED BY WJEC/EDUQAS *****
;***** errorlevel -302      ; hide page warnings *****

;Common registers
INDF      EQU  H'00'      ; register used when calling indirect address
PCL       EQU  H'02'      ; program counter
STATUS    EQU  H'03'      ; name program location 3 as STATUS
FSR       EQU  H'04'      ; register which holds Indirect address
PORTA    EQU  H'05'
PORTB    EQU  H'06'      ; name program location 6 as PORTB
INTCON   EQU  H'0B'      ;
OPTION_REG EQU  H'81'
TRISA    EQU  H'85'
TRISB    EQU  H'86'

;User registers
B0       EQU  H'20'      ; general use byte registers B0 to B27
B1       EQU  H'21'
B2       EQU  H'22'
B3       EQU  H'23'
B4       EQU  H'24'
B5       EQU  H'25'
B6       EQU  H'26'
B7       EQU  H'27'
B8       EQU  H'28'
B9       EQU  H'29'
B10      EQU  H'2A'
B11      EQU  H'2B'
B12      EQU  H'2C'
B13      EQU  H'2D'
B14      EQU  H'2E'
B15      EQU  H'2F'
B16      EQU  H'30'
B17      EQU  H'31'
B18      EQU  H'32'
B19      EQU  H'33'
B20      EQU  H'34'
B21      EQU  H'35'
WAIT1    EQU  H'40'      ; counters used in wait delays
WAIT10   EQU  H'41'
WAIT100  EQU  H'42'
WAIT1000 EQU  H'43'
WAIT0    EQU  H'44'
ADCTEMP  EQU  H'45'      ; adc loop counter

;Common register bits
W        EQU  0          ; working register flag
F        EQU  1          ; file register flag
C        EQU  0          ; carry flag

```

```

Z          EQU    2           ; zero
DC         EQU    1           ; digit carry
WR         EQU    1
WREN       EQU    2
RS         EQU    0
RD         EQU    0
RP0        EQU    5           ; register page bit
INT0IF     EQU    1           ; interrupt 0 flag
INT0IE     EQU    4           ; interrupt 0 enable
GIE        EQU    7           ; global interrupt enable

; The following line embeds configuration data into the PICmicro
;      LIST P=16F88

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
#ifndef PIC16F88

#define SupportedChip
LIST P=16F88

#ifndef UseCrystal
#ifndef UseWatchdog
__CONFIG H'2007', H'3F66' ; HS mode with WDT enabled
__CONFIG H'2008', H'3FFC'
#else
__CONFIG H'2007', H'3F62' ; HS mode
__CONFIG H'2008', H'3FFC'
#endif
#else
#ifndef UseInternal
#ifndef UseWatchdog
__CONFIG H'2007', H'3F74' ; INTRC mode with WDT enabled
__CONFIG H'2008', H'3FFC'
#else
__CONFIG H'2007', H'3F70' ; INTRC mode
__CONFIG H'2008', H'3FFC'
#endif
#else
#ifndef UseWatchdog
__CONFIG H'2007', H'3F76' ; EXTRC mode with WDT enabled
__CONFIG H'2008', H'3FFC'
#else
__CONFIG H'2007', H'3F72' ; EXTRC mode
__CONFIG H'2008', H'3FFC'
#endif
#endif
#endif
#endif
#endif

#define PAGE0   BCF    STATUS,5
#define PAGE1   BSF    STATUS,5
#define PAGE_LO BCF    STATUS,6
#define PAGE_HI BSF    STATUS,6

```

```

;Chip-specific registers
ADRESH EQU H'1E'
ADCON0 EQU H'1F'
OSCCON EQU H'8F'
ADRESL EQU H'9E'
ADCON1 EQU H'9F'
ANSEL EQU H'9B'
EEADR EQU H'10D'
EEADDRH EQU H'10F'
EEDATA EQU H'10C'
EEDATH EQU H'10E'
EECON1 EQU H'18C' ;eeprom write register 1
EECON2 EQU H'18D' ;eeprom write register 2

        ORG 0 ;reset vector
        GOTO 5 ;go to start of program
        ORG 4 ;interrupt vector

#ifndef UseInterrupt
        GOTO interrupt ;state interrupt jump address
#else
        GOTO 5 ;go to start of program
#endif
        ORG 5 ;start of program memory

        CLRF PORTA ;clear Port A data pins
        CLRF PORTB ;clear Port B data pins

        PAGE1
        CLRF ANSEL ;clear the ANSEL register
        CLRF TRISA ;set all Port A as output
        CLRF TRISB ;set all Port B as output

#ifndef UseInternal
        MOVLW H'60'
        MOVWF OSCCON
#endif

PAGE0

        GOTO start001 ;jump to start of user's program

#endif
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
#ifndef PIC16F84A

#define SupportedChip
LIST P=16F84A

#ifndef UseCrystal
#ifndef UseWatchdog
__CONFIG H'3FF6' ;HS mode with WDT enabled

```

```
#else
    __CONFIG H'3FF2'          ; HS mode
#endif
#else
#ifndef UseInternal
    error "16F84a has no internal RC clock"
#else
#ifndef UseWatchdog
    __CONFIG H'3FF7'          ; RC mode with WDT enabled
#else
    __CONFIG H'3FF3'          ; RC mode
#endif
#endif
#endif

#define PAGE0      BCF      STATUS,5
#define PAGE1      BSF      STATUS,5
#define PAGE_LO    NOP
#define PAGE_HI    NOP

;Chip-specific registers
EEADR      EQU      H'09'
EEDATA      EQU      H'08'
EECON1      EQU      H'88'      ;eeprom write register 1
EECON2      EQU      H'89'      ;eeprom write register 2

        ORG      0          ; reset vector
        GOTO    5          ; go to start of program
        ORG      4          ; Interrupt vector

#ifndef UseInterrupt
        GOTO    interrupt ; state interrupt jump address
#else
        GOTO    5          ; go to start of program
#endif
        ORG      5          ; start of program memory

        CLRF    PORTA      ; clear port A data pins
        CLRF    PORTB      ; clear port B data pins
        PAGE1
        CLRF    TRISA      ; set all port A as output
        CLRF    TRISB      ; set all port B as output
        PAGE0

        GOTO    start001   ; jump to start of user's program

#endif
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
#ifndef PIC16F877A

#define SupportedChip
LIST P=16F877a
```

```
#ifdef UseCrystal
    #ifdef UseWatchdog
        __CONFIG H'3F06'          ; HS mode with WDT enabled
    #else
        __CONFIG H'3F02          ; HS mode
    #endif
#else
    #ifdef UseInternal
        error "16F877a has no internal RC clock"
    #else
        #ifdef UseWatchdog
            __CONFIG H'3F07'      ; RC mode with WDT enabled
        #else
            __CONFIG H'3F03'      ; RC mode
        #endif
    #endif
#endif

#define PAGE0      BCF      STATUS,5
#define PAGE1      BSF      STATUS,5
#define PAGE_LO    BCF      STATUS,6
#define PAGE_HI    BSF      STATUS,6

;Chip-specific registers
ADRESH           EQU      H'1E'
ADCON0           EQU      H'1F'
ADRESL           EQU      H'9E'
ADCON1           EQU      H'9F'
EEADR            EQU      H'10D'
EEADRH           EQU      H'10F'
EEDATA           EQU      H'10C'
EEDATH           EQU      H'10E'
EECON1           EQU      H'18C' ;eeprom write register 1
EECON2           EQU      H'18D' ;eeprom write register 2

        ORG      0          ; reset vector
        GOTO    5          ; go to start of program
        ORG      4          ; interrupt vector

#ifdef UseInterrupt
        GOTO    interrupt   ; state interrupt jump address
#else
        GOTO    5          ; go to start of program
#endif
        ORG      5          ; start of program memory

        CLRF    PORTA      ; clear port A data pins
        CLRF    PORTB      ; clear port B data pins

PAGE1
MOVLW    H'06'      ; set all pins as digital inputs
MOVWF    ADCON1      ;
```

```
CLRF      TRISA      ; set all port A as output via Page 1
CLRF      TRISB      ; set all port B as output via Page 1

PAGE0

GOTO      start001    ; jump to start of user's program

#endif
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
#ifndef SupportedChip
    error "No chip specified or the specified chip is not supported"
#endif
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```

```
;*****  
;  
        SUBROUTINES  
;*****  
  
; Predefined wait subroutines - wait1ms, wait10ms, wait100ms, wait1000ms  
  
#ifdef UseCrystal  
; timings for crystal (19.6608MHz)  
  
wait250us  
    ; (244 x 5) + 9 = 1229 instructions = 0.25ms @ 19.6608MHz  
    movlw d'244'          ;1  
    movwf WAIT0           ;1  
  
loop__250  
    clrwdt                ;1 this loop 1+1+1+2 = 5 instructions  
    nop                   ;1  
    decfsz WAIT0,F       ;1  
    goto loop__250         ;2  
  
    nop                   ;1  
    nop                   ;1  
    nop                   ;1  
    return                ;2  
  
wait1ms  
    movlw d'4'              ;1  
    movwf WAIT1             ;1  
  
loop__1  
    call wait250us         ;  
    decfsz WAIT1,F        ;  
    goto loop__1            ;  
    return                 ;  
  
wait10ms  
    movlw d'10'              ;  
    movwf WAIT10            ;  
  
loop__2  
    call wait1ms            ;  
    decfsz WAIT10,F        ;  
    goto loop__2            ;  
    return                 ;  
  
wait100ms  
    movlw d'100'             ;  
    movwf WAIT100           ;  
  
loop__3  
    call wait1ms            ;  
    decfsz WAIT100,F        ;  
    goto loop__3            ;  
    return                 ;  
  
wait1000ms  
    movlw d'10'              ;  
    movwf WAIT1000           ;
```

```
loop__4
    call wait100ms           ;
    decfsz WAIT1000,F        ;
    goto loop__4              ;
    return                    ;

#ifndef UseInternal
; timings for internal oscillator (4MHz)
wait1ms
    ; (198 x 5) + 10 (including call and return) = 1000
    movlw d'198'              ;1
    movwf WAIT1                ;1

loop5ns
    clrwdt                  ;1 this loop 1+1+1+2 = 5 instructions
    nop                      ;1
    decfsz WAIT1,F            ;1
    goto loop5ns               ;2

    nop                      ;1
    nop                      ;1
    nop                      ;1
    nop                      ;1
    return                   ;2

wait10ms
    movlw d'10'                ;10 x 1ms = 10ms
    movwf WAIT10               ;
loop10ms
    call wait1ms               ;
    decfsz WAIT10,F            ;
    goto loop10ms               ;
    return                     ;

wait100ms
    movlw d'100'               ;100 x 1ms = 100ms
    movwf WAIT100               ;
loop100ms
    call wait1ms               ;
    decfsz WAIT100,F           ;
    goto loop100ms              ;
    return                     ;

wait1000ms
    movlw d'10'                 ;10 x 100ms = 1000ms
    movwf WAIT1000               ;
loop1000ms
    call wait100ms              ;
    decfsz WAIT1000,F          ;
    goto loop1000ms             ;
    return                     ;
```

```

    #else
; timings for external oscillator (variable, approx 1Hz)
wait1ms
    return ; return straight away (these delays are not applicable with external
           ; RC oscillator at 1Hz)

wait10ms
    return ; return straight away (these delays are not applicable with external
           ; RC oscillator at 1Hz)

wait100ms
    return ; return straight away (these delays are not applicable with external
           ; RC oscillator at 1Hz)

wait1000ms
    return ; return straight away (these delays are not applicable with external
           ; RC oscillator at 1Hz)

#endif
#endif

; Predefined ADC subroutines – readadc0, readadc1, readadc2

readadc0
    movlw b'00000001' ; setup mask for pin A.0
    call  readadc      ; do the adc conversion
    movwf B0          ; save result in B0
    return

readadc1
    movlw b'00000010' ; setup mask for pin A.1
    call  readadc      ; do the adc conversion
    movwf B1          ; save result in B1
    return

readadc2
    movlw b'00000100' ; setup mask for pin A.2
    call  readadc      ; do the adc conversion
    movwf B2          ; save result in B2
    return

readadc
    ; generic sub routine to read ADC 0, 1 or 2 (pass appropriate mask in W)
    ; to start conversion we need mask (001, 010, 100) in ANSEL bits 0-2
    ; but the actual channel number (0, 1, 2) in ADCON0 channel select bits
    ; then set the ADCON0, GO bit to start the conversion

    bcf   STATUS,RP0    ; select register page 0
    movwf ADCTEMP       ; 0000??? get mask value
    bsf   STATUS,RP0    ; select register page 1
#endif PIC16F88
    movwf ANSEL         ; move mask value 001,010,100 into ANSEL
#endif

```

```

#define PIC16F877a
    movlw      b'00000010'          ; set AN0 to AN4 to analogue input
    movwf      ADCON1             ;
#endif
    bcf       STATUS,RP0           ; select register page 0
    rlf       ADCTEMP,F          ; 0000???x  rotate twice
    rlf       ADCTEMP,W          ; 000???xx
    andlw     b'00011000'          ; 000??000  mask off the unwanted bits
    iorlw     b'00000001'          ; 000?001  set the 'ADC on' bit
    movwf      ADCON0             ; move working into ADCON0
    movlw     d'10'                ; 10 x 3 = 30us acquisition time
    movwf      ADCTEMP            ; re-use ADC1 register as a counter
loopacq
    decfsz   ADCTEMP,F          ; loop around to create short delay
    goto     loopacq             ; each loop is 1+2 = 3 instructions = 3us @
                                ; 4MHz
    bsf       ADCON0,2            ; now start the conversion
loopadc
    clrwdt   ADCON0,2            ; pat the watchdog
    btfsc    ADCON0,2            ; is conversion finished?
    goto     loopadc             ; no, so wait a bit more
    movf     ADRESH,W            ; move result into W
    bsf       STATUS,RP0           ; select register page 1
#endif PIC16F88
    clrf      ANSEL              ; reset ADC so all digital inputs
#endif
#endif PIC16F877a
    clrf      ADCON1             ; reset ADC so all digital inputs
    bsf       ADCON1,1             ;
    bsf       ADCON1,2             ;
#endif
    bcf       STATUS,RP0           ; select register page 0
    return
;*****
; Delay routines
;*****
; Delay = 0.1 seconds
; Clock frequency = 19.6608 MHz

; Actual delay = 0.1 seconds = 491520 cycles
; Error = -1.18423789293e-14 %

    cblock   0x4D
    d1
    d2
    d3
    endc

```

```
Delay ;491510 cycles
    movlw      0x47
    movwf      d1
    movlw      0x13
    movwf      d2
    movlw      0x02
    movwf      d3

Delay_1
    decfsz   d1, f
    goto     $+2
    decfsz   d2, f
    goto     $+2
    decfsz   d3, f
    goto     Delay_1

                                ;3 cycles
    goto     $+1
    goto     $+1
    goto     $+1          ;4 cycles (including call)
    return

;*****
; The following label signifies the start of the user's program
; i.e. the first line after the include statement.
;*****
```

start001