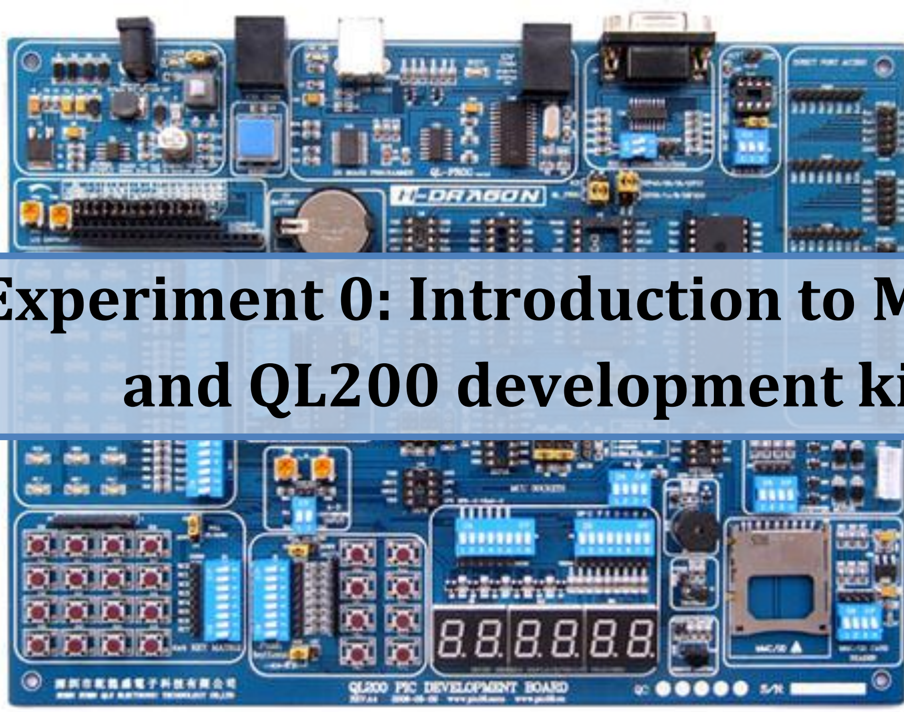
University of Jordan
Faculty of Engineering and Technology
Department of Computer Engineering
Embedded Systems Laboratory 0907334



Experiment 0: Introduction to MPLAB and QL200 development kit

Objectives

The main objectives of this experiment are to familiarize you with:

- Microchip MPLAB Integrated Development Environment (IDE) and the whole process of building a project, writing simple codes, and compiling the project.
- Code simulation
- QL200 development kit
- QL-PROG software and learn how to program the PIC using it

Starting MPLAB

After installation, shortcut of this software will appear on desktop.

<u>Create asm file using MPLAB</u>

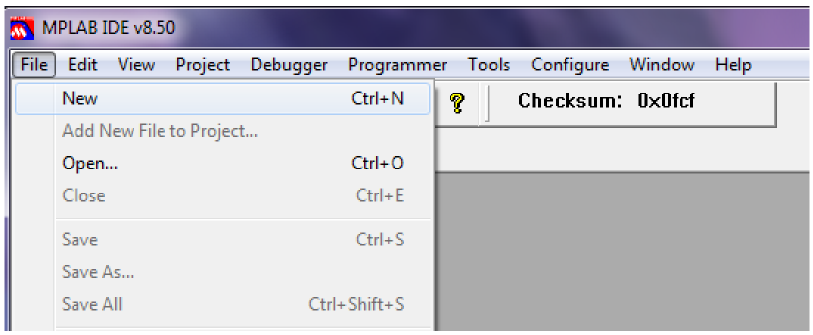
a) Double click on the "**MPLAB**" program icon found on the desktop.

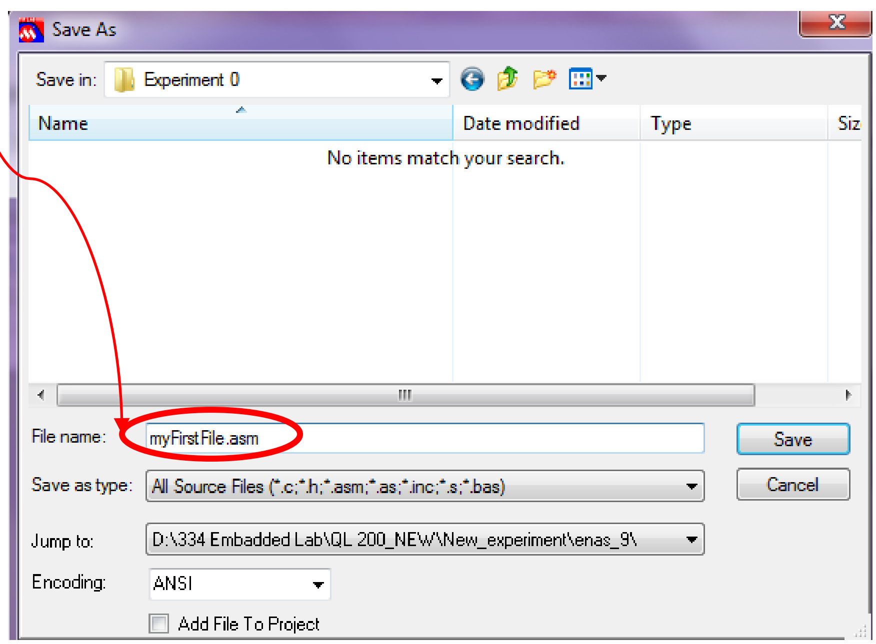
Note: All programs written, simulated and debugged in MPLAB should be stored in files with .asm extension.

b) *To create asm, follow these simple steps:*

i. File → New

ii. File → Save as, in the save dialog box; name the file as "myFirstFile.asm" **WITHOUT THE DOUBLE QUATATIONS MARKS**, this will instruct MPLAB to save the file in .asm format.



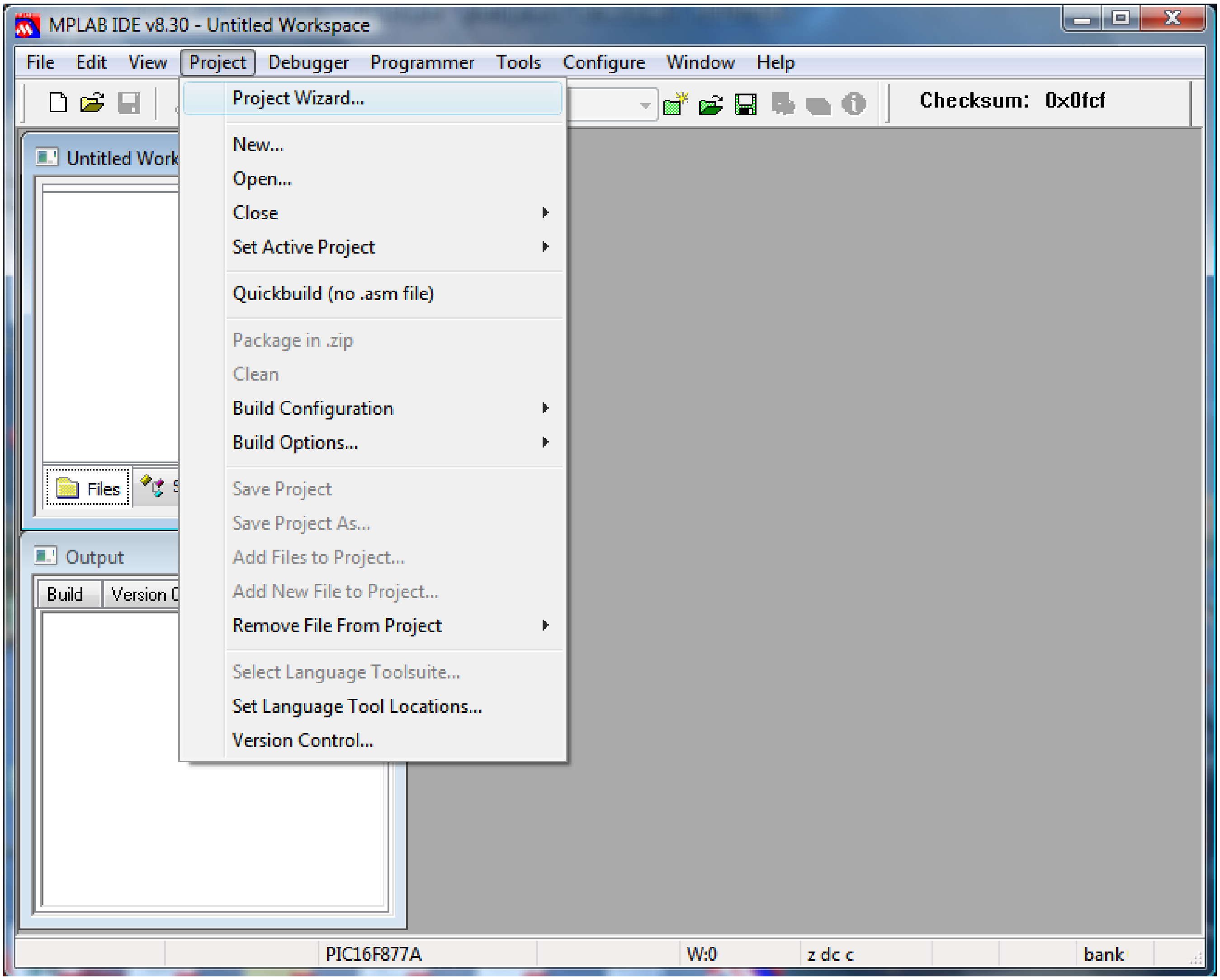


NOTE: All your files should be stored in a short path:

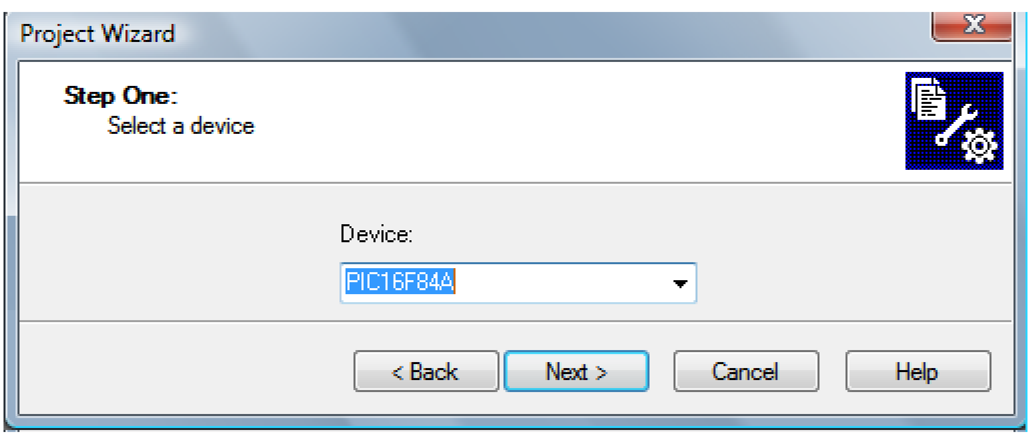
The total number of characters in a path should not exceed 64	Char No.	
C:\ or D:\ or ...	3	✓
D:\Embedded\	12	✓
D:\Embedded\Lab	15	✓
D:\Engineer\Year_Three\Summer_Semester\Embedded_Lab\Experiment_1\MyProgram.asm	78	×
Any file on Desktop		×

Create a project in MPLAB by following these simple steps:

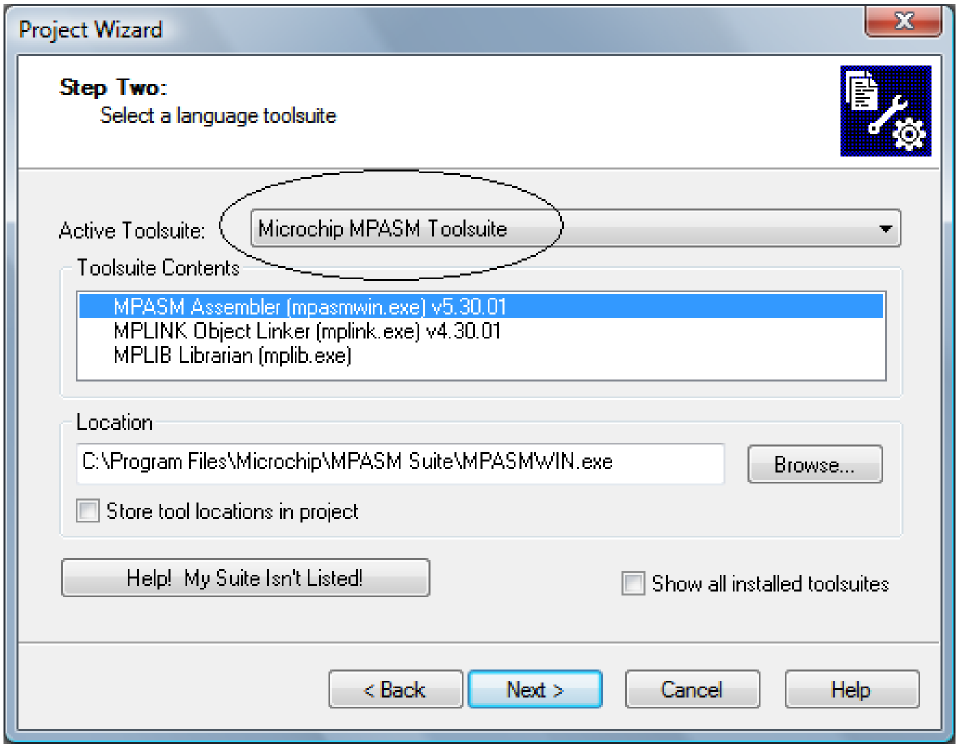
1. Select the Project → Project Wizard menu item → Next



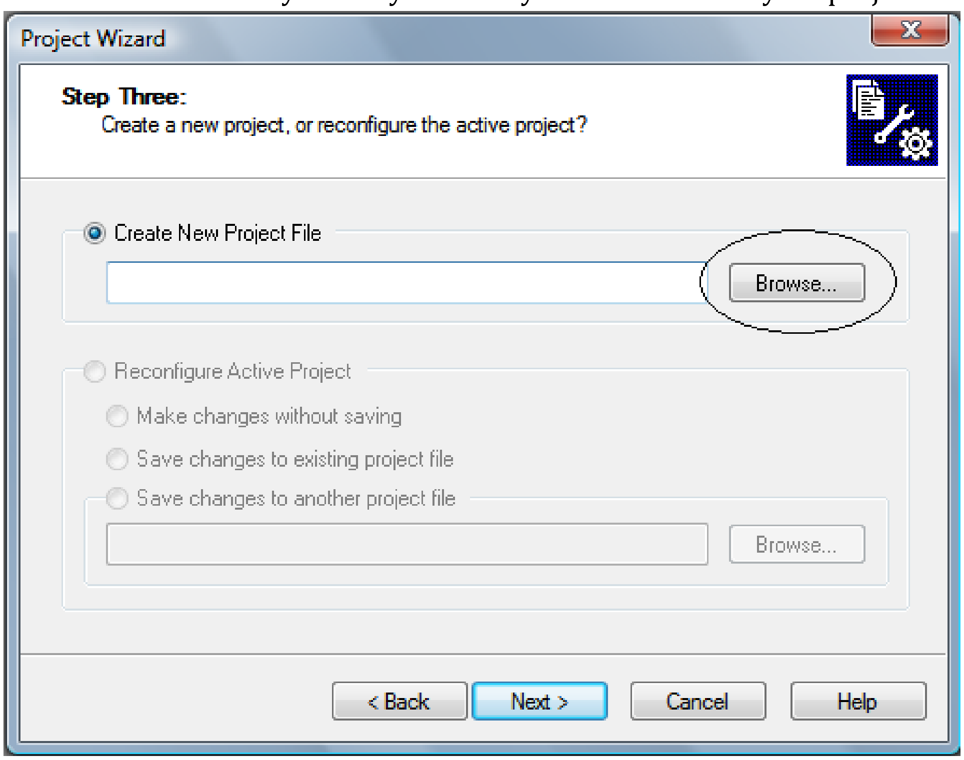
2. In the device selection menu, choose 16F84A (or your target PIC) → Next



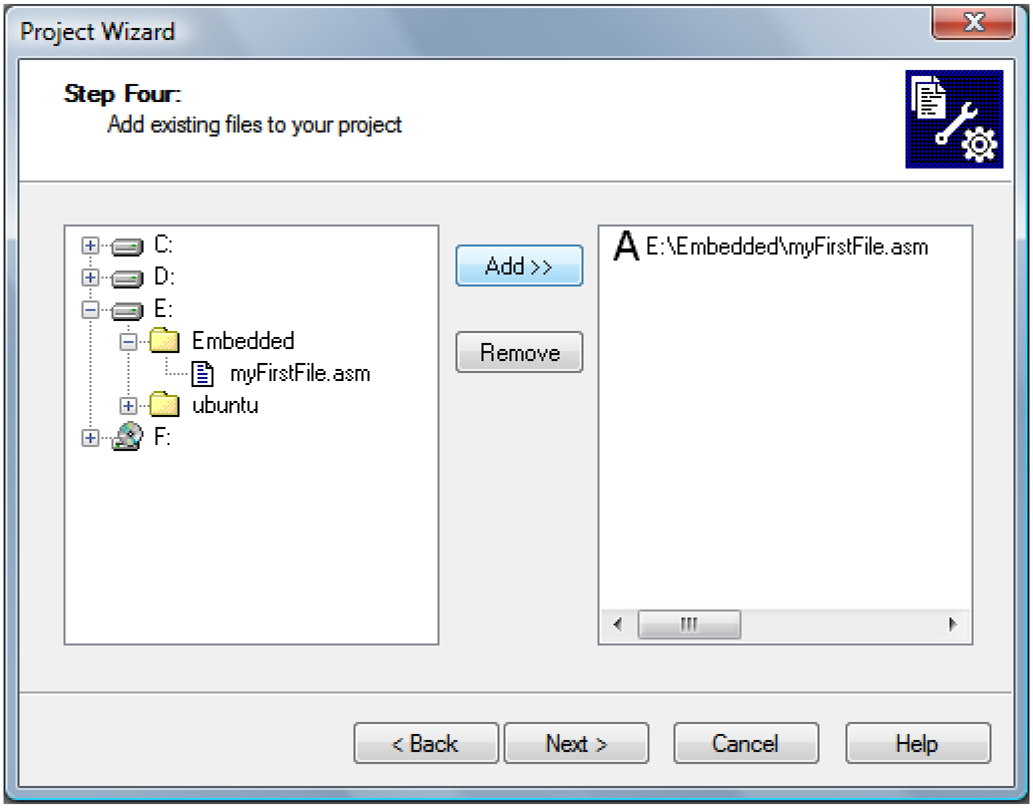
3. In the Active Toolsuite, choose Microchip MPASM Toolsuite → Click next.
DO NOT CHANGE ANYTHING IN THIS SCREEN



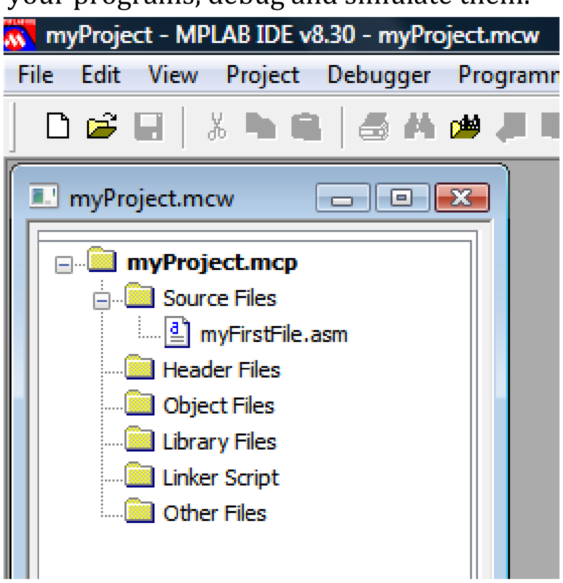
4. Browse to the directory where you saved your ASM file. Give your project a name → Save → Next.



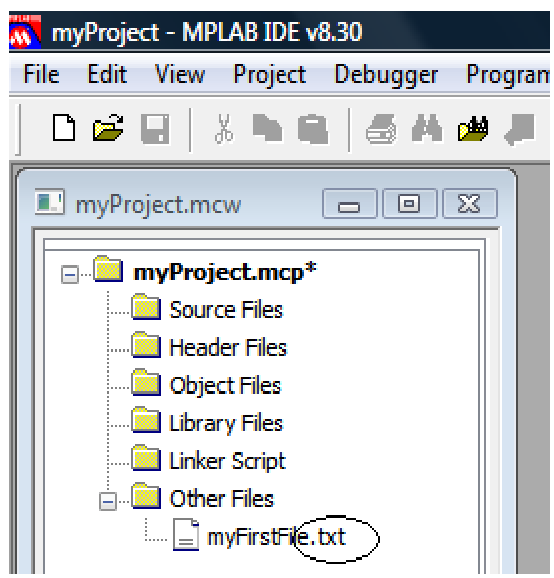
5. If, in Step 4, you navigated correctly to your file destination you should see it in the left pane otherwise choose back and browse to the correct path. When done Click add your file to the project (here: myFirstFile.asm). Make sure that the letter A is beside your file and not any other letter → Click next →Click Finish.



6. You should see your ASM file under *Source file, now you are ready to begin*
Double click on the myFirstFile.asm file in the project file tree to open. This is where you will write your programs, debug and simulate them.



CORRECT



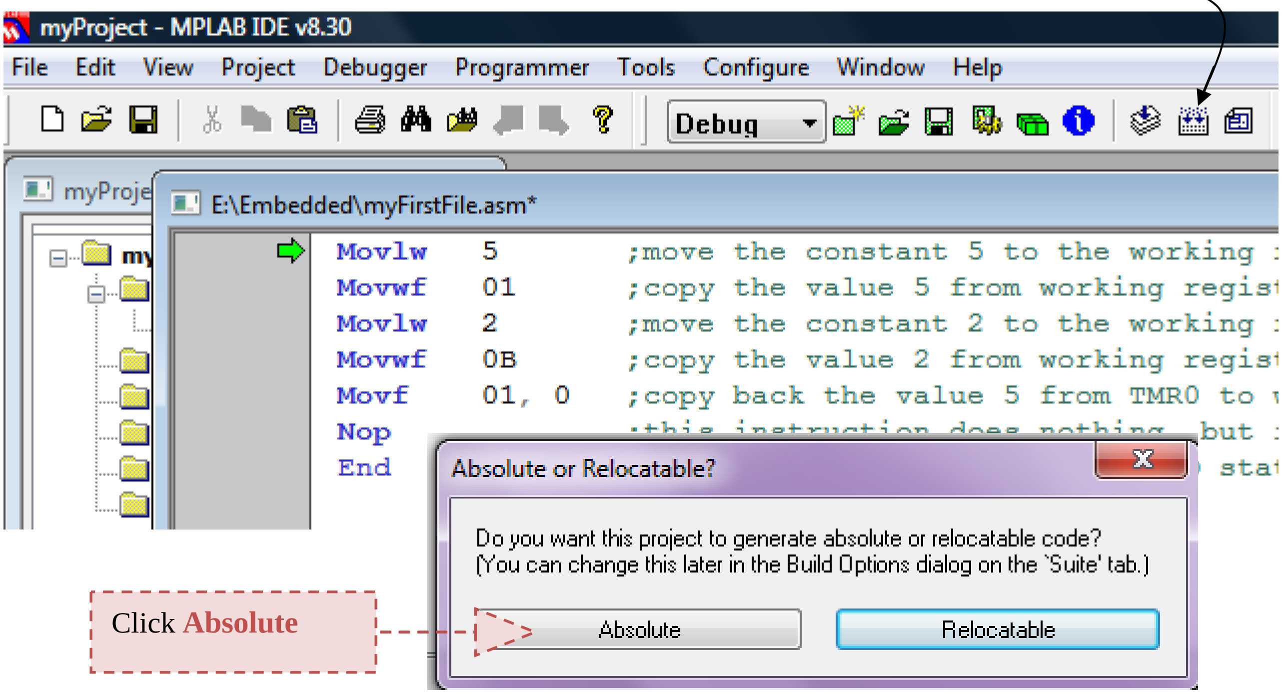
WRONG

Now we will simulate a program in MPLAB and check the results

In MPLAB write the following program:

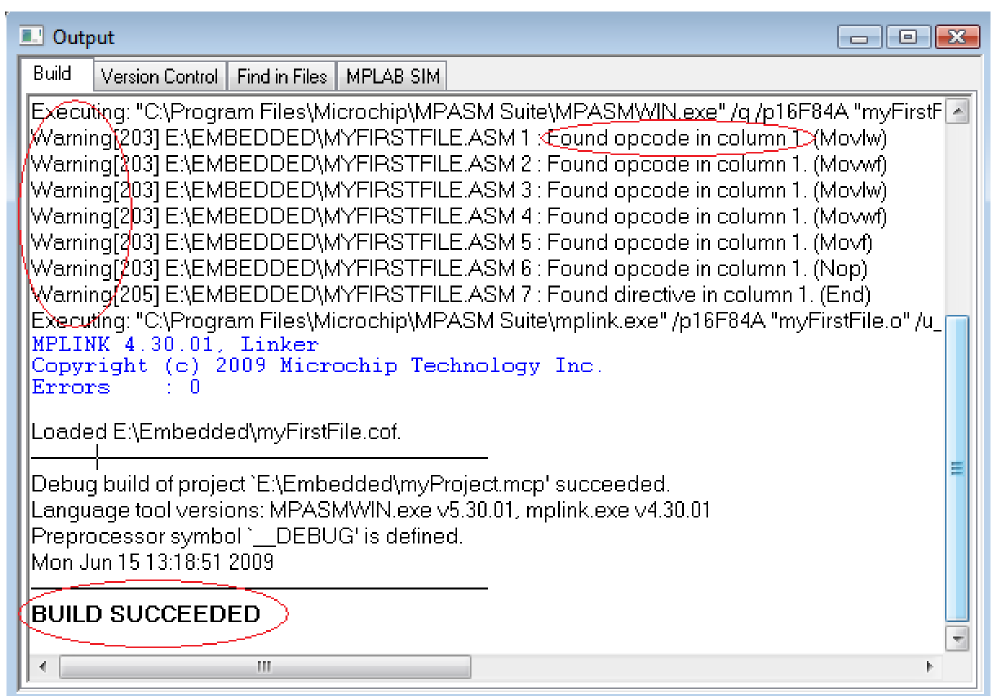
Movlw	5	; move the constant 5 to the working register
Movwf	01	; copy the value 5 from working register to TMR0 (address 01)
Movlw	2	; move the constant 2 to the working register
Movwf	0B	; copy the value 2 from working register to INTCON (address 0B)
Movf	01, 0	; copy back the value 5 from TMR0 to working register
Nop		; this instruction does nothing, but it is important to write for now
End		; every program must have an END statement

After writing the above instructions we should build the project, do so by pressing ***build***



An output window should show:

BUILD SUCCEDDED

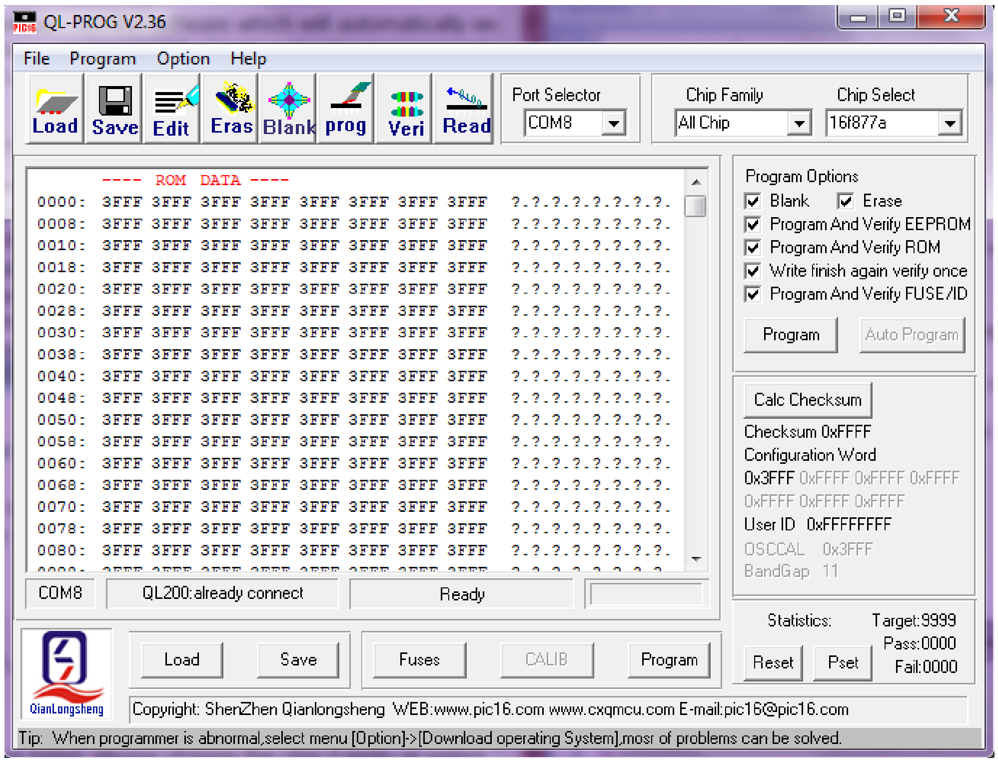


QL-PROG – How to Program

Prepared by Eng. Enas Jaara

After installation, shortcut of this software will appear on desktop.

1. **Connect hardware and power up the kit,** run the programming software **QL-PROG** (Double click it to run the software**)** which will automatically search programmer hardware. It will appear as shown in the below diagram



2. **Select Chip Family and Chip model**

   Choose All Chip from the chip family and choose 16F877A from the chip select

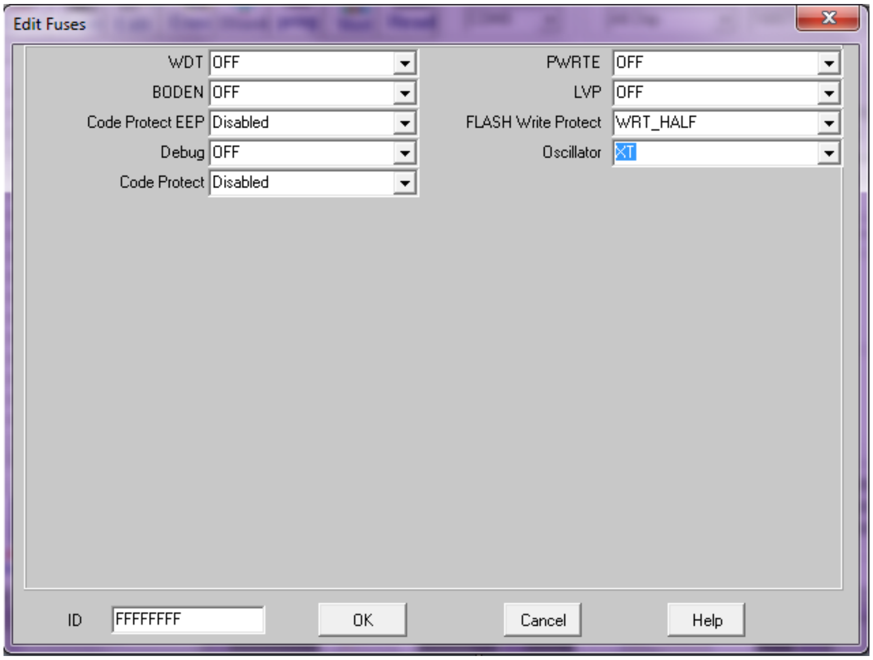
3. Press **Erase** button on programming software panel to Erase the chip data

4. **Load File to Program**

Press "**Load**" button on programming software panel to load machine code file (HEX file) of the chip you desire to program. load the LCD1.hex found on D:\Experiment0

5. Set Configuration Bit

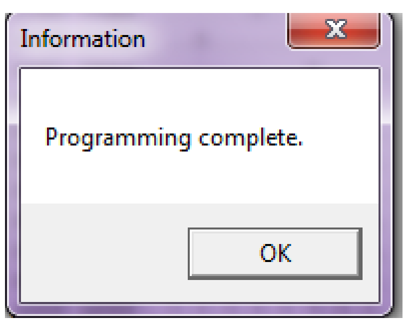
You may set or change the configuration bit of chip by running pressing "**Fuses**" button on software panel. After running the command software, pop-up window to set configuration bit will appear as shown in below diagram. Set the options according to your requirement and click "OK" button.



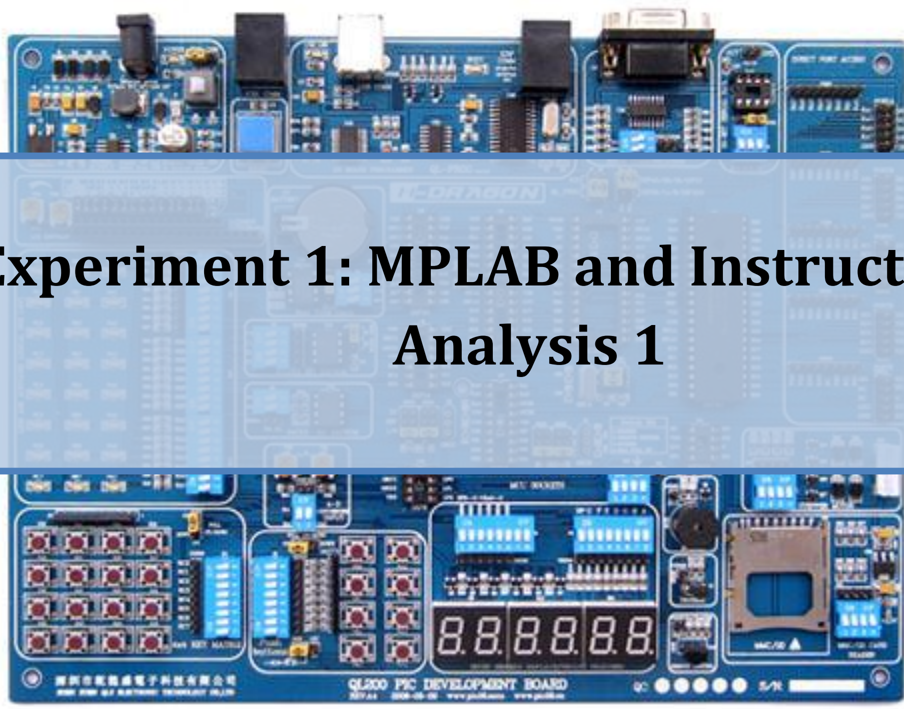
If any of the above option differs, it is because you have chosen the wrong PIC, so go to **chip select** and choose your appropriate PIC.

6. Program the PIC

Press "Program" button to begin programming. After completion, there will be messages of "**Programming complete**".



University of Jordan
Faculty of Engineering and Technology
Department of Computer Engineering
Embedded Systems Laboratory 0907334



1 Experiment 1: MPLAB and Instruction Set Analysis 1

Objectives

The main objectives of this experiment are to familiarize you with:

- The MOV instructions
- Writing simple codes, compiling the project and Code simulation
- The concept of bank switching
- The MPASM directives
- Microcontroller Flags
- Arithmetic and logical operations

Pre-lab requirements

Before starting this experiment, you should have already acquired the MPLAB software and the related PIC datasheets from drive D on any of the lab PC's. You are encouraged to install the latest version of MPLAB (provided in the lab) especially if you have Windows Vista

Starting up with instructions

Movement instructions

You should know by now that most PIC instructions (logical and arithmetic) work through the working register "W", that is one of their operands must always be the working register "W", the other operand might be either a constant or a memory location. Many operations store their result in the working register; therefore we can conclude that we need the following movement operations:

1. Moving constants to the working register (Loading)
2. Moving values from the data memory to the working register (Loading)
3. Moving values from the working register to the data memory (Storing)

INSTRUCTIONS ARE CASE INSENSITIVE: YOU CAN WRITE IN EITHER SMALL OR CAPITAL LETTERS

- MOV**LW**: moves a literal (constant) to the working register (final destination). The constant is specified by the instruction. You can directly load constants as decimal, binary, hexadecimal, octal and ASCII. The following examples illustrate:

DEFAULT INPUT IS HEXADECIMAL

1. MOVLW 05 : moves the constant 5 to the working register
2. MOVLW 10 : moves the constant **16** to the working register.
3. MOVLW 0xAB : moves the constant $\mathbf{AB_h}$ to the working register
4. MOVLW H'7F' : moves the constant $\mathbf{7F_h}$ to the working register
5. MOVLW CD : WRONG, if a hexadecimal number starts with a character, you should write it as 0CD or 0xCD or H'CD'
6. MOVLW d'10' : moves the **decimal** value 10 to the working register.
7. MOVLW .10 : moves the **decimal** value 10 to the working register.
8. MOVLW b '10011110' : moves the **binary** value 10011110 to the working register.
9. MOVLW O '76' : moves the **octal** value 76 to the working register.
10. MOVLW A'g' : moves the **ASCII** value **g** to the working register.

- MOV**WF: *COPIES*** the value found in the working register into the data memory, **but to which location?** The location is specified along with the instruction and according to the memory map.

So what is the memory map?

A memory map shows all available registers (in data memory) of a certain PIC along with their addresses, it is organized as a table format and has two parts:

1. **Upper part:** which lists all the Special Function Registers (SFR) in a PIC, these registers normally have specific functions and are used to control the PIC operation
2. **Lower part:** which shows the General Purpose Registers (GPR) in a PIC; GPRs are data memory locations that the user is free to use as he wishes.

Memory Maps of different PICs are different. Refer to the datasheets for the appropriate data map

REGISTER FILE MAP - PIC16F84A

File Address	Bank 0	Bank 1	File Address
00h	Indirect addr.(1)	Indirect addr.(1)	80h
01h	TMR0	OPTION_REG	81h
02h	PCL	PCL	82h
03h	STATUS	STATUS	83h
04h	FSR	FSR	84h
05h	PORTA	TRISA	85h
06h	PORTB	TRISB	86h
07h	—	—	87h
08h	EEDATA	EECON1	88h
09h	EEADR	EECON2(1)	89h
0Ah	PCLATH	PCLATH	8Ah
0Bh	INTCON	INTCON	8Bh
0Ch–4Fh	68 General Purpose Registers (SRAM)	Mapped (accesses) in Bank 0	8Ch–CFh
50h–7Fh			D0h–FFh

☐ Unimplemented data memory location, read as '0'.

Note 1: Not a physical register.

Examples:

1. MOVWF 01 : COPIES the value found in W to TMR0
2. MOVWF 05 : COPIES the value found in W to PORTA
3. MOVWF 0C : COPIES the value found in W to a GPR (location 0C)
4. MOVWF 32 : COPIES the value found in W to a GPR (location 32)
5. MOVWF 52 : WRONG, out of data memory range of the PIC 16F84a (GPR range is from 0C-4F and 8C to CF)

- **MOVF:** ***COPIES*** a value found in the data memory to the **working register** **OR to** **itself**.

Therefore we expect a second operand to specify whether the destination is the working register or the register itself.

For now: a 0 means the W, a 1 means the register itself.

Examples:

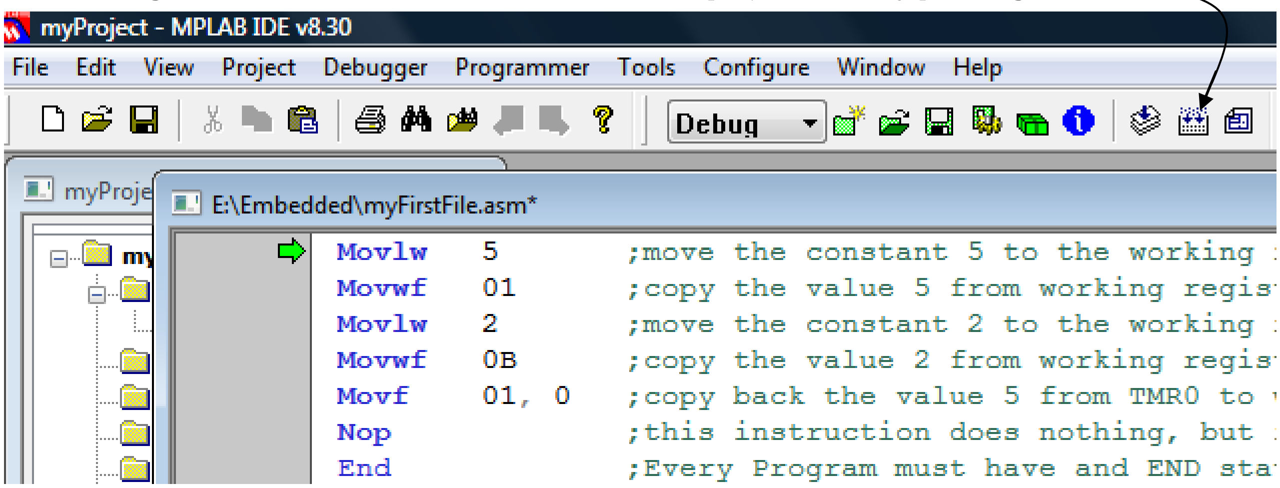
1. MOVF 05, 0 : copies the content of PORTA to the working register
2. MOVF 2D, 0 : copies the content of the GPR 2D the working register
3. MOVF 05, 1 : copies the content of PORTA to itself
4. MOVF 2D, 1 : copies the content of the GPR 2D to itself

Now we will simulate a program in MPLAB and check the results

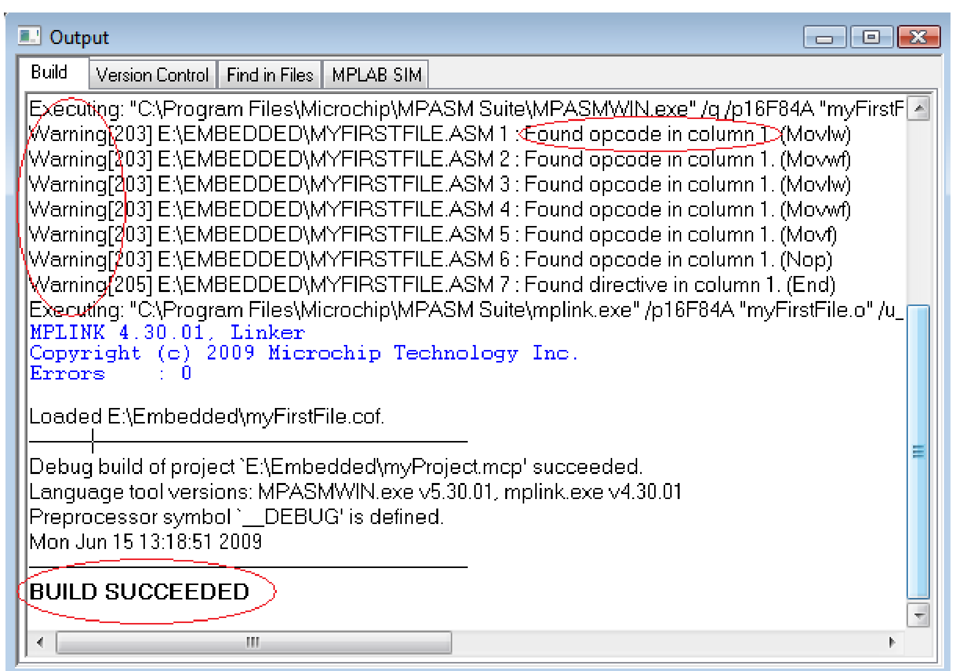
In MPLAB write the following program:

```
Movlw      5       ; move the constant 5 to the working register
Movwf      01      ; copy the value 5 from working register to TMR0 (address 01)
Movlw      2       ; move the constant 2 to the working register
Movwf      0B      ; copy the value 2 from working register to INTCON (address 0B)
Movf       01, 0   ; copy back the value 5 from TMR0 to working register
Nop                ; this instruction does nothing, but it is important to write for now
End                ; every program must have an END statement
```

After writing the above instructions we should build the project, do so by pressing ***build***



An output window should show: BUILD SUCCEDDED



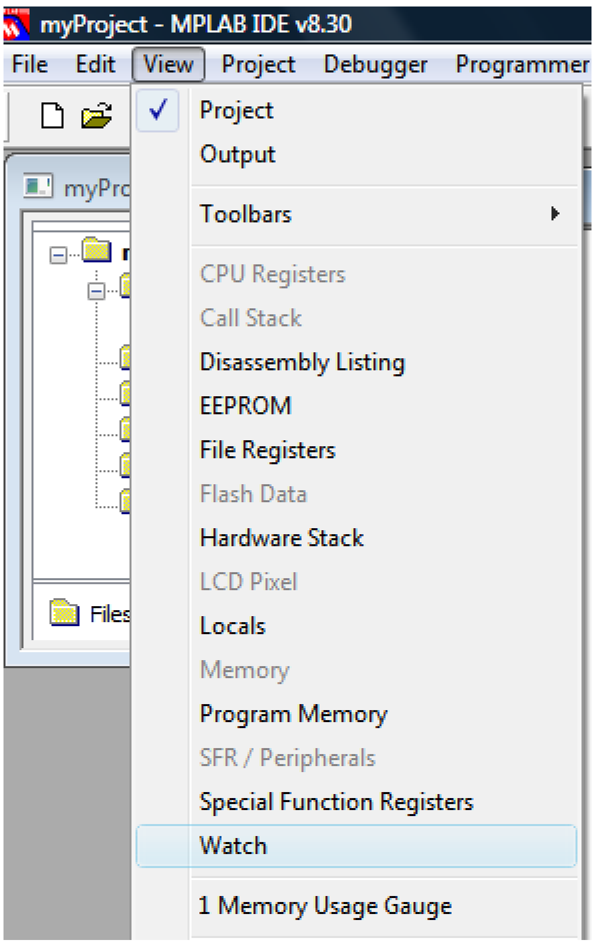
BUILD SUCCEED DOES NOT MEAN THAT YOUR PROGRAM IS CORRECT, IT SIMPLY MEANS THAT THERE ARE NO **SYNTAX** ERRORS FOUND, SO WATCH OUT FOR ANY LOGICAL ERRORS YOU MIGHT MAKE.

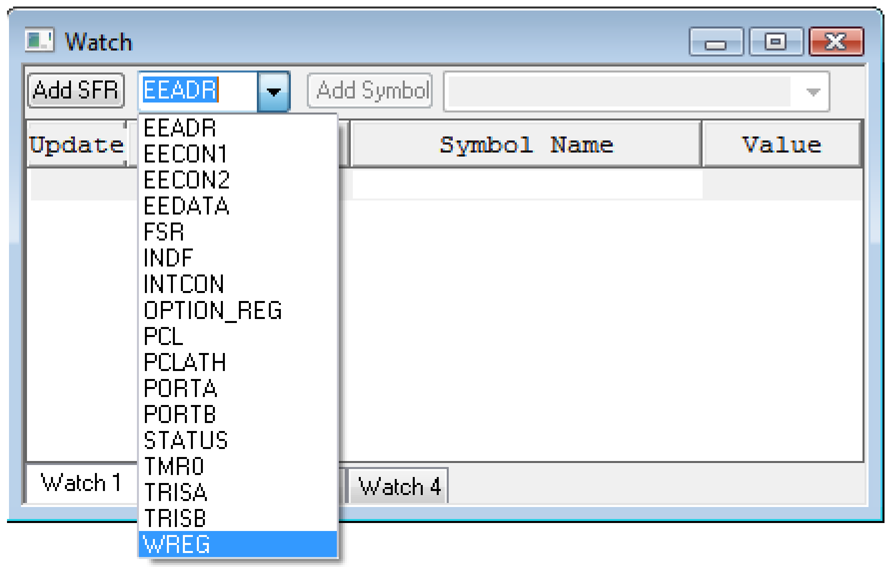
Notice that there are several warnings after building the file, warnings do not affect the execution of the program but they are worth reading. This warning reads: "Found opcode in column 1", column 1 is reserved for labels; however, we have written instructions (opcode) instead thus the warning.

TO SOLVE THIS WARNING SIMPLY TYPE FEW BLANK SPACES BEFORE EACH INSTRUCTION OR PRESS TAB

Preparing for simulation

Go to View Menu → Watch

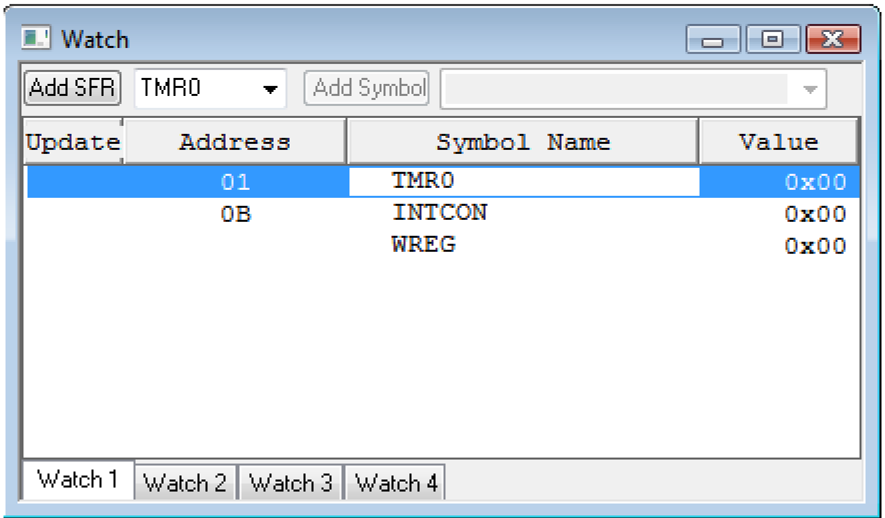




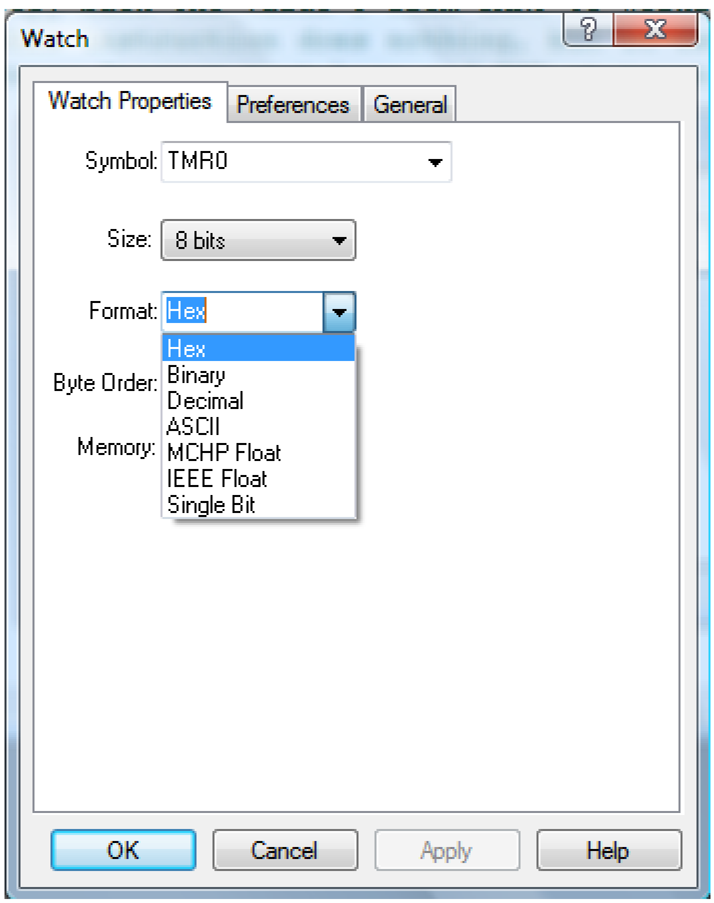
From the drop out menu choose the registers we want to watch during simulation and click ADD SFR for each one
Add the following:

- WREG: working register
- TMR0
- INTCON

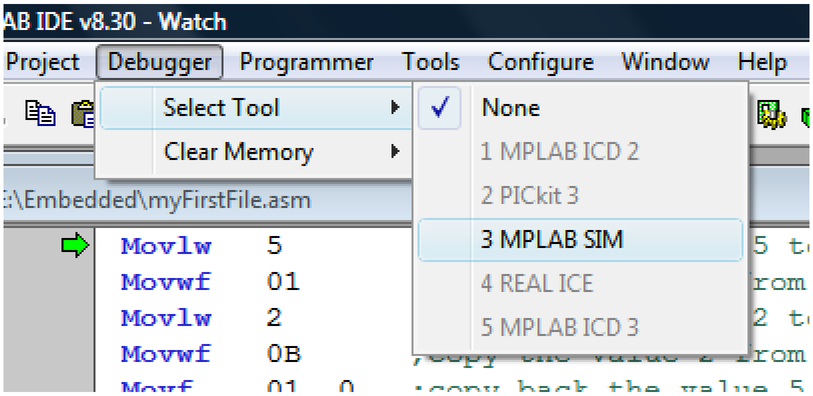
You should have the following:



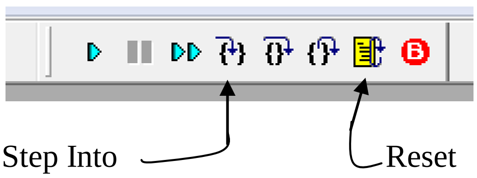
Notice that the default format is in hexadecimal, to change it (if you need to) simply right-click on the row → **Properties** and choose the new format you wish.

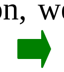


From the **Debugger Menu** → choose **Select Tool** → then **MPLAB SIM**



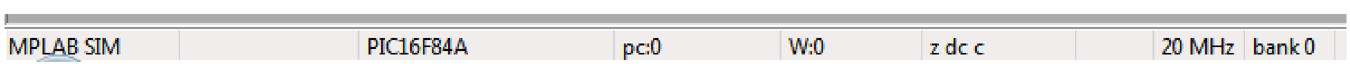
Now new buttons will appear in the toolbar:



1. To begin the simulation, we will start by resetting the PIC; do so by pressing the yellow reset button. A green arrow  will appear next to the first instruction.

   The green arrow means that the program counter is pointing to this instruction *which has not been executed yet.*

   Notice the status bar below:



   Keep an eye on the value of the program counter (pc: initially 0), see how it changes as we simulate the program

2. Press the "Step Into" button one at a time and check the Watch window each time an instruction executes; keep pressing "Step Into" until you reach the NOP instruction then STOP.

   Compare the results as seen in the Watch window with those expected.

Directives

Directives are not instructions. They are **assembler commands** that appear in the source code but are not usually translated directly into opcodes. They are used to control the **assembler**: its input, output, and data allocation. They are not converted to machine code (.hex file) and therefore not downloaded to the PIC.

The "END" directive

If you refer to the Appendix at the end of this experiment, you will notice that there is no end instruction among the PIC 16 series instructions, so what is "END"?

The "END" command is a directive which tells the MPLAB IDE that we have finished our program. It has nothing to do with neither the actual program nor the PIC.

The END should always be the last statement in your program
Anything which is written after the end command will not be executed and any variable names will be undefined.

Making your program easier to understand: the "equ" and "include" directives

As you have just noticed, it is difficult to write, read, debug or understand programs while dealing with memory addresses as numbers. Therefore, we will learn to use new directives to facilitate program reading.

The "EQU" directive

The equate directive is used to **assign** labels to numeric values. They are used to *DEFINE CONSTANTS* or to *ASSIGN NAMES TO MEMORY ADDRESSES OR INDIVIDUAL BITS IN A REGISTER* and then use the name instead of the numeric address.

```
Timer0      equ 01
Intcon      equ 0B
Workrg      equ 0
Movlw       5                   ; move the constant 5 to the working register
Movwf       Timer0              ; copy the value 5 from working register to TMR0 (address 01)
Movlw       2                   ; move the constant 2 to the working register
Movwf       Intcon              ; copy the value 2 from working register to INTCON (address 0B)
Movf        Timer0, Workrg      ; copy back the value 5 from TMR0 to working register
Nop                             ; this instruction does nothing, but it is important to write it for now
End
```

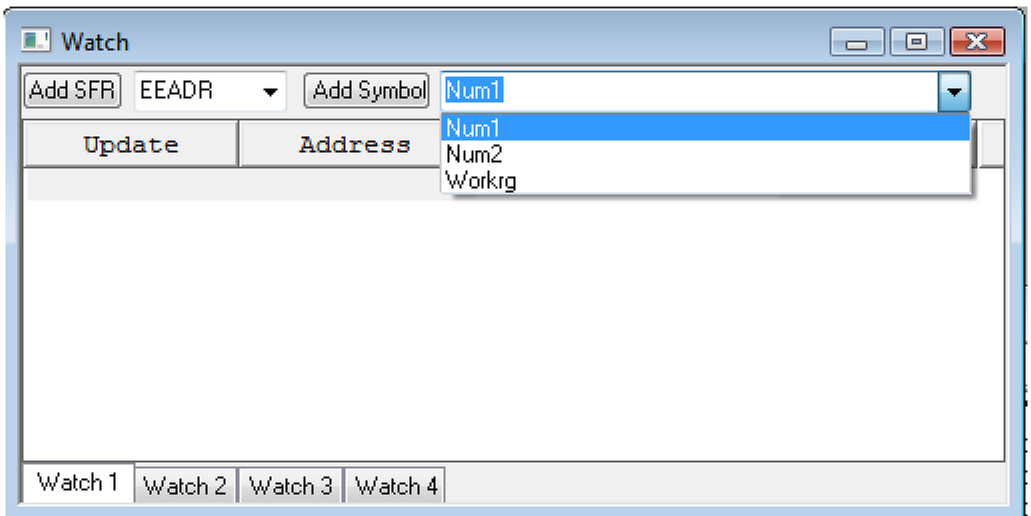
Notice how it is easier now to read and understand the program, you can directly know the actions executed by the program without referring back to the memory map by simply giving each address a name at the beginning of your program.

DIRECTIVES THEMSELVES **ARE NOT CASE-SENSITIVE** BUT THE **LABELS** YOU DEFINE **ARE**. SO YOU MUST USE THE NAME AS YOU HAVE DEFINED IT SINCE IT IS CASE-SENSITIVE.

As you have already seen, the GPRs in a memory map (lower part) do not have names as the SFRs (Upper part), so it would be difficult to use their addresses each time we want to use them. Here, the *"equate"* statement proves helpful.

```
Num1      equ 20            ;GPR @ location 20
Num2      equ 40            ;GPR @ location 40
Workrg    equ 0
Movlw     5                 ; move the constant 5 to the working register
Movwf     Num1              ; copy the value 5 from working register to Num1 (address 20)
Movlw     2                 ; move the constant 2 to the working register
Movwf     Num2              ; copy the value 2 from working register to Num2 (address 40)
Movf      Num1, Workrg      ; copy back the value 5 from Num1 to working register
Nop                         ; this instruction does nothing, but it is important to write it for now
End
```

When simulating the above code, you need to add Num1, Num2 to the watch window, however, since Num1 and Num2 are not SFRs but GPRs, you will not find them in the drop out menu of the "Add SFR", instead you will find them in the drop out menu of the "Add symbol".



The "INCLUDE" directive

Suppose we are to write a huge program that uses all registers. It will be a tiresome task to define all Special Function Registers (SFR) and bit names using "equate" statements. Therefore we use the include directive. The include directive calls a file which has all the equate statements defined for you and ready to use, its syntax is

#include "PXXXXXXX.inc"   where XXXXXX is the PIC part number

Older version of include without #, still supported.

Example: **#include "P16F84A.inc"**

The only **condition** when using the include directive is to use the names as Microchip defined them which are **ALL CAPITAL LETTERS** and **AS WRITTEN IN THE DATA SHEET**. If you don't do so, the MPLAB will tell you that the variable is undefined!

```
#include "P16F84A.inc"
Movlw      5          ; move the constant 5 to the working register
Movwf      TMR0       ; copy the value 5 from working register to TMR0 (address 01)
Movlw      2          ; move the constant 2 to the working register
Movwf      INTCON     ; copy the value 2 from working register to INTCON (address 0B)
Movf       TMR0, W    ; copy back the value 5 from TMR0 to working register
Nop                   ; this instruction does nothing, but it is important to write it for now
End
```

The "Cblock" directive

You have learnt that you can assign GPR locations names using the equate statements to facilitate dealing with them. Though this is correct, it is not recommended by Microchip as a good programming practice. Instead you are instructed to use cblocks when defining and declaring GPRs. So then, what is the use of the "equ" directive?

From now on, follow these two simple programming rules:

1. The **"EQU"** directive is used to define **constants**
2. The **"cblock"** is used to define **variables** in the data memory.

The cblock defines variables in sequential locations, see the following declaration

```
Cblock 0x35
        VarX
        VarY
        VarZ
endc
```

Here, VarX has the starting address of the cblock, which is 0x35, VarY has the sequential address 0x36 and VarZ the address of 0x37

What if I want to define variable at locations which are not sequential, that is some addresses are at 0x25 others at 0x40?

Simply use another cblock statement, you can add as many cblock statements as you need

The Origin "org" directive

The origin directive is used to place the instruction ***which exactly comes after it*** at the location it specifies.

Examples:

```
Org     0x00
Movlw   05          ;This instruction has address 0 in program memory
Addwf   TMR0        ;This instruction has address 1 in program memory
Org     0x04        ;Program memory locations 2 and 3 are empty, skip to address 4 where it contains
Addlw   08          ;this instruction

Org     0x13        ;WRONG, org only takes even addresses
```

In This Course, Never Use Any Origin Directives Except For Org 0x00 And 0x04, Changing Instructions' Locations In The Program Memory Can Lead To Numerous Errors.

The Concept of Bank Switching

Write, build and simulate the following program in your MPLAB editor. This program is very similar to the ones discussed above but with a change of memory locations.

```
#include "P16F84A.inc"
Movlw       5               ; move the constant 5 to the working register
Movwf       TRISA           ; copy the value 5 from working register to TRISA (address 85)
Movlw       2               ; move the constant 2 to the working register
Movwf       OPTION_REG      ; copy 2 from working register to OPTION_REG (address 81)
Movf        TRISA, W        ; copy back the value 5 from TRISA to working register
Nop                         ; this instruction does nothing, but it is important to write it for now
End
```

After simulation, you will notice that both TRISA and OPTION_REG were not filled with the values 5 and 2 respectively! But why?

Notice that the memory map is divided into two columns, each column is called a bank, here we have two banks: bank 0 and bank 1. In order to access bank 1, we have to switch to that bank first and same for bank 0. But how do we make the switch?

Look at the details of the STATUS register in the figure below, there are two bits RP0 and RP1, these bits control which bank we are in:

- If RP0 is 0 then we are in bank 0
- If RP0 is 1 then we are in bank 1

We can change RP0 by using the bcf/bsf instructions

- BCF STATUS, RP0 → RP0 in STATUS is 0 → switch to bank 0
- BSF STATUS, RP0 → RP0 in STATUS is 1 → switch to bank 1

*BCF: **B**it **C**lear **F**ile Register (makes a specified bit in a specified file register a 0)*
*BSF: **B**it **S**et **F**ile Register (makes a specified bit in a specified file register a 1)*

Try the program again with the following change and check the results:

```
#include "P16F84A.inc"
BSF         STATUS, RP0
Movlw       5                ; move the constant 5 to the working register
Movwf       TRISA            ; copy the value 5 from working register to TRISA (address 85)
Movlw       2                ; move the constant 2 to the working register
Movwf       OPTION_REG       ; copy 2 from working register to OPTION_REG (address 81)
Movf        TRISA, W         ; copy back the value 5 from TRISA to working register
BCF         STATUS, RP0
Nop                          ; this instruction does nothing, but it is important to write it for now
End
```

The "Banksel" directive

When using medium-range and high-end microcontrollers, it will be a hard task to check the memory map for each register we will use. Therefore the **BANKSEL** directive is used instead to simplify this issue. This directive is a command to the assembler and linker to generate bank selecting code to set the bank to the bank containing the designated *label*

Example:
BANKSEL TRISA will be replaced by the assembler, which will automatically know which bank the register is in and generate the appropriate bank selection instructions:

```
Bsf STATUS, RP0
Bcf STATUS, RP1
```

In the PIC16F877A, there are four banks; therefore you need two bits to make the switch between any of them. An additional bit in the STATUS register is RP1, which is used to make the change between the additional two banks.
One drawback of using BANKSEL is that it always generates two instructions even when the switch is between bank0 and bank1 which only requires changing RP0. We could write the code above in the same manner using Banksel

```
#include "P16F84A.inc"
Banksel     TRISA
Movlw       5                ; move the constant 5 to the working register
Movwf       TRISA            ; copy the value 5 from working register to TRISA (address 85)
Movlw       2                ; move the constant 2 to the working register
Movwf       OPTION_REG       ; copy 2 from working register to OPTION_REG (address 81)
Movf        TRISA, W         ; copy back the value 5 from TRISA to working register
Banksel     PORTA
Nop                          ; this instruction does nothing, but it is important to write it for now
End
```

Check the program memory window to see how BANKSEL is replaced in the above code and the difference in between the two codes in this page.

FLAGS

The PIC 16 series has three indicator flags found in the STATUS register; they are the C, DC, and Z flags. See the description below. Not all instructions affect the flags; some instructions affect some of the flags while others affect all the flags. Refer to the Appendix at the end of this experiment and review which instructions affect which flags.

The **MOVLW** and **MOVWF** do not affect any of the flags while the **MOVF** instruction affects the zero flag. Copying the register to itself does make sense now because if the file has the value of zero the zero flag will be one. Therefore the MOVF instruction is used to affect the zero flag and consequently know if a register has the value of 0. (Suppose you are having a down counter and want to check if the result is zero or not)

STATUS REGISTER

R/W-0	R/W-0	R/W-0	R-1	R-1	R/W-x	R/W-x	R/W-x
IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC[(1)]	C[(1)]
bit 7							bit 0

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

bit 6-5 **RP<1:0>:** Register Bank Select bits (used for direct addressing)
- `00` = Bank 0
- `01` = Bank 1
- `10` = Bank 2
- `11` = Bank 3

bit 2 **Z:** Zero bit
- `1` = The result of an arithmetic or logic operation is zero
- `0` = The result of an arithmetic or logic operation is not zero

bit 1 **DC:** Digit Carry/$\overline{\text{Borrow}}$ bit (`ADDWF`, `ADDLW`, `SUBLW`, `SUBWF` instructions)[(1)]
- `1` = A carry-out from the 4th low-order bit of the result occurred
- `0` = No carry-out from the 4th low-order bit of the result

bit 0 **C:** Carry/$\overline{\text{Borrow}}$ bit[(1)] (`ADDWF`, `ADDLW`, `SUBLW`, `SUBWF` instructions)[(1)]
- `1` = A carry-out from the Most Significant bit of the result occurred
- `0` = No carry-out from the Most Significant bit of the result occurred

Note 1: For $\overline{\text{Borrow}}$, the polarity is reversed. A subtraction is executed by adding the two's complement of the second operand. For rotate (`RRF`, `RLF`) instructions, this bit is loaded with either the high-order or low-order bit of the source register.

Types of Logical and Arithmetic Instructions and Result Destination

The PIC16 series logical and arithmetic instructions are easy to understand by just reading the instruction, for from the name you readily know what this instruction does. There are the ADD, SUB, AND, XOR, IOR (the ordinary ***I***nclusive ***OR***). They only differ by their operands and the result destination. The following table illustrates:

	Type I – Literal Type	**Type II – File Register Type**
Syntax	xxx***LW*** ***k*** *where k is constant*	xxx***WF*** ***f, d*** *where f is file register and* *d is the destination (F, W)*
Instructions	Addlw, sublw, andlw, iorlw and xorlw	Addwf, subwf, andwf, iorwf, xorwf
Operands	1. A literal (given by the instruction) 2. The working register	1. A file register in the data memory (either SFR or GPR) 2. The working register
Result destination	The working register only	Two Options: 1. **W**: the Working register 2. **F**: The same File given in the instruction
How does it work?	**W** = **L** *operation* **W**	**F** = **F** operation **W** The value of F is overwritten by the result, original value lost **W** = **F** operation **W** The value of F is the original value, result stored in working register instead
	The order is important in the subtract operation	

Examples (assuming you are using the include statement and appropriate equ statements for defining GPRs)	xorlw 0BB
	W = W ^ 0BB
	
	sublw .85
	W = 85_d – W
		Andwf TMR0, W
		W = TMR0 & W
		
		addwf NUM1, F
		NUM1 = NUM1 + W
		
		Subwf PORTA, F
		PORTA = PORTA - W

	Notice that in subtraction, the W has the minus sign	

Many other instructions of the PIC16 series instruction set are of Type II; refer back to the Appendix at the end of this experiment for study.

Starting Up with basic programs

Program One: Fibonacci Series Generator

In mathematics, the Fibonacci numbers are the following sequence of numbers:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89

The first two Fibonacci numbers are 0 and 1, and each remaining number is the sum of the previous two

```
include "p16f84a.inc"
Fib0    equ 20          ;At the end of the program the Fibonacci series numbers from 0 to 5 will
Fib1    equ 21          ;be stored in Fib0:Fib5
Fib2    equ 22
Fib3    equ 23
Fib4    equ 24
Fib5    equ 25

Clrw                    ;This instruction clears the working register, W = 0
clrf    Fib0            ;The clrf instruction clears a file register specified, here Fib0 = 0
movf    Fib0, w         ;initializing Fib1 to the value 1 by adding 1 to Fib0 and storing it in Fib1
addlw   1
movwf   Fib1

movf    Fib0, W         ; Fib2 = Fib1 + Fib0
addwf   Fib1, W
movwf   Fib2

movf    Fib1, W         ; Fib3 = Fib2 + Fib1
addwf   Fib2, W
movwf   Fib3

movf    Fib2, W         ; Fib4 = Fib3 + Fib2
addwf   Fib3, W
movwf   Fib4

movf    Fib3, W         ; Fib5 = Fib4 + Fib3
addwf   Fib4, W
movwf   Fib5
nop
end
```

1. Start a new MPLAB session, add the file ***example1.asm*** to your project
2. Build the project
3. Select **Debugger** ⇨ **Select Tool** ⇨**MPLAB SIM**
4. Add the necessary variables and the working register to the watch window (remember that user defined variables are found under the "**Add Symbol**" list)

5. Simulate the program step by step, analyze and study the function of each instruction. **Stop at the "nop" instruction**
6. Study the comments and compare them to the results at each stage and after executing the instructions
7. As you simulate your code, keep an eye on the MPLAB status bar below (the results shown in this status bar are not related to the program, they are for demo purposes only)

The status bar below allows you to instantly check the value of the flags after each instruction is executed
In the figure above, the flags are z, DC, C

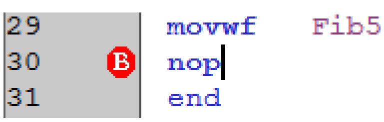
- A **capital letter** means that the value of the flag is **one**; meanwhile a **small letter** means a value of **zero**. In this case, the result is not zero; however, digit carry and a carry are present.

Another faster method of simulation: Run and break points

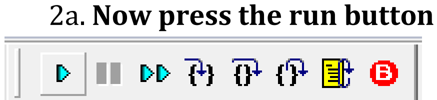
Many times you will need to make some changes to your code, additions, omissions and bug fixes. It is not then flexible to step into your code step by step to observe the changes you have made especially when your program is large. It would be a good idea to execute your code all at once or up to a certain point and then read the results from the watch window.

Now suppose we want to execute the Fibonacci series code at once - to do so, follows these steps:

1. Double click on the "nop" instruction (line 30), a red circle with a letter "**B**" inside is shown to the left of the instruction. This is called a breakpoint. Breakpoints instruct the simulator to stop code execution at this point. *All instructions **before** the breakpoint are only executed*

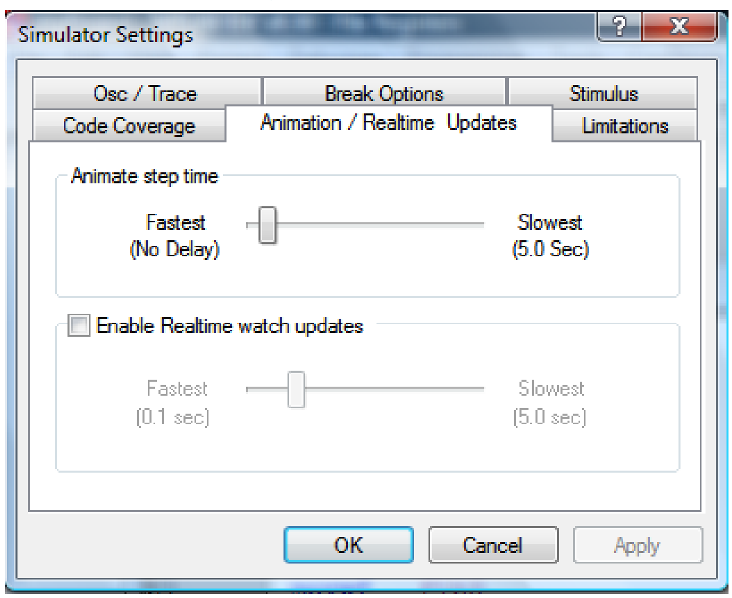


2a. **Now press the run button**



2b. Alternatively, you can instruct the IDE to automatically step into the code an instruction at a time by simply pressing "**animate**"

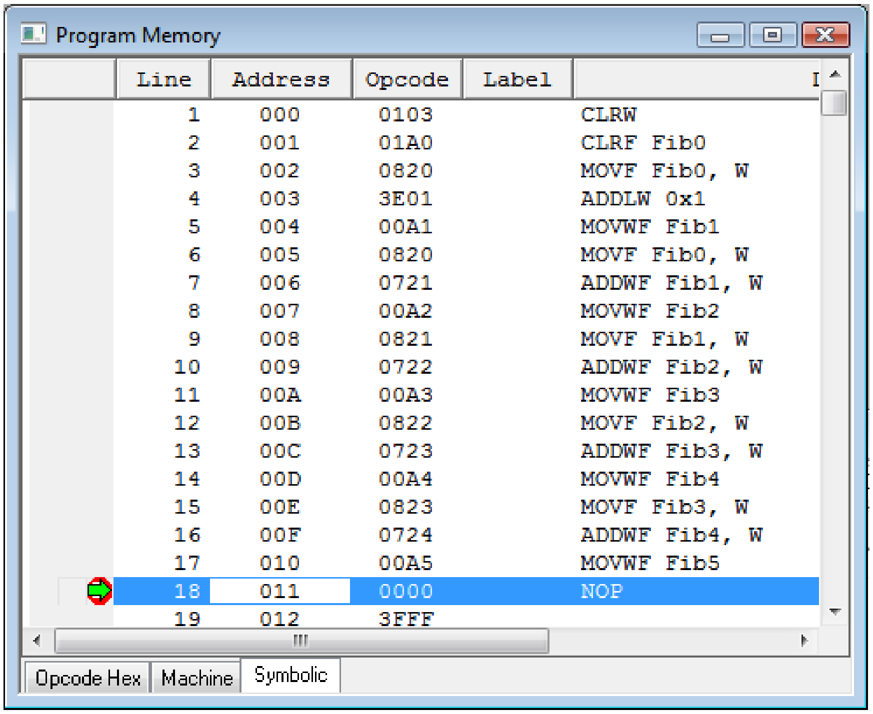
You can control the speed of simulation as follows:



1. **Debugger ⇘ Settings ⇘ Animation/ Real time Updates**
2. Drag the slider to set the speed of simulation you find convenient

Program Memory Space Usage

Though we have written about 31 lines in the editor, the total number of program memory space occupied is far less, remember that directives are not instructions and that they are not downloaded to the target microcontroller. To get an approximate idea of how much space does the program occupy: Select **View** ⇘ **Program Memory** ⇘ **Symbolic** tab



Line	Address	Opcode	Label	
1	000	0103		CLRW
2	001	01A0		CLRF Fib0
3	002	0820		MOVF Fib0, W
4	003	3E01		ADDLW 0x1
5	004	00A1		MOVWF Fib1
6	005	0820		MOVF Fib0, W
7	006	0721		ADDWF Fib1, W
8	007	00A2		MOVWF Fib2
9	008	0821		MOVF Fib1, W
10	009	0722		ADDWF Fib2, W
11	00A	00A3		MOVWF Fib3
12	00B	0822		MOVF Fib2, W
13	00C	0723		ADDWF Fib3, W
14	00D	00A4		MOVWF Fib4
15	00E	0823		MOVF Fib3, W
16	00F	0724		ADDWF Fib4, W
17	010	00A5		MOVWF Fib5
18	011	0000		NOP
19	012	3FFF		

Note that the last instruction written is “nop” (end is a directive). The total space occupied is only 18 memory locations

The “**opcode**” field shows the actual machine code of each instruction which is downloaded to the PIC

Program Two: Implementing the function Result = (X + Y) ⊕ Z

This example is quite an easy one, initially the variable X, Y, Z are loaded with the values which make the truth table

```
include "p16F84A.inc"

cblock  0x25
        VarX
        VarY
        VarZ
        Result
endc

        org     0x00
Main                            ;loading the truth table
        movlw B'01010101'       ;ZYX    Result
        movwf VarX              ;000    0       (bit7_VarZ, bit7_VarY, bit7_VarX)
        movlw B'00110011'       ;001    1       (bit6_VarZ, bit6_VarY, bit6_VarX)
        movwf VarY              ;010    1                    .
        movlw B'00001111'       ;011    1                    .
        movwf VarZ              ;100    1                    .
```

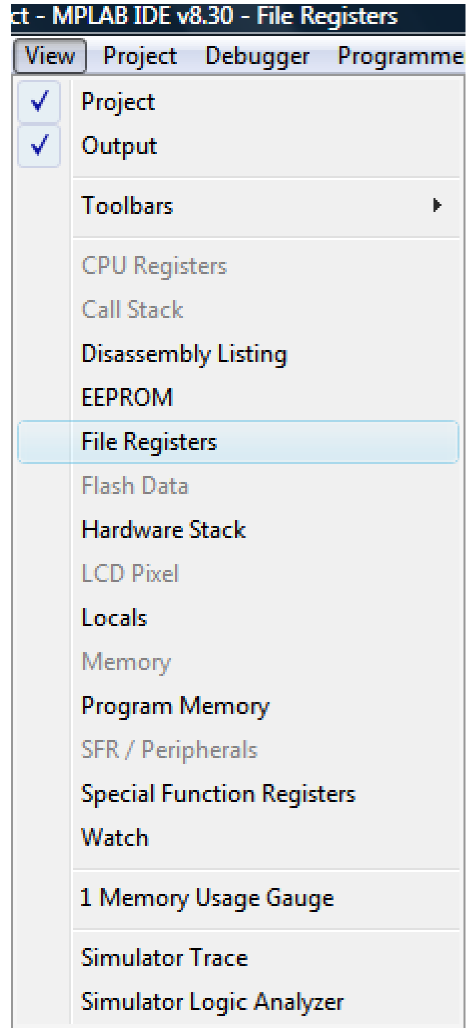
```
                                        ;101     0                          .
                                        ;110     0                          .
                                        ;111     0              (bit0_VarZ,  bit0_VarY,  bit0_VarX)
        movf    VarX, w
        iorwf   VarY, w
        xorwf   VarZ, w
        movwf   Result
        nop
        end
```

1. Start a new MPLAB session, add the file ***example2.asm*** to your project
2. Build the project
3. Select **Debugger** ➪ **Select Tool** ➪**MPLAB SIM**
4. Add the necessary variables and the working register to the watch window (remember that user defined variables are found under the "**Add Symbol**" list)
5. Simulate the program step by step, analyze and study the function of each instruction. **Stop at the "nop" instruction**
6. Study the comments and compare them to the results at each stage and after executing the instructions

Appendix A: Instruction Listing

Mnemonic, Operands		Description	Cycles	14-Bit Opcode				Status Affected	Notes
				MSb			LSb		
BYTE-ORIENTED FILE REGISTER OPERATIONS									
ADDWF	f, d	Add W and f	1	00	0111	dfff	ffff	C,DC,Z	1,2
ANDWF	f, d	AND W with f	1	00	0101	dfff	ffff	Z	1,2
CLRF	f	Clear f	1	00	0001	1fff	ffff	Z	2
CLRW	-	Clear W	1	00	0001	0xxx	xxxx	Z	
COMF	f, d	Complement f	1	00	1001	dfff	ffff	Z	1,2
DECF	f, d	Decrement f	1	00	0011	dfff	ffff	Z	1,2
DECFSZ	f, d	Decrement f, Skip if 0	1 (2)	00	1011	dfff	ffff		1,2,3
INCF	f, d	Increment f	1	00	1010	dfff	ffff	Z	1,2
INCFSZ	f, d	Increment f, Skip if 0	1 (2)	00	1111	dfff	ffff		1,2,3
IORWF	f, d	Inclusive OR W with f	1	00	0100	dfff	ffff	Z	1,2
MOVF	f, d	Move f	1	00	1000	dfff	ffff	Z	1,2
MOVWF	f	Move W to f	1	00	0000	1fff	ffff		
NOP	-	No Operation	1	00	0000	0xx0	0000		
RLF	f, d	Rotate Left f through Carry	1	00	1101	dfff	ffff	C	1,2
RRF	f, d	Rotate Right f through Carry	1	00	1100	dfff	ffff	C	1,2
SUBWF	f, d	Subtract W from f	1	00	0010	dfff	ffff	C,DC,Z	1,2
SWAPF	f, d	Swap nibbles in f	1	00	1110	dfff	ffff		1,2
XORWF	f, d	Exclusive OR W with f	1	00	0110	dfff	ffff	Z	1,2
BIT-ORIENTED FILE REGISTER OPERATIONS									
BCF	f, b	Bit Clear f	1	01	00bb	bfff	ffff		1,2
BSF	f, b	Bit Set f	1	01	01bb	bfff	ffff		1,2
BTFSC	f, b	Bit Test f, Skip if Clear	1 (2)	01	10bb	bfff	ffff		3
BTFSS	f, b	Bit Test f, Skip if Set	1 (2)	01	11bb	bfff	ffff		3
LITERAL AND CONTROL OPERATIONS									
ADDLW	k	Add literal and W	1	11	111x	kkkk	kkkk	C,DC,Z	
ANDLW	k	AND literal with W	1	11	1001	kkkk	kkkk	Z	
CALL	k	Call subroutine	2	10	0kkk	kkkk	kkkk		
CLRWDT	-	Clear Watchdog Timer	1	00	0000	0110	0100	$\overline{TO}$,$\overline{PD}$	
GOTO	k	Go to address	2	10	1kkk	kkkk	kkkk		
IORLW	k	Inclusive OR literal with W	1	11	1000	kkkk	kkkk	Z	
MOVLW	k	Move literal to W	1	11	00xx	kkkk	kkkk		
RETFIE	-	Return from interrupt	2	00	0000	0000	1001		
RETLW	k	Return with literal in W	2	11	01xx	kkkk	kkkk		
RETURN	-	Return from Subroutine	2	00	0000	0000	1000		
SLEEP	-	Go into standby mode	1	00	0000	0110	0011	$\overline{TO}$,$\overline{PD}$	
SUBLW	k	Subtract W from literal	1	11	110x	kkkk	kkkk	C,DC,Z	
XORLW	k	Exclusive OR literal with W	1	11	1010	kkkk	kkkk	Z	

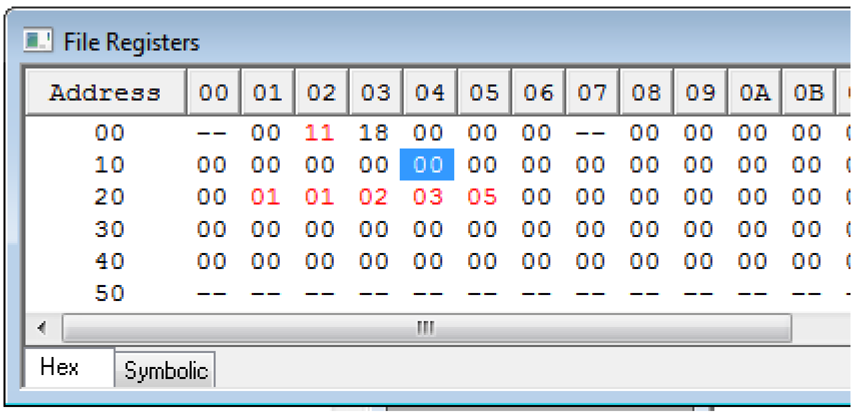
Another method to view the content of data memory is through the File Registers menu:



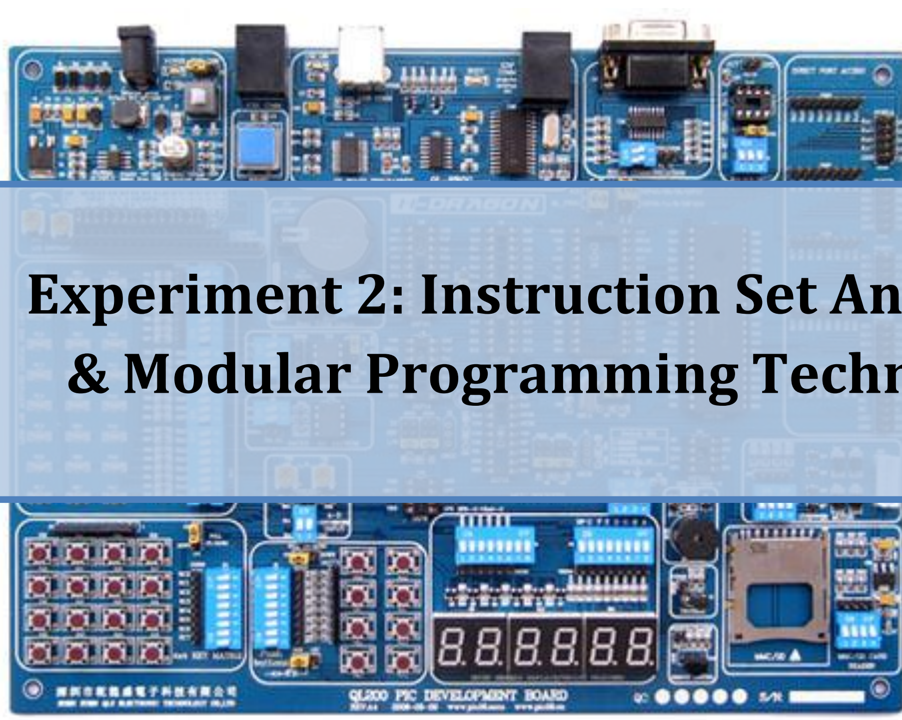
- ❖ Select **View** Menu ↳ **File Registers**

After building the Example1.asm codes, start looking at address 20 (which in our code corresponds to Fib0), to the right you will see the adjacent file registers from 21 to 2F.

Observe that **after code execution**, these memory locations are filed with Fibonacci series value as anticipated.



University of Jordan
Faculty of Engineering and Technology
Department of Computer Engineering
Embedded Systems Laboratory 0907334



2

Experiment 2: Instruction Set Analysis 2 & Modular Programming Techniques

Objectives

The main objectives of this experiment are to familiarize you with:

- Program flow control instructions
- Conditional and repetition structures
- The concept of modular programming
- Macros and Subroutines

Pre-lab requirements

Before starting this experiment, you should have already familiarized yourself with MPLAB software and how to create, simulate and debug a project.

Introducing conditionals

The PIC 16series instruction set has four instructions which implement a sort of conditional statement: ***btfsc*** , ***btfss, decfsz and incfsz*** instructions.

1. **btfsc** checks for the condition that a bit is clear: 0 (***B***it ***T***est ***F***ile, ***S***kip if ***C***lear)
2. **btfss** checks for the condition that a bit is set one: 1 (***B***it ***T***est ***F***ile, ***S***kip if ***Set***)
3. Review ***decfsz*** *and* ***incfsz*** *functions from the datasheet*

Example 1:

```
movlw 0x09
btfsc PORTA, 0
movwf  Num1
movwf  Num2
```

The above instruction tests bit 0 of PORTA and checks whether it is clear (0) or not

- If it is clear (0), the program will **skip** "movwf Num1" and will only execute "movwf Num2"
  Only Num2 has the value 0x09
- If it is set (1), it will not skip but **execute** "movwf Num1" and then **proceed** to "movwf Num2"
  In the end, both Num1 and Num2 have the value of 0x09

You have seen above that **if the condition fails**, the code will continue normally and both instructions will be executed.

Example 2:

```
        movlw 0x09
        btfsc PORTA, 0
        goto firstcondition
        goto secondCondition
Proceed
        …….. your remaining code
firstcondition
        movwf  Num1
        goto Proceed
secondCondition
        movwf  Num2
        goto Proceed
```

> *Firstcondition, secondCondition, and Proceed are called Labels, Labels are used to give names for a specific block of instructions and are referred to as shown above to change the program execution order.*

Example 2 is basically the same as Example 1 with one main difference:

- If it is clear (0), the program will **skip** "*goto firstcondition*" and will only execute "*goto secondCondition*", the program will then execute "*movwf Num2" and then "gotoProceed"*
  Only Num2 has the value 0x09
- If it is set (1), it will not skip but **execute** "*goto firstcondition*", the program will then execute "*movwf Num1" and then "gotoProceed"*
  Only Num1has the value 0x09

Conditional using Subtraction and how the Carry/$\overline{\text{Borrow}}$ flag is affected?

The Carry concept is easy when dealing with addition operations but it differs in borrow operations according to Microchip implementation.

Carry is a physical flag; you will find it in the STATUS register,
Borrow is not implemented; it is in your mind ☺
In the following examples we will show the status of the Carry/$\overline{\text{Borrow}}$ flag and how it differs between addition and subtraction operations:

Ex1) 99-66 10011001 – <u>01100110</u> Expect no borrow since 99 > 66 10011001+ <u>10011010</u> 2's complement of 66 ^{1}00110011 There is carry (C = 1), since Borrow is the complement of Carry, then Borrow is 0 (No borrow) which is correct	Ex 2) 66 – 99 01100110- <u>10011001</u> Expect borrow since 66 < 99 01100110+ <u>01100111</u> ^{0}11001101 There is no carry (C = 0), since Borrow is the complement of Carry, then Borrow is 1 (There is borrow) which is correct

Program One: Check if a value is greater or smaller than 10, if greater Result will have the ASCII value G, if smaller, it will have the ASCII value S.

```
include "p16F84A.inc"
cblock  0x25
        testNum
        Result
endc
        org     0x00
Main
        movf    testNum, W
        sublw   .10             ;10d - testNum
        btfss   STATUS, C
        goto    Greater         ;C = 0, that's B = 1, then testNum > 10
        goto    Smaller         ;C = 1, that's B = 0, then testNum < 10
Greater
        movlw A'G'
        movwf Result
         goto    Finish
Smaller
        movlw A'S'
        movwf Result
Finish
        nop
        end
```

1. Start a new MPLAB session, add the file ***example3.asm*** to your project
2. Build the project
3. Select **Debugger** ⇘ **Select Tool** ⇘**MPLAB SIM**
4. Add the necessary variables and the working register to the watch window (remember that user defined variables are found under the "**Add Symbol**" list)
5. Enter values into testNum, simulate the program step by step, concentrate on what happens at lines10-12
6. Keep an eye on the Flags at the status bar below while simulating the code
7. Enter other values lesser and greater and observe how the code behaves

❖ What is the value stored in Result when testNum = 10? Is this correct? Can you think of a solution?

Program Two: Counting the Number of Ones in a Register's Lower Nibble
Introducing simple conditional statements

This program will take a hexadecimal number as an input in the lower nibbles (bits 3:0) in a register called testNum.The number will be masked by anding it with 0F, (remember that 0 & Anything = 0, while 1 & anything will remain the same), we used masking because if the user accidentally wrote a number in the higher nibble (bits 3:0), it will be forced to zero. The number in the lower nibble will not be affected (anded with 1). The masked result will be saved in a register called tempNum.
Now tempNum will be rotated to the right, bit0 (least significant bit) will move to the C flag of the STATUS register after rotation. Then it will be tested whether it 0 or 1. If it is 1, the numOfOnes register will be incremented. Else the program proceeds. This operation will continue for 4 times (because the number of bits in the lower nibble is 4)

```
1   include "p16f84a.inc"
2
3   cblock 0x20
4           testNum                 ;GPR @ location 20
5           tempNum                 ;GPR @ location 21
6   endc
7
8   cblock 0x30
9           numOfOnes               ;GPR @ location 30
10  endc
11
12  org 0x00
13  clrf    numOfOnes      ;Initially number of ones is 0
14  movf    testNum, W     ;Since we only need to test the number of ones in the lower nibble, we
15                         ;mask them by 0F (preserve lower nibble and discard higher nibble)
16  andlw   0x0F           ;in case a user enters a number in the upper digit. Save masked result
17  movwf tempNum          ;in tempNum
18  rrf     tempNum, F     ;rotate tempNum to the right through carry, that is the least
19                         ;significant bit of tempNum (bit0) goes into the C flag of the
20                         ;STATUS register, while the old value of C flag goes into bit 7 of
21                         ;tempNum
```

Byte 8 bits							
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Higher 4 bits				**Lower 4 bits**			
Upper Nibble				**Lower Nibble**			

```
22 btfsc  STATUS, C     ;tests the C flag, if it has the value of 1, increment number of ones and
23 incf   numOfOnes, F;proceed, else proceed without incrementing
24 rrf    tempNum, F
25 btfsc  STATUS, C      ;Same as above
26 incf   numOfOnes, F
27 rrf    tempNum, F
28 btfsc  STATUS, C
29 incf   numOfOnes, F
30 rrf    tempNum, F
31 btfsc  STATUS, C
32 incf   numOfOnes, F
33 nop
34 end
```

As you can see in the above program, we did not write instructions to load testNum with an initial value to test; this code is general and can take any input. So, how do you test this program with general input?

After building your project, adding variables to the watch window and selecting MPLAB SIM simulation tool, simply double click on testNum in the watch window and fill in the value you want. Then Run the program.

Change the value of testNum and re-run the program again, check if numOfOnes hold the correct value.

Coding for efficiency: The repetition structures

You have observed in the code above that instructions from 18 to 32 are simply the same instructions repeated over and over four times for each bit tested.

Now we will introduce the repetition structures, similar in function to the ***"for"*** and ***"while"*** loops you have learnt in high level languages.

Program Three: Counting the Number of Ones in a Register's Lower Nibble Using a Repetition Structure

```
        include "p16f84a.inc"
        cblock 0x20
                testNum
                tempNum
        endc

        cblock 0x30
                numOfOnes
                counter         ;since repetition structures require a counter, one is declared
        endc

        org 0x00
        clrf    numOfOnes
        movlw  0x04             ;counter is initialized by 4, the number of the bits to be tested
```

```
            movwf counter
            movf    testNum, W
            andlw   0x0F
            movwf tempNum
Again
            rrf      tempNum, F
            btfsc    STATUS, C
            incf     numOfOnes, F
            decfsz  counter, F            ; The contents of register counter are decremented then test :
             goto     Again               ; if the counter reaches 0, it will skip to "nop" and program ends
            nop                           ; if the counter is > 0, it will repeat "goto Again"
            end
```

Introducing the Concept of Modular Programming

Modular programming is a software design technique in which the software is divided into several separate parts, where each part accomplishes a certain independent function. This *"Divide and Conquer"* approach allows for easier program development, debugging as well as easier future maintenance and upgrade.

Modular programming is like writing C++ or Java **functions**, where you can use the function many times only differing in the parameters. Two structures which are similar to functions are **Macros** and **Subroutines** ***which are used to implement modular programming.***

Subroutines

Subroutines are the closest equivalent to functions

- Subroutines start with a **Label** giving them a name and end with the instruction **return**

Examples:

doMath
        Instruction 1
        Instruction 2
        .
        .
        Instruction n
        return
	Process
	        Instruction 1
	        Instruction 2
	        .
	        .
	Calculate
	        Instruction 7
	        Instruction 8
	        return
	This is still one subroutine, no matter the number of labels in between

- Subroutines can be written anywhere in the program after the org and before the end directives
- Subroutines are used in the following way: Call subroutineName
- Subroutines are stored **once** in the program memory, each time they are used, they are executed from that location

- Subroutines alter the flow of the program, thus they affect the stack
  Example:

```
Main
        Instruction1
        Instruction2
        Call doMath
        Instruction4
        Instruction5
        Nop
        Nop

doMath
        Instruction35
        Instruction36
        Instruction37
        return
```

So what is the stack and how is it used?

Initially the program executes sequentially; instructions 1 then 2 then 3, when the instruction Call doMath is executed, the program will no longer execute sequentially, instead it will start executing Instructions35, then 36 then 37, when it executes **return**, what will happen? Where will it go and what instruction will be executed?

When the Call doMath instruction is executed, the address of the next instruction (which as you should already know id found in the program counter) Instruction4 is saved in a special memory called the stack. When the return instruction is executed, it reads the **last** address saved in the stack, which is the address of Instruction4 and then continues from there.

----Read section 2.4.1 of the P16F84A datasheet for more information regarding the stack----

Macros

Macros are declared in the following way (similar to the declaration of cblocks)

```
macroName    macro
                    Instruction 1
                    Instruction 2
                        .
                        .
                    Instruction n
             endm
```

- Macros should be declared before writing the code instructions. It is not recommended to declare macros in the middle of your program.
- Macros are used by only writing their name: macroName
- Each time you use a macro, it will be replaced by its body, refer to the example below. Therefore, the program will execute sequentially, the flow of the program will not change. The Stack is not affected

Programs Four and Five

The following simple program demonstrates the differences between using macros and subroutines. They essentially perform the same operation: Num2 = Num1 + Num2

Example4 using Macro

```
include "p16f84a.inc"

cblock 0x30
        Num1
        Num2
endc

Summation    macro
        movf    Num1, W   ;Macro Body
        addwf   Num2, F
        endm

        org 0x00
Main
        Movlw 4
        Movwf Num1
        Movlw 8
        Movwf Num2
        Summation
        Movlw 1
        Movwf Num1
        Movlw 9
        Movwf Num2
        Summation

finish
        nop
        end
```

Example5 using Subroutine

```
include "p16f84a.inc"

cblock 0x30
        Num1
        Num2
endc






        org 0x00
Main
        Movlw 4
        Movwf Num1
        Movlw 8
        Movwf Num2
        Call Summation
        Movlw 1
        Movwf Num1
        Movlw 9
        Movwf Num2
        Call Summation
        goto finish

Summation
        movf    Num1, W
        addwf   Num2, F
        return
finish
        nop
        end
```

Analyzing the two programs and highlighting the differences

For **both** applications, go to **View → Program Memory**, let's see the differences:

Opcode	Label	
3004	Main	MOVLW 0x4
00B0		MOVWF Num1
3008		MOVLW 0x8
00B1		MOVWF Num2
0830		MOVF Num1, W
07B1		ADDWF Num2, F
3001		MOVLW 0x1
00B0		MOVWF Num1
3009		MOVLW 0x9
00B1		MOVWF Num2
0830		MOVF Num1, W
07B1		ADDWF Num2, F
0000	finish	NOP
3FFF		
3FFF		
3FFF		

```
13      org 0x00
14  Main
15      Movlw   4
16      Movwf   Num1
17      Movlw   8
18      Movwf   Num2
19      Summation
20      Movlw   1
21      Movwf   Num1
22      Movlw   9
23      Movwf   Num2
24      Summation
25
26  finish
27      nop
28      end
```

Figure 1. The example using macros

In the program memory window, notice that the macro name is replaced by its **body**. The instructions movf Num1, W and addwf Num2, F replace the macro name @ lines 19 and 24. Using macros clearly affects the space used by the program as it increases due to code copy.

Address	Opcode	Label	
000	3004	Main	MOVLW 0x4
001	00B0		MOVWF Num1
002	3008		MOVLW 0x8
003	00B1		MOVWF Num2
004	200B		CALL Summation
005	3001		MOVLW 0x1
006	00B0		MOVWF Num1
007	3009		MOVLW 0x9
008	00B1		MOVWF Num2
009	200B		CALL Summation
00A	280E		GOTO finish
00B	0830	Summation	MOVF Num1, W
00C	07B1		ADDWF Num2, F
00D	0008		RETURN
00E	0000	finish	NOP
00F	3FFF		
010	3FFF		
011	3FFF		
012	3FFF		
013	3FFF		

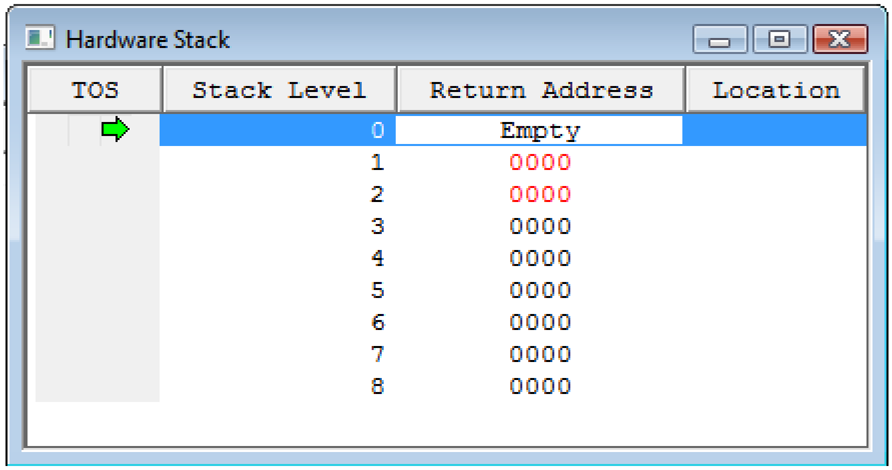
```
Main
    Movlw   4
    Movwf   Num1
    Movlw   8
    Movwf   Num2
    Call  Summation
    Movlw   1
    Movwf   Num1
    Movlw   9
    Movwf   Num2
    Call  Summation
    goto  finish

Summation
    movf    Num1, W
    addwf   Num2, F
    return
finish
    nop
    end
```

Figure 2. The example using subroutines

Now notice that the subroutine is only stored once in the program memory. No code replacement is present. You can also observe from the program memory that the program utilizing the macro executes sequentially from start to end, while the second program alters the program flow.

For ***Program Two***, do the following:

1. After building the project, go to **View → Hardware Stack**



2. Simulate the program up to the point when the green arrow points to the first Call Summation instruction.

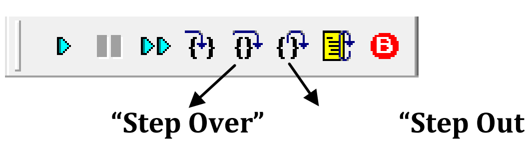
3. Look at the status bar below your MPLAB screen, what is the value of pc (program counter) (Note that the program counter has the address of the next instruction to be executed, that is Call Summation, Remember the instruction the arrow points to is not yet executed)
4. Now execute (use Single step) the Call Summation instruction.
   - After doing step4, what is the address of PC?
   - What is now stored at the TOS (Top of Stack)? (Refer to the Hardware Stack window)
   - How many levels of stack are used?
5. Now, continue simulating the program (subroutine). After executing the return instruction
   - What is the address of PC?
   - What is now stored at the TOS?
   - How many levels of stack are used?
6. Repeat the steps above for the second Call Summation instruction?

   The operation of saving the address on the stack - and any other variables - when calling a subroutine and later retrieving the address – and variables if any - when the subroutine finishes executing is called **context switching.**

Important Notes:

1. Assuming both a macro and a subroutine has the exact same body (same instructions), the execution of the subroutine takes slightly more time due to context switching.
2. You can use macro inside a macro, call a subroutine inside a subroutine, use a macro inside a subroutine and call a subroutine inside a macro

Further Simulation Techniques: Step Over and Step Out



Step Over is used when you want to execute the subroutine as a whole unit without seeing how each individual instruction is executed. It is usually used when you know that that the subroutine executes correctly and you are only interested to see the results.

1. Simulate program two up to the point when the green arrow points to the first Call Summation instruction.
2. Press Step Over, observe how the simulation runs

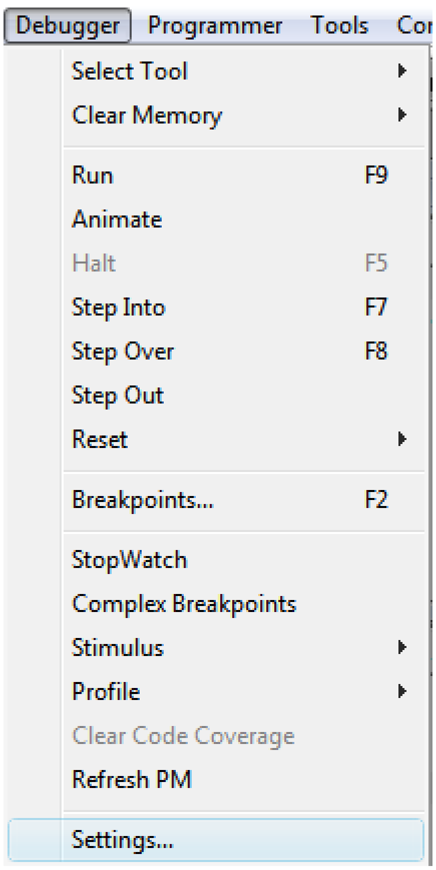
Step Out resembles Step Over, the only difference is that you use it **when you are already inside the subroutine and you want to continue** executing the subroutine as a whole unit without seeing how each **remaining** individual instruction is executed.

1. Simulate the program up to the point when the green arrow points to the first instruction inside the Summation subroutine: movf Num1, W
3. Press Step Out, , observe how the simulation runs

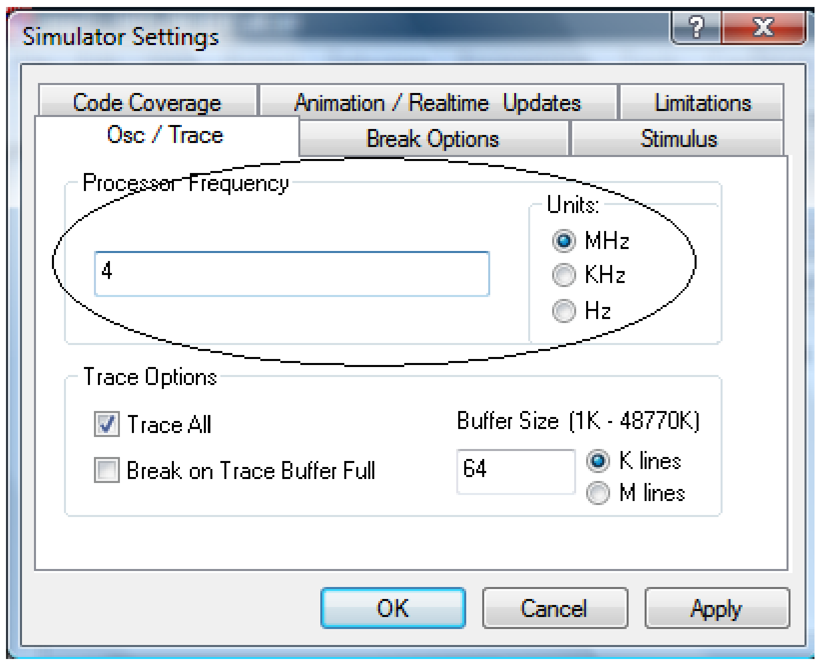
In both cases, the instruction are executed but you only see the end result of the subroutine

Time Calculation

To calculate the total time spent in executing the whole program or a certain subroutine, do the following:



1. Set the oscillator (external clock speed) as follows:

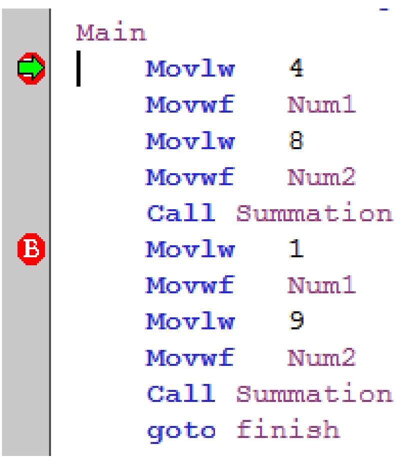


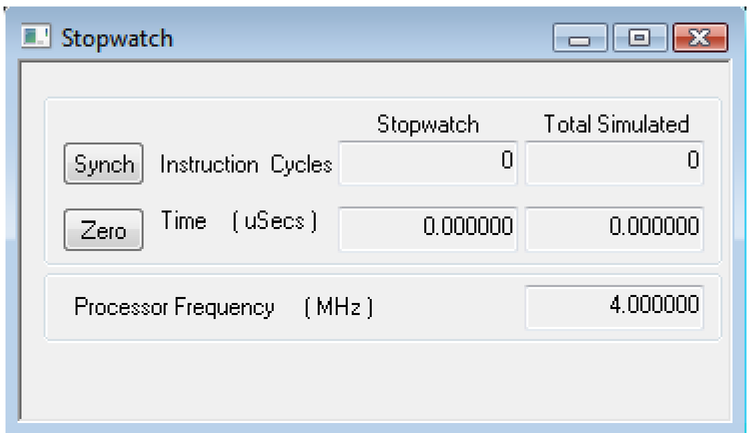
2. Set the processor frequency to 4MHz

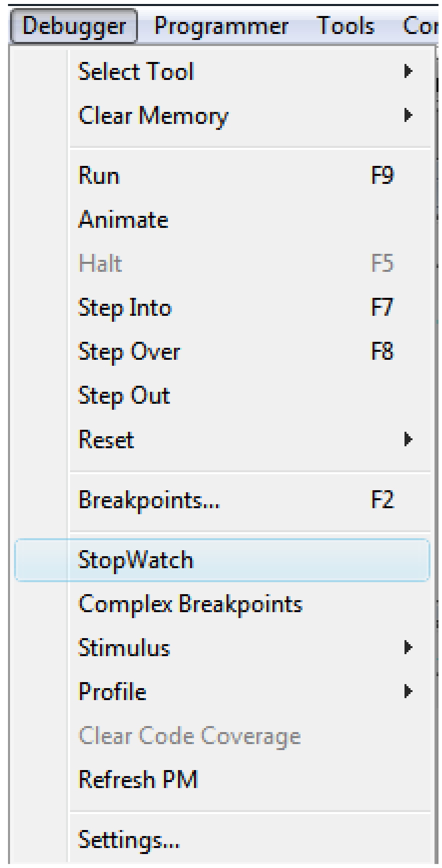
This means that each instruction cycle time is 4MHz/4 = 1MHz and T = 1/f = 1/MHz = 1µs

3. Now set breakpoints at the beginning and end of the code you want to calculate time for

4. Go to **Debugger → Stop Watch**







5. Now run the program, when the pointer stops at the first breakpoint → Press Zero
6. Run the program again. When the pointer reaches the second breakpoint, read the time from the stopwatch. This is the time spent in executing the code between the breakpoints.

Modular Programming

How to think Modular Programming?

Initially, you will have to read and analyze the problem statement carefully, based on this you will have to

1. Divide the problem into several separate tasks,
2. Look for similar required functionality

Non Modular and Modular Programming Approachs: Read the following problem statement

A PIC microcontroller will take as an input two sensor readings and store them in Num1 and Num2, it will then process the values and multiply both by 5 and store them in Num1_5, and Num2_5. At a later stage, the program will multiply Num1 and Num2 by 25 and store them in Num1_25 and Num2_25 respectively.

Analyzing the problem above, it is clear that it has the following functionality:

- Multiply Num1 by 5
- Multiply Num2 by 5
- Multiply Num1 by 25
- Multiply Num2 by 25

As you already know, we do not have a multiply instruction in the PIC 16F84A instruction set, so we do it by addition since:

```
2 x 3 = 2 + 2 + 2                        ; add 2 three times
7 x 9 = 7 + 7 + 7 + 7 + 7 + 7 + 7 + 7 + 7 ; add 7 nine times
```

So we write a loop as follows (example 4 x 9, add four nines), initially one nine is placed in W then we construct a loop to add the remaining 8 nines:

```
        movlw .8            ; because we put the first 4 in W, then we add the remaining 8 fours to it
        movwf counter
        movf  temp, w       ; 1st four in W
add
        addwf  temp, w
        decfsz counter, f   ; decrement counter, if not zero keep adding, else continue
        goto   add
        ; continue with code
```

A Non Modular Programming Approach		**Modular Programming Approach**	
Write multiply code for each operation above		Write one "Multiply by 5" code, use it two times Write one "Multiply by 25" code, use it two times Note that you do not need to write the "Multiply by 25" code from scratch, since 25 is 5x5, you can simply use "Multiply by 5" two times!	
	Code lines: 38		Code lines: 27
get Num1	1	get Num1	1
Write whole code to multiply Num1 by 5	7	call "multiply by 5" function	1
Store in Num1_5	1	Store in Num1_5	1
get Num2	1	get Num2	1
Write whole code to multiply Num2 by 5	7	call "multiply by 5" function	1
Store in Num2_5	1	Store in Num2_5	1
get Num1	1	get Num1	1
Write whole code to multiply Num1 by 25	7	call "multiply by 25" function	1
Store in Num1_25	1	Store in Num1_25	1
get Num2	1	get Num2	1
Write whole code to multiply Num2 by2 5	7	call "multiply by 25" function	1
Store in Num2_25	1	Store in Num2_25	1
goto finish	1	goto finish	1
nop	1	nop	1
		A single Multiply by 5 function	8
		A single Multiply by 5 function	5

include "p16f84a.inc"
cblock 0x30
    Num1
    Num2
    Num1_5
    Num2_5
    Num1_25
    Num2_25
    temp
    counter
endc
 	
    org 0x00
Main
    movf Num1, w ;Num1 x 5
    movwf temp
    movlw .4
    movwf counter
    movf temp, w
add1
    addwf temp, w
    decfsz counter, f
    goto add1
    movwf Num1_5
 	
    movf Num2, w ;Num2 x 5
    movwf temp
    movlw .4
    movwf counter
    movf temp, w
add2
    addwf temp, w
    decfsz counter, f
    goto add2
    movwf Num2_5
 	
    movf Num1, w ;Num1 x 25
    movwf temp
    movlw .24
    movwf counter
    movf temp, w
add3
    addwf temp, w
    decfsz counter, f
    goto add3
	include "p16f84a.inc"
	cblock 0x30
	    Num1
	    Num2
	    Num1_5
	    Num2_5
	    Num1_25
	    Num2_25
	    temp
	    counter
	endc
	
	    org 0x00
	Main
	    movf Num1, w ;Num1 x 5
	    call Mul5
	    movwf Num1_5
	
	    movf Num2, w ;Num2 x 5
	    call Mul5
	    movwf Num2_5
	
	    movf Num1, w ;Num1 x 25
	    call Mul25
	    movwf Num1_25
	
	    movf Num2, w ;Num2 x 25
	    call Mul25
	    movwf Num2_25
	    goto finish
	
	Mul5
	    movwf temp
	    movlw .4
	    movwf counter
	    movf temp, w
	add
	    addwf temp, w
	    decfsz counter, f
	    goto add
	    return
	
	Mul25
	    movwf temp
	    call Mul5
	    movwf temp

```
        movwf Num1_25

        movf    Num2, w     ;Num2 x 25
        movwf temp
        movlw .24
        movwf counter
        movf    temp, w
add4
        addwf  temp, w
        decfsz  counter, f
        goto     add4
        movwf Num2_25
        goto     finish
finish
        nop
        end
```

```
        call Mul5
        return
finish
        nop
        end
```

Notes on passing parameters to subroutines

Subroutines and macros are **general** codes; they work on many variables and generate results. So how do we tell the macro/subroutine that we want to work on this specific variable?

We have two approaches:

Place the input at the working register
Take the output from the working register

Example:

```
Main
   Movlw 03                   ;input to W
   Call     MUL_by4
   Movwf Result1              ;output from W
   Movlw 07                   ;input to W
   Call     MUL_by4
   Movwf Result2              ;output from W
   Nop
      .
      .
MUL_by4
   Movwf temp
   Rlf      temp,F
   Rlf      temp, F
   Movf   temp, W             ;place result in W
   Return
```

Store the input(s) in external variables
Load the output(s) from external variables

Example:

```
   Movf   Num1, W        ;load Num with Num1
   Movwf Num
   Call     MUL_by4
   Movf    Result, W     ;read the result and store
   Movwf Result1         ;it in Result1
   Movf   Num2, W        ;load Num with Num2
   Movwf Num
   Call     MUL_by4
   Movf   Result, W      ;read the result and store
   Movwf Result2         ;it in Result2

MUL_by4
   Rlf      Num,F
   Rlf      Num, W
   Movwf Result               ;place result in W
   Return
```

In this approach, the MUL_by4 subroutine takes the input from W (movwf), processes it then places the result back in W. Notice that we initially load W by the numbers we work on (here 03 and 07) then we take their values from W and save them in Result1 and Result2 respectively	In this approach the MUL_by4 subroutine expects to find the input in Num and saves the output in Result. Therefore, before calling the subroutine we load Num by the value we want (here Num1) and then take the value from Result and save it in Result1. The same is repeated for Num2
This approach is useful when the subroutine/macro has only one input and one output	This approach is useful when the subroutine/macro takes many inputs and produces multiple outputs

Special types of subroutines: Look up tables

Look up tables are a special type of subroutines which are used to retrieve values depending on the input they receive. They are invoked in the same as any subroutine: Call tableName

They work on the basis that they change the program counter value and therefore alter the flow of instruction execution

The **retlw** instruction is a **return** instruction with *the benefit that it returns a value in W* when it is executed.

Syntax:

```
lookUpTableName
        addwf  PCL, F   ;add the number found in the program counter to PCL (Program counter)
        nop
        retlw  Value         ;if W has 1, execute this
        retlw  Value         ;if W has 2, execute this
        retlw  Value
        …
        retlw  Value
```

Value can be in any format: decimal, hexadecimal, octal, binary and ASCII. It depends on the application you want to use this look-up table in.

Program Six: Displaying the 26 English Alphabets

This program works as follows:

Counter is loaded with the number 1 because we want to get the first letter of the alphabet, when we call the look-up table, it will retrieve the letter 'A'. The counter is incremented by 1 and then checked if we have reached the 26th letter of the alphabet (27 – the initial 1), if not we proceed to display the second letter 'B' and the third 'C' and so on. When we have displayed all the alphabets, counter will have the value 27 after which the program exits.

```
1   include "p16f84a.inc"
2   cblock  0x25
3           counter                         ;holds the number of Alphabet displayed
4           Value                           ;holds the alphabet value
5   endc
6           org 0x00
7   Main
8           movlw  1                        ;Initially no alphabet is displayed
9           movwf  counter
10  Loop
11          movf    counter, W
12          call    Alphabet                ;display Alphabet
13          movwf  Value
14          incf    counter, F              ;Each time, increment the counter by 1
15          movf    counter, w              ;if counter reaches 27, exit loop else continue
16          sublw   .27
17          btfss   STATUS, Z
18          goto    Loop
19          goto    finish
20  Alphabet
21          addwf  PCL, F
22          nop
23          retlw   'A'
24          retlw   'B'
25          retlw   'C'
26          retlw   'D'
27          retlw   'E'
28                .
29                .
30          retlw   'Z'
31  finish
32          nop
33          end
```

1. Complete the look-up table above with the missing alphabet
2. Add both counter and value to the watch window.
3. Place a breakpoint @ instruction 14: incf counter, F
4. Run the program, keep pressing run and observe the values of the variables in the Watch window

Appendix A: Documenting your program

It is a good programming practice to document your program in order to make it easier for you or others to read and understand it. For that reason we use comments. A proper way of documenting your code is to write **a functional comment,** which is a comment that **describes the function** of one or a set of instructions. Comments are defined after a semicolon (;) and are **not** read by MPLAB IDE

BSF STATUS, RP0

; Switch to Bank 1	Good comment	√
; Set the RP0 bit in the Status Register to 1	Bad Comment, no new added info	✗

How to professionally document your program?

At the beginning of your program, you are encouraged to add the following header which gives an insight to your code, its description, creator, version, date of last revision, etc… Most importantly, it is encouraged to document the necessary connections and classify them as input/output.

```
;*****************************************************************************************
; * Program name: Example Program
; * Program description: This program …….
; *
; * Program version: 1.0
; * Created by Embedded lab engineers
; * Date Created: September 1st, 2008
; * Date Last Revised: September 16th, 2008
;*****************************************************************************************
; * Inputs:
; *       Switch 0 (Emergency) to RB0 as interrupt
; *       Switch 1 (Start Motor) to RB1
; *       Switch 2 (Stop Motor) to RB2
; *       Switch 3 (LCD On) to RB3
; * Outputs:
; *       RB4 to Motor
;*        RB5 to Green LED (Circuit is powered on)
;*****************************************************************************************
```

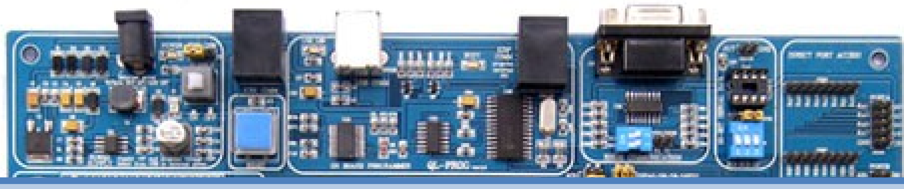
1. Your code declarations go here: *includes, equates, cblocks, macros, origin, etc…*
2. Your code goes here…
3. When using subroutines/macros, it is advised to add a header like this one before each to properly document and explain the function of the respected subroutine/macro.

```
;*****************************************************************************************
;* Subroutine Name: ExampleSub
;* Function: This subroutine multiplies the value found in the working register by 16
;* Input: Working register
;* Output: Working register * 16
;****************************************
```

Appendix B: Instruction Listing

Mnemonic, Operands		Description	Cycles	14-Bit Opcode MSb			LSb	Status Affected	Notes
BYTE-ORIENTED FILE REGISTER OPERATIONS									
ADDWF	f, d	Add W and f	1	00	0111	dfff	ffff	C,DC,Z	1,2
ANDWF	f, d	AND W with f	1	00	0101	dfff	ffff	Z	1,2
CLRF	f	Clear f	1	00	0001	1fff	ffff	Z	2
CLRW	-	Clear W	1	00	0001	0xxx	xxxx	Z	
COMF	f, d	Complement f	1	00	1001	dfff	ffff	Z	1,2
DECF	f, d	Decrement f	1	00	0011	dfff	ffff	Z	1,2
DECFSZ	f, d	Decrement f, Skip if 0	1 (2)	00	1011	dfff	ffff		1,2,3
INCF	f, d	Increment f	1	00	1010	dfff	ffff	Z	1,2
INCFSZ	f, d	Increment f, Skip if 0	1 (2)	00	1111	dfff	ffff		1,2,3
IORWF	f, d	Inclusive OR W with f	1	00	0100	dfff	ffff	Z	1,2
MOVF	f, d	Move f	1	00	1000	dfff	ffff	Z	1,2
MOVWF	f	Move W to f	1	00	0000	1fff	ffff		
NOP	-	No Operation	1	00	0000	0xx0	0000		
RLF	f, d	Rotate Left f through Carry	1	00	1101	dfff	ffff	C	1,2
RRF	f, d	Rotate Right f through Carry	1	00	1100	dfff	ffff	C	1,2
SUBWF	f, d	Subtract W from f	1	00	0010	dfff	ffff	C,DC,Z	1,2
SWAPF	f, d	Swap nibbles in f	1	00	1110	dfff	ffff		1,2
XORWF	f, d	Exclusive OR W with f	1	00	0110	dfff	ffff	Z	1,2
BIT-ORIENTED FILE REGISTER OPERATIONS									
BCF	f, b	Bit Clear f	1	01	00bb	bfff	ffff		1,2
BSF	f, b	Bit Set f	1	01	01bb	bfff	ffff		1,2
BTFSC	f, b	Bit Test f, Skip if Clear	1 (2)	01	10bb	bfff	ffff		3
BTFSS	f, b	Bit Test f, Skip if Set	1 (2)	01	11bb	bfff	ffff		3
LITERAL AND CONTROL OPERATIONS									
ADDLW	k	Add literal and W	1	11	111x	kkkk	kkkk	C,DC,Z	
ANDLW	k	AND literal with W	1	11	1001	kkkk	kkkk	Z	
CALL	k	Call subroutine	2	10	0kkk	kkkk	kkkk		
CLRWDT	-	Clear Watchdog Timer	1	00	0000	0110	0100	$\overline{TO}$,$\overline{PD}$	
GOTO	k	Go to address	2	10	1kkk	kkkk	kkkk		
IORLW	k	Inclusive OR literal with W	1	11	1000	kkkk	kkkk	Z	
MOVLW	k	Move literal to W	1	11	00xx	kkkk	kkkk		
RETFIE	-	Return from interrupt	2	00	0000	0000	1001		
RETLW	k	Return with literal in W	2	11	01xx	kkkk	kkkk		
RETURN	-	Return from Subroutine	2	00	0000	0000	1000		
SLEEP	-	Go into standby mode	1	00	0000	0110	0011	$\overline{TO}$,$\overline{PD}$	
SUBLW	k	Subtract W from literal	1	11	110x	kkkk	kkkk	C,DC,Z	
XORLW	k	Exclusive OR literal with W	1	11	1010	kkkk	kkkk	Z	

University of Jordan
Faculty of Engineering and Technology
Department of Computer Engineering
Embedded Systems Laboratory 0907334



3 Experiment 3: Basic Embedded System Analysis and Design

Objectives

- Empowering students with logical and analytical skills to solve real life system design problems
- To become familiar with the process of system requirement analysis and definition, system and subsystem design, flow analysis and flowchart design, software design and optimization
- Stressing software and hardware co-design techniques by introducing the *Proteus* IDE package

Starting-Up System Design

When we attempt to design a system that is required to perform complex tasks, it is essential that one thinks about the design flow and establish an overall system design before immediately jumping into implementation and coding in order for the program be written flawlessly and smoothly and the system functions correctly. In this way you don't waste time writing a flawed incomplete program, or which addresses the wrong problem or which is missing some flow scenarios.

A well-established diagramming technique is the flow chart which tracks down system execution flow. A flowchart is a schematic representation of an algorithm, showing the steps as different shapes, and their order by connecting them with unidirectional arrows. Flowcharts are used in designing or documenting programs. As programs get more complex, flowcharts help us follow and maintain the program flow. This makes a program easier to write, read, and understand. Other techniques used are state machines which are not covered in this course.

Complex systems need be broken into smaller pieces where each carries out few simple related tasks of the overall system. The system is thus built from these simple subsystems. One need only care about how these subsystems interface with each other. Subroutines allow the programmer to divide the program into smaller parts which are easier to code. In system design methodology, this is called the "Divide and Conquer" approach.

The basic steps in system design are:

Step 1: Requirements Definition

1. Reading the problem statement for what is needed to do, divide if it is complex.
2. What do I need to solve? Should I do it in software or hardware …
3. Determine the inputs and outputs for the hardware.

Step 2: System and Subsystem Design

4. Partition overall architecture into appropriate sub-systems.
5. Draw a detailed flowchart for each sub-systems

Step 3: Implementation

6. Translate flowcharts into code
7. Integrate subsystem into one code/design

Step 4: System Testing and Debugging

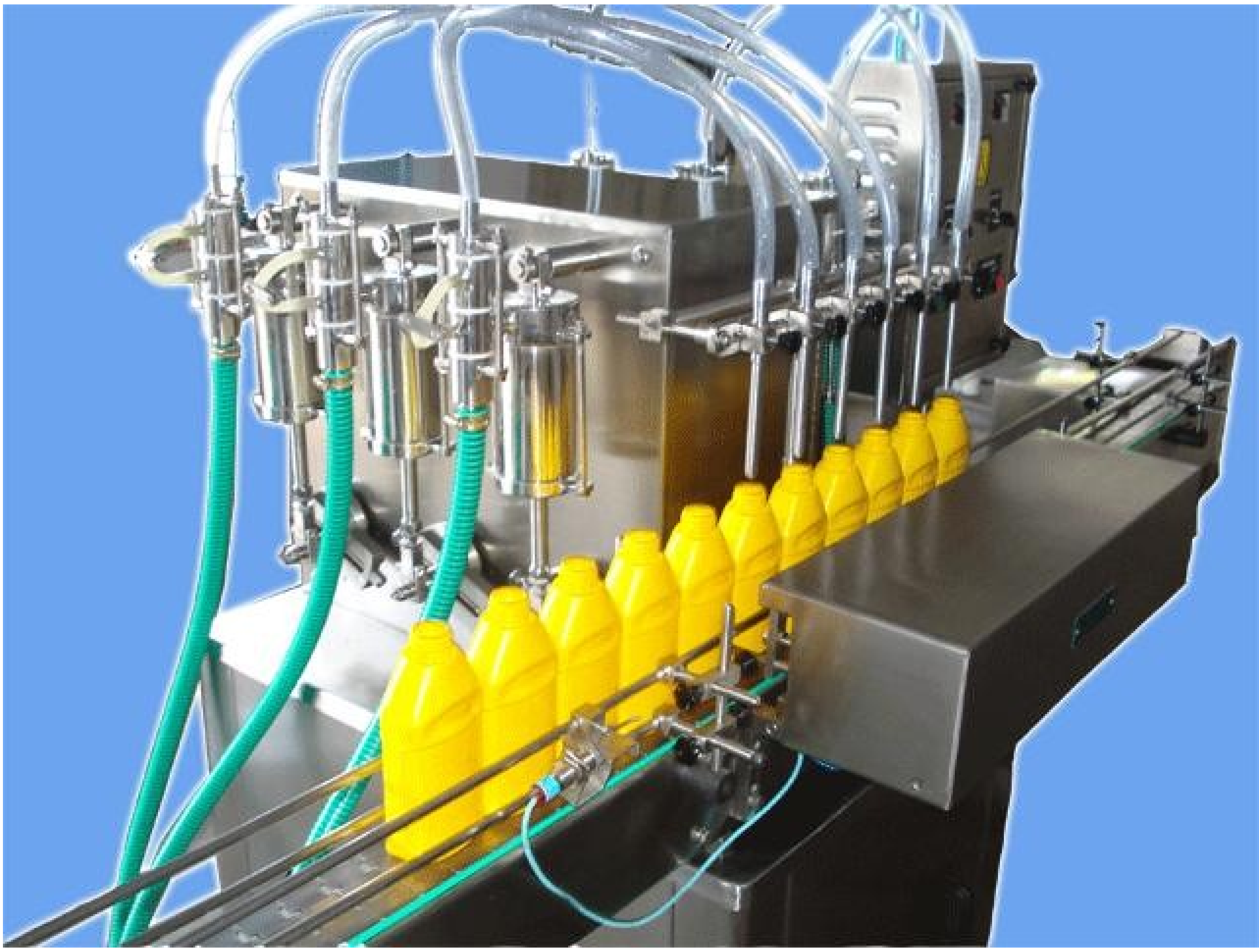
8. Run the program/hardware and see if it works correctly. If not, attempt to fix the program by reviewing the above steps and refining your design along with it

The above steps prove essential as programs get harder and larger. Next we will present a real life example from the industrial automation field.

Example – An Industrial Filling Machine

Problem Statement

We are to design an embedded system which controls a filling machine that works as follows: Empty bottles move on a conveyer belt, when a bottle is detected, the conveyor belt stops, a filling machine starts working for a specified period of time after which the filling machine stops. The total number of filled bottles is increased by one and shown on a common cathode 7-Segments display, the conveyor belt starts again and the machine does the same thing for the next bottle and so on. When the total number of bottles reaches nine the machine stops for manual packaging. Meanwhile, one LED lights on an 8-LED-row and moves back and forth. The conveyor belt does not start again until the resume button is pressed. Moreover, the LED array turns off! See the figure on the next page for the machine layout:



A Typical Filling Machine

Step1: Requirements Definition and Analysis

Now we will analyze the problem statement above and determine the required hardware and their role as input or output.

Output means a signal need be sent from the PIC to external hardware for control purposes. Input means a signal is received from external hardware into the PIC for processing. Processing means a certain code which does the job is required; this subroutine is internal processing and doesn't interact with the outside world!

Empty bottles move on a conveyer belt, when a bottle is detected, the conveyor belt stops

- ❖ There is a motor which controls the conveyor: "conveyor motor". Output
- ❖ There is a sensor which detects the presence of a bottle: "bottle sensor". Input

A filling machine starts working for a specified period of time after which the filling machine stops

- ❖ There is a pump/motor which is turned on/off to fill the bottle: "filling motor". Output
- ❖ We need a mechanism to calculate this time period. Processing

The total number of filled bottles is increased by one and shown on a common cathode 7-Segments display

- Clearly we need some sort of a counter. Memory location (GPR) reserved
- We need to output the value of this counter to a 7-segment display. Output

The conveyor belt starts again and the machine does the same thing for the next bottle and so on. When the total number of bottles reaches nine the machine stops for manual packaging.

- Continuously check for counter value if it reaches 9. Processing

Meanwhile, one LED lights on an 8-LED-row and moves back and forth. The conveyor belt does not start again until the resume button is pressed. Moreover, the LED array turns off!

- We need a code to control the LED lights. Output
- We need a mechanism to check for the resume button key press. Input

As you have seen above, we need to interact with external components; outputs like the motors, 7-Segments and the LEDs, as well as inputs from sensors or switches. Almost any embedded system needs to transfer digital data between its CPU and the outside world. This transfer achieved through input /output ports.

A quick look to the 16F84A or 16F877A memory maps will reveal multiple I/O ports: PORTA and PORTB for the 16F84A, and the additional PORTC, PORTD and PORTE for the 16F877A. Each port has its own TRISx Register which controls whether this PORTx will be an input port, output port, or a combination of both (individual bits control).

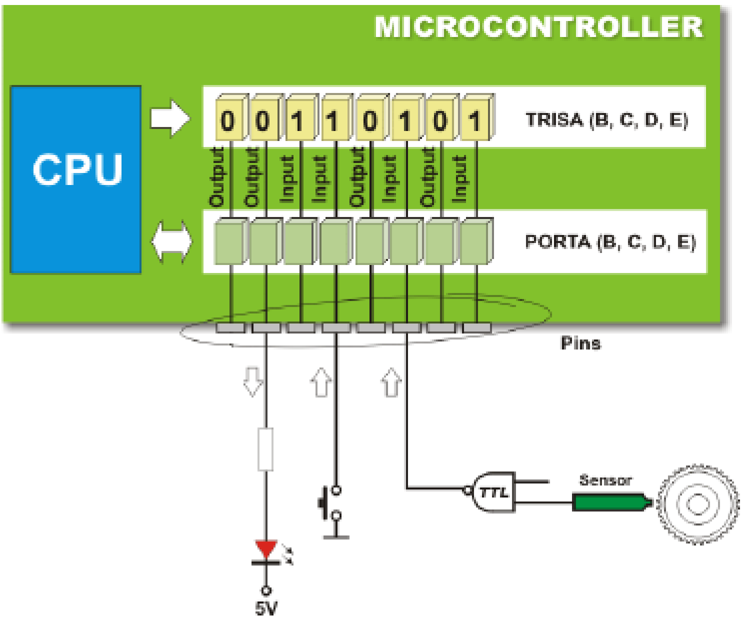
Ports A and E have a special configuration.

PORTA pins are multiplexed with **analog inputs** for the A/D converters. The operation of each pin is selected by clearing/setting the appropriate control bits in the **ADCON1**.

Instructions needed to configure all PORTA and E pins as general digital I/O pins :

```
BANKSEL    ADCON1
MOVLW      06H       ;set PORTA as general
MOVWF      ADCON1    ;Digital I/O PORT
```

PIC microcontrollers' ports are general-purpose bi-directional digital ports. The state of *TRISx* Register controls the direction of the *PORTx* bits. A logic one in a bit position configures the PIC to act as an input and if it has a zero to act as an output. However, a pin can only act as either input or output at any one time but not simultaneously. This means that each pin has a **distinct direction** state.



Examples:

Movlw 0x0F Movwf TRISB	Clrf TRISC	Clrf TRISD Comf TRISD, F	Movlw B'00110011' Movwf TRISB
The high nibble of PORTB is output, low nibble is input	Whole PORTC as output	Whole PORTD as input	Bits 2, 3, 6, 7 as output Bits 0, 1, 4, 5 as input

How to decide whether microcontroller's ports must be configured as inputs or outputs?
Input ports "Get Data" from the outside world into the microcontroller while output ports "Send Data" to the outside world.

- LEDs, 7-Segment displays, motors, and LCDs (write mode) that are interfaced to microcontroller's ports should be configured as output.
- Switches, push buttons, sensors, keypad and LCDs (read mode) that are interfaced to microcontroller's ports should be configured as input.

For the above filling machine example, we will use the following configuration.

Inputs:
- RA2: Bottle sensor
- RA3: Resume button

Outputs:
- RB0 to RB7: LEDs
- RC0: Machine motor ON/OFF
- RC1: Filling machine ON/OFF
- RD0 to RD6: 7-Segments outputs from "a" to "g" respectively

Step 2: System and Subsystem Design

Divide the overall system into appropriate sub-systems. The design of a subsystem includes:

(a) Defining the processes/functions that are carried out by the subsystem.
(b) Determining the input and output of the subsystem (Subsystem Interface)

Commonly, programs have "Initial" and "Main" subroutines, the Initial subroutine is used to initialize all ports, SFRs and GPR's used in the program and thus is only executed once at system startup, the Main subroutine contains all the subroutines which perform the functions of the system, many applications require that these functions be carried out repeatedly, thus the program loops through the Main subroutine code infinitely.

Note: when designing a system, expect not that you should reach the same system design as your friends/colleagues. Each one of you has her/his own thinking style and therefore designs the system differently; some might divide a certain problem into two subsystems, others into three or four. As long as you achieve a simple, easy to understand, maintainable and correct fully working system, then the goal is achieved! Therefore, the following subsystem design of the above problem is not the only one to approach and solve the problem. You may divide your subsystems differently depending on the philosophy and system structure you deem as appropriate.

Step 3: Implementation

As introduced before, the system should start with an initial subroutine. The nature of the system requires it runs continuously, consequently, the program code will loop through specific subroutines which implement the system function, we have decided to implement the code in three Major and two Minor subroutines – aside from the Initial subroutine:

Major Subroutines (in body of the Main):

Update_Seven_Seg subroutine: reads the total number of bottles filled and displays it on the 7-segment display.

Test_and_Process subroutine: waits for bottle, stops the conveyor, fills the bottle, and restarts the conveyor.

Test_Resume subroutine: checks if total number of bottles filled is nine, if so, stops the machine, continuously rotates the LEDs and tests for resume button press (this is done by calling the LEDs subroutine)

Minor Subroutines (outside the body of Main, called by those inside):

LEDs: moves the LED in the LED array back and forth and testing for the resume button press meanwhile.

Simplest_Delay: introduces a software delay used to give enough time for the LED to be seen as on.

The Initial and Main Codes:

```
Main
        Call Initial                ; Initialize Ports, SFRs and GPR's
Main_Loop
        Call Update_Seven_Seg       ; Test the number of Bottles and displays it on the 7-Seg.
        Call Test_and_Process       ; Keep testing the bottle sensor, if bottle found, process it,
                                    ; else wait until a bottle is detected
        Call Test_Resume            ; Check if No. of bottles is 9, if yes test if resume button is
                                    ; pressed, else skip and continue code
        goto Main_Loop              ; Do it again
Initial
        CLRF        BottleNumber        ; Start count display from zero
        BANKSEL     TRISD               ; Set register access to bank 1
        CLRF        TRISC               ; Set up all bits of PORTC as outputs
        CLRF        TRISD               ; Set up all bits of PORTD as outputs, connected to
                                        ; Common Cathode 7- Segments Display
        CLRF        TRISB               ; Set up all bits of PORTB as outputs, connected to
                                        ; LED array
        MOVLW       0x0C                ; Set up bits (1-2) of PORTA as inputs; RA3:
        MOVWF       TRISA               ; resume button, RA2: bottle sensor, others not used
        BANKSEL     ADCON1
        MOVLW       06H
        MOVWF       ADCON1              ;set  PORTA as general Digital I/O PORT
        BANKSEL             PORTA
        CLRF                PORTB       ; Initially, all LEDs are off
        BSF                 PORTC, 0    ; Start conveyer motor
        RETURN
```

Subsystem Flow Chart Analysis and Code Implementation

Clearly, the signals sent to the 7-Segments display are not decimal values but according to the 7Segments layout. Refer to the Hardware Guide for more information.

We have to translate the decimal number of bottles found in the bottle counter: BottleNumber to the appropriate common cathode 7-Segments number representation.

To do so we define the representations as constants and use a Look-up table to get the correct representation for each bottle number.

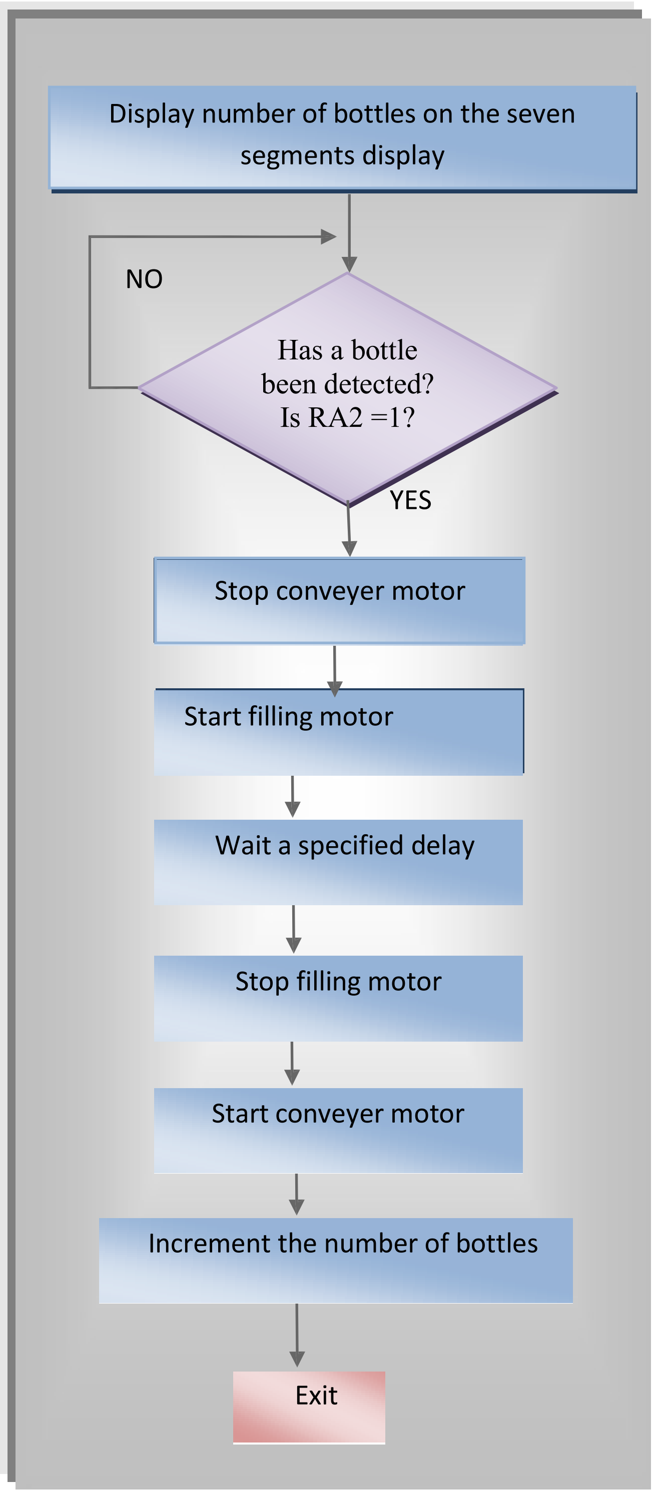
```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Update_Seven_Seg subroutine ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
This subroutine returns the appropriate common cathode 7-Segments
representation of the number of bottles in order for it to be displayed by the
consecutive subroutine
```

```
; Assuming the order is dp g f e d c b a

Zero    equ    B'00111111'
One     equ    B'00000110'
Twoo    equ    B'01011011'
               .
               .
Nine    equ    B'01101111'
```

```
Update_Seven_Seg

        Movf    BottleNumber,W
        Addwf   PCL, F
        Retlw   Zero
        Retlw   One
        Retlw   Two
        Retlw   Three
        Retlw   Four
        Retlw   Five
        Retlw   Six
        Retlw   Seven
        Retlw   Eight
        Retlw   Nine
```

;;; Test_and_Process subroutine;;;

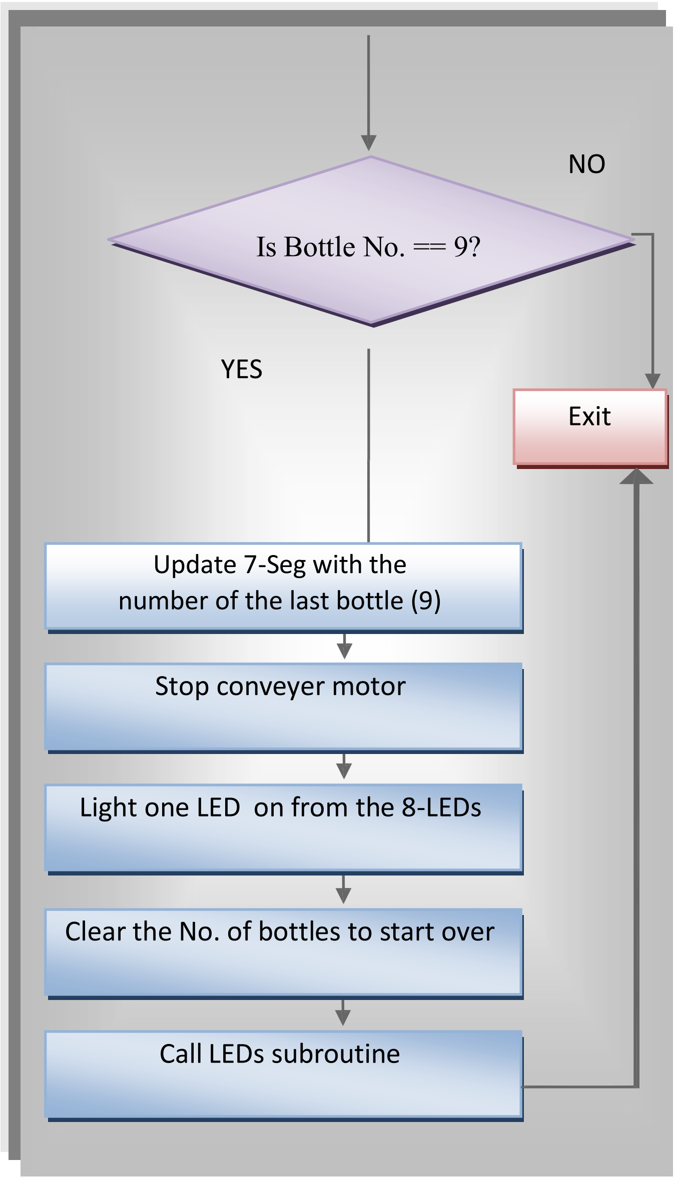
This subroutine tests if a bottle is present or not, if a bottle is detected, the conveyor motor stops, the filling machine starts working for a specified period of time after which the filling machine stops. The conveyor belt starts moving again. Finally the number of bottles is incremented




```
Test_and_Process
        movwf PORTD     ; display on the 7-Seg
poll
        btfss   PORTA,2   ; Test the bottle sensor
        goto    poll
        bcf     PORTC,0    ; stop conveyer motor
        bsf     PORTC,1    ; start filling motor
        call    Simplest_Delay;Insert  delay
        bcf     PORTC,1    ; stop filling motor
        bsf     PORTC,0    ; start conveyer motor
        incf    BottleNumber,F
        return
```

;;; Test_Resume Subroutine;;;

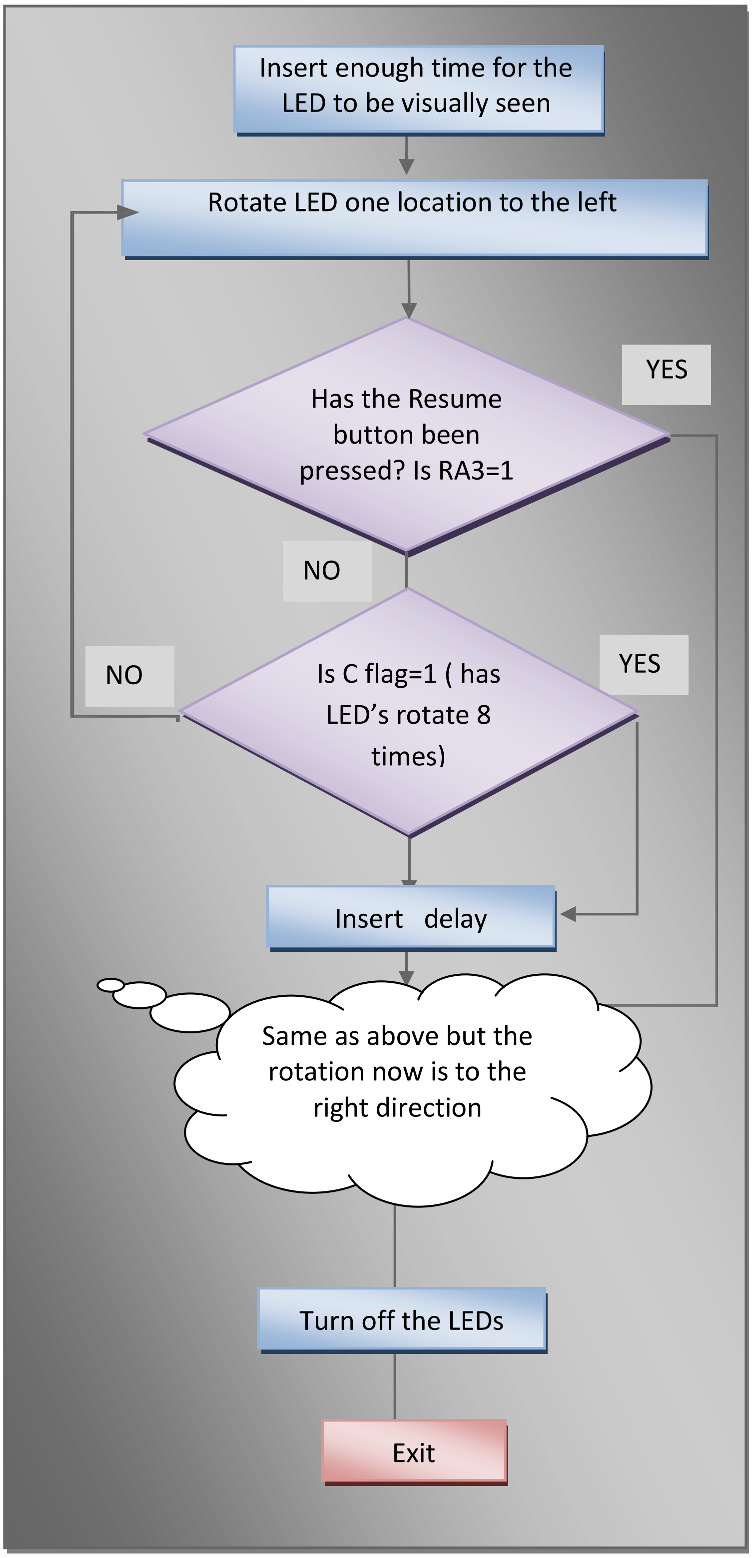
This subroutine checks if the total number of bottles reaches nine, if not it will exit, if yes the conveyer motor stops for manual packaging. Meanwhile one LED lights on an 8-LED-row and moves back and forth. The conveyor belt does not start again until the resume button is pressed




```
Test_Resume
        movf    BottleNumber, w
        sublw   .9
        btfss   STATUS, Z
        goto    fin1
        call    Update_Seven_Seg
        movwf   PORTD           ; display on the 7-seg
        bcf     PORTC, 0        ; stop conveyer motor
        bsf     PORTB, 0        ; light 1 LED
        bcf     STATUS,C
        clrf    BottleNumber    ; Reset System
        call    LEDs            ; rotate LEDs
fin1
        return
```

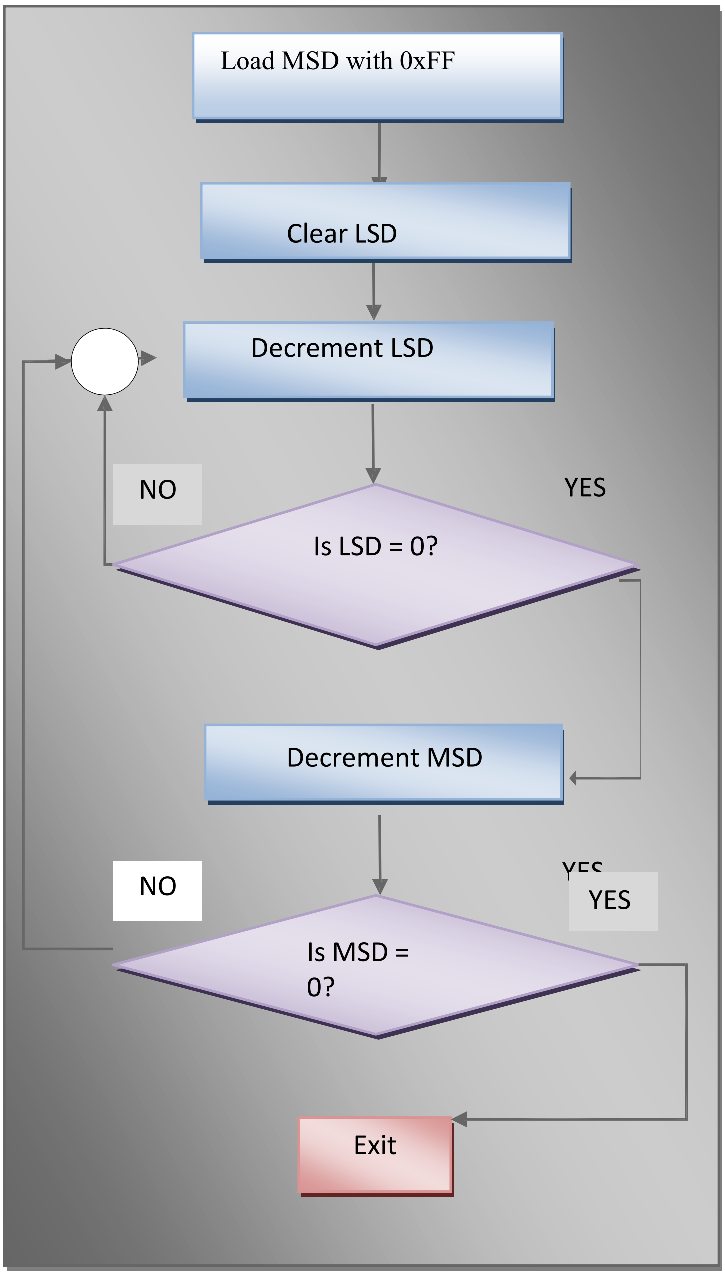
;; LEDs Subroutine;;;

This subroutine lights one LED on an 8-LED-row and continuously moves back and forth in this fashion. In between, the resume button is checked. If pressed, the conveyor motor starts again and the LED array turns off else the LEDs keep rotating and the resume button checked.




```
LEDs
Rotate_Left
        Call   Simplest_Delay
        rlf     PORTB, F
        btfsc   PORTA, 3        ;check Resume button
        goto    fin
        btfss   STATUS, C
        goto    Rotate_Left
Rotate_Right
        call    Simplest_Delay
        rrf     PORTB , F
        btfsc   PORTA, 3        ;check Resume button
        goto    fin
        btfss   STATUS, C
        goto    Rotate_Right
        goto    Rotate_Left
fin
        clrf    PORTB
        return
```

;; Simplest_Delay Subroutine;;
This subroutine inserts delay to be used as a digit delay in 7-seg multiplexing and as LED delay in the LED's array



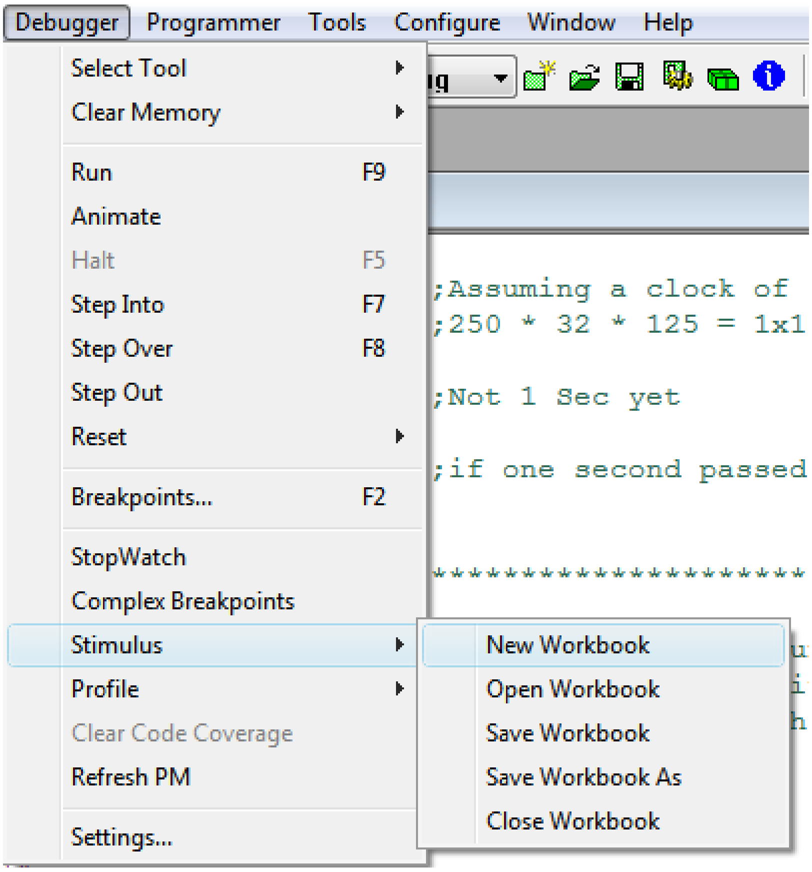

```
Simplest_Delay
        Movlw       0xFF
        movwf       msd
        clrf        lsd
loop2
        decfsz      lsd, f
        goto        loop2
        decfsz      msd, f
        goto        loop2
        return
```

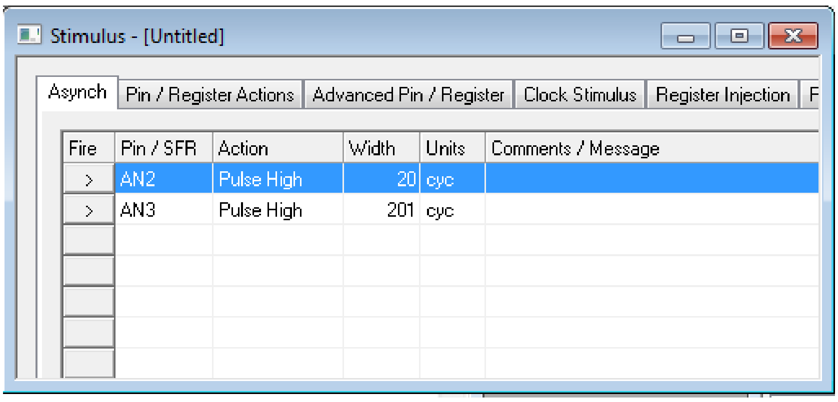
How to Simulate This Code in MPLAB?

You have learnt so far that in order to simulate inputs to the PIC, you usually entered them through the Watch window. However, this is only valid and true when you are dealing with internal memory registers. In order to simulate external inputs to the PIC pins, we are to use what is called a Stimulus.

There are multiple actions which you can apply to an input pin, choose whatever you see as appropriate to simulate your program. Here we have chosen to simulate the button press as a pulse.

1. Add RA2(AN2) and RA3(AN3) to the Stimulus window and BottleNumber to Watch window.





2. Place a break point at Instruction BTFSS PORTA,2 in the Test and Process subroutine. This will allow us to change the reading of the bottle sensors.

3. Place another break point at Instruction BTFSC PORTA, 3 in the LEDs subroutine. This will allow us to change the reading of the resume button.

4. Run your code, you will go to the First break point then press “Step Into” you will observe that you have stuck in loop.

5. Now Press *“Fire”,* the arrow next to the RA2 in the Stimulus pin, what do you observe?

6. Now press “Step Into” again , observe how the value of BottleNumber change.

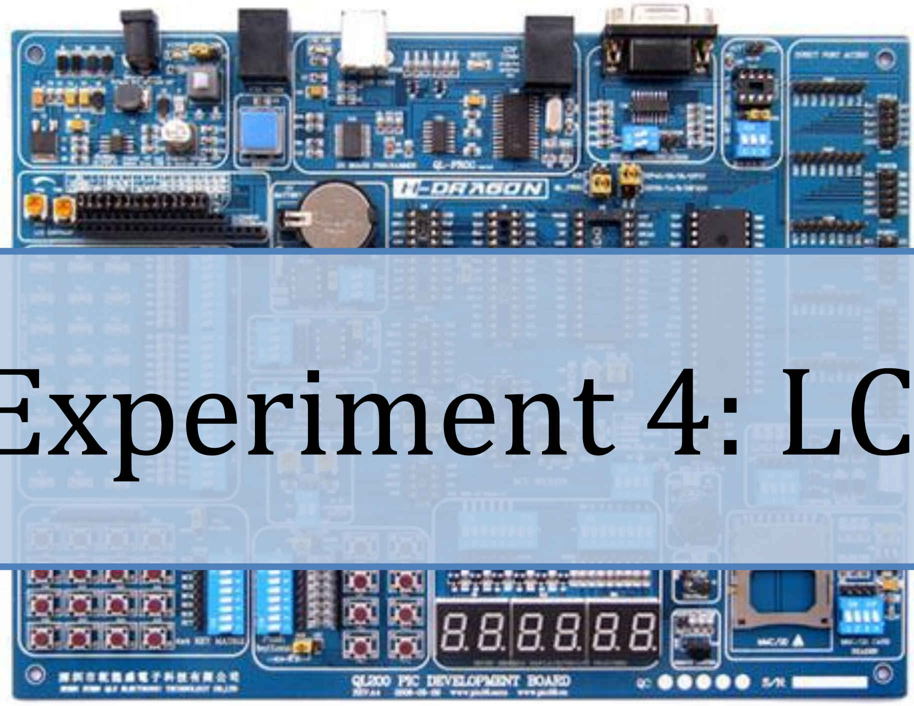
7. press “Run” then “fire” again, observe how the value in BottleNumber changes whenever you reach the first breakpoint.
   Note: You will reach the second breakpoint when nine bottles were detected.

8. Press “Step Into “ you will observe that you have stuck in loop.

9. Now Press *“Fire”,* the arrow next to the RA3 in the Stimulus pin.

10. Now press “Step Into” again, observe how the value of BottleNumber changes to ZER

University of Jordan
Faculty of Engineering and Technology
Department of Computer Engineering
Embedded Systems Laboratory 0907334



4 Experiment 4: LCD

Objectives

- To become familiar with HD44780 controller based LCDs and how to use them
- Knowing the various modes of operation of the LCD (8-bit/4-bit interface, 2-lines/1-line, CG-ROM).
- Distinguishing between the commands for the instruction register and data register.
- Stressing software and hardware co-design techniques by using the *Proteus* IDE package to simulate the LCD.

Introduction

What is an LCD?

A ***L****iquid* ***C****rystal* ***D****isplays* (LCD) is a thin, flat display device made up of any number of color or monochrome pixels arrayed in front of a light source or reflector. It is often utilized in battery-powered electronic devices because it uses very small amounts of electric power.

LCDs have the ability to display numbers, letters, words and a variety of symbols. This experiment teaches you about LCDs which are based upon the Hitachi HD44780 controller chipset. LCDs come in different shapes and sizes with 8, 16, 20, 24, 32, and 40 characters as standard in 1, 2 and 4–line versions. **However, all LCD's regardless of their external shape are internally built as a 40x2 format. See Figure 2 below**

Figure 1: A typical LCD module

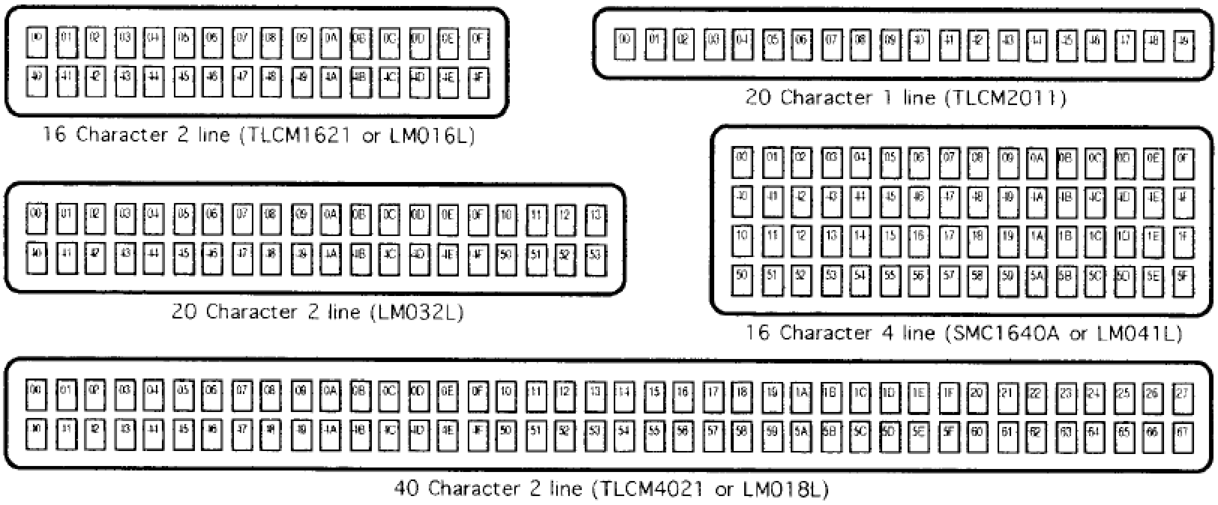


Figure 2: Different LCD modules shapes and sizes

Display position	1	2	3	4	5		39	40
DDRAM address (hexadecimal)	00	01	02	03	04	....	26	27
	40	41	42	43	44	....	66	67

Figure 3: Display address assignments for HD44780 controller based LCDs

LCD I/O

Most LCD modules conform to a standard interface specification. A 14-pin access is provided having eight data lines, three control lines and three power lines as shown below. Some LCD modules have 16 pins where the two additional pins are typically used for backlight purposes

Note: This image might differ from the actual LCD module, the order can be from left to right or vice versa therefore you should pay attention, pin 1 is marked to avoid confusion (printed on one of the pins).

Powering up the LCD requires connecting three lines: one for the positive power ***Vdd*** (usually +5V), one for negative power (or ground) ***Vss***. The ***Vee*** pin is usually connected to a potentiometer which is used to vary the contrast of the LCD display. We will connect this pin to the GND.

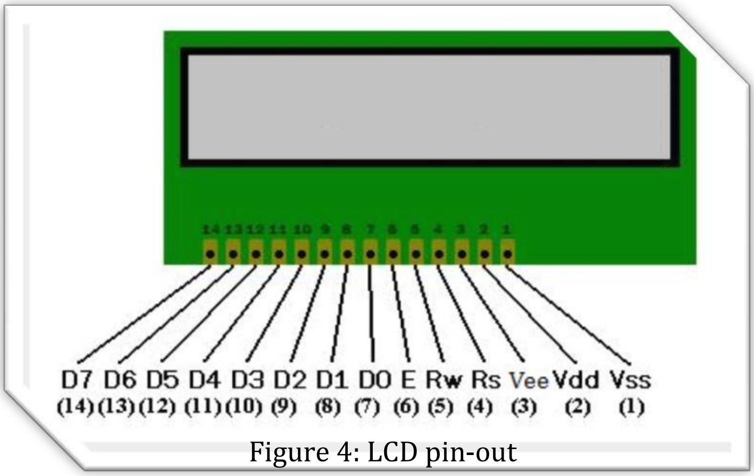


Figure 4: LCD pin-out

As you can see from the figure, the LCD connects to the microcontroller through three control lines: RS, RW and E, and through eight data lines D0-D7.

With 16-pin LCDs, you can use the L+ and L- pins to turn the backlight (BL) on/off.

PIN NO	NAME	FUNCTION
L+	Anode	Background Light
L-	Cathode	Background Light
1	Vcc	Ground
2	Vdd	+ve Supply
3	Vee	Contrast
4	RS	Register Select
5	R/W	Read/Write
6	E	Enable
7	D0	Data Bit 0
8	D1	Data Bit 1
9	D2	Data Bit 2
10	D3	Data Bit 3
11	D4	Data Bit 4
12	D5	Data Bit 5
13	D6	Data Bit 6
14	D7	Data Bit 7

Figure 5: LCD pin-out details

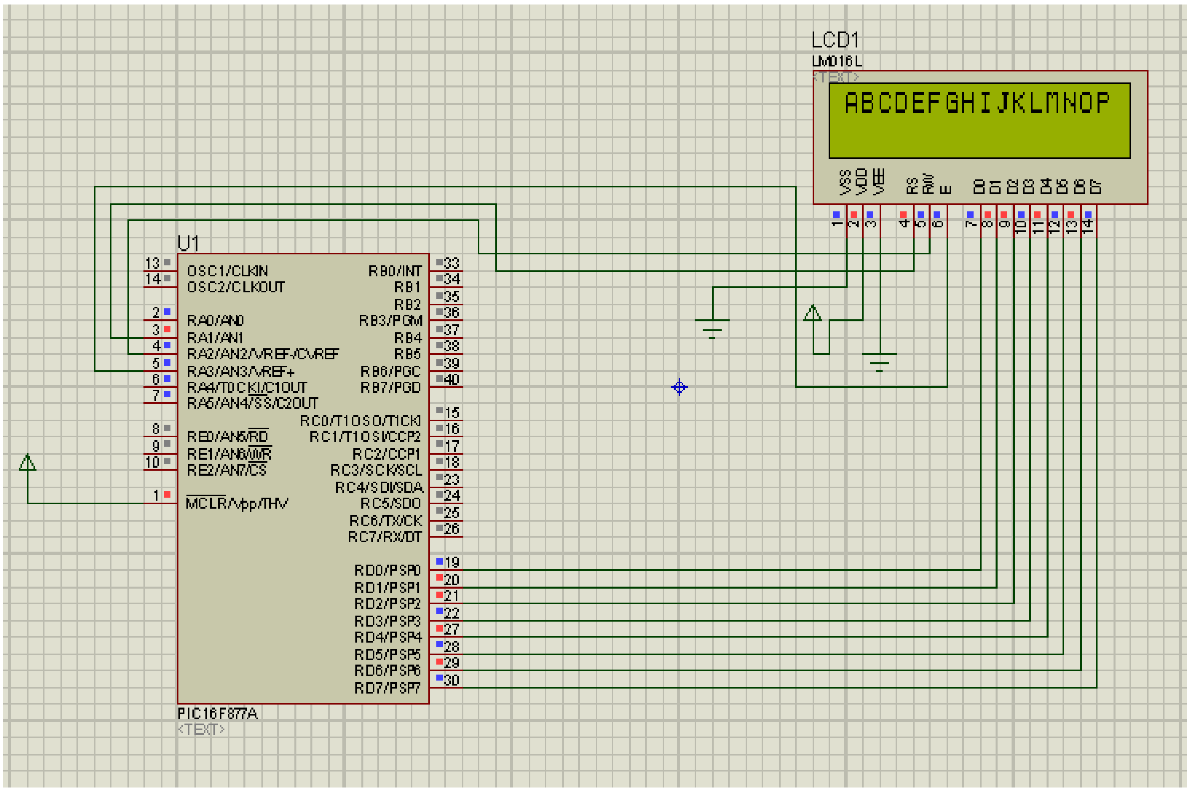


Figure 6: A typical interfacing between a PIC16F877A and an LCD module

When powered up, the LCD display should show a series of dark squares. These cells are actually in their off state. When power is applied, the LCD is reset; therefore we should issue a command to set it on. Moreover, you should issue some commands which configure the LCD. (See the table which lists all possible configurations below in the code and the explanation to each field)

Sending Commands/Data to the LCD

Using an LCD is a simple procedure once you learn it. Simply put you will place a value on the LCD lines D0-D7 (this value might be an ASCII value (character to be displayed), or another hexadecimal value corresponding to a certain command). So how will the LCD differentiate if this value on D0-D7 is corresponding to data or command?

Observe the figure below, as you might see the only difference is in the RS signal (**R**egister **S**elect), this is the only way for the LCD controller to know whether it is dealing with a character or a command!

Command				Binary							
	RS	R/W	E	D7	D6	D5	D4	D3	D2	D1	D0
Write Data to CG or DD RAM	1	0	↓	ASCII Value							
Write Command	0	0	↓	Refer to the Command Table below							

Figure 7: Necessary control signals for Data/Commands

Setting the necessary control signals in software:

F~~or this experiment assume t~~hat **RS (Register Select)** is connected to PORTA1 , **R/W (Read/Write)** to PORTA2 (In this lab experiment we are only writing to the LCD, reading from the LCD is left to the student as home study)and **E(Enable)** is connected to PORTA3. Moreover, assume that the LCD lines D0-D7 are directly connected to PORTD.

we will introduce two subroutines; one will set the necessary control signals for sending a character (send_char), the other for sending a command (send_cmd).

	send_char		send_cmd
1	movwf PORTD	1	movwf PORTD
2	bsf PORTA,1	2	bcf PORTA, 1
3	bsf PORTA, 3	3	bsf PORTA, 3
3	nop	3	nop
3	bcf PORTA, 3	3	bcf PORTA, 3
4	bcf PORTA, 2	4	bcf PORTA,2
	call delay		call delay
	return		return
Steps to send character to LCD 1.Place the ASCII character on the D0-D7 lines 2. Register Select (RS) = 1 to send characters 3. "Enable" Pulse (Set High – Delay – Set Low) 4. Delay to give LCD the time needed to display the character		Steps to send a command to LCD 1.Place the command on the D0-D7 lines 2. Register Select (RS) = 0 to send commands 3. "Enable" Pulse (Set High – Delay – Set Low) 4. Delay to give LCD the time needed to carry out the command	

Table 1: Sending Characters/Commands Steps

Displaying Characters

All English letters and numbers (as well as special characters, Japanese and Greek letters) are built in the LCD module in such a way that it **conforms to the ASCII standard**. In order to display a character, you only need to send its ASCII code to the LCD which it uses to display the character.

To display a character on the LCD simply move the ASCII character to the working register (for this experiment) then call send_char subroutine.

Notice that from column 1 to D, the character resolution is 5 pixels wide x 7 pixels high (5x7) (column 0 is a special case, it is 5x8, but considered as 5x7, more on this later) whereas the character resolution of columns E and F is 5 pixels wide x 10 pixels high (5x10). We should change the resolution if we are to use characters from different resolution columns, this can be done using a command discussed later.

Figure 8: LCD Characters Map

Command	Binary D7	D6	D5	D4	D3	D2	D1	D0	Hex
Clear Display	0	0	0	0	0	0	0	1	01
Display & Cursor Home	0	0	0	0	0	0	1	x	02 or 03
Character Entry Mode	0	0	0	0	0	1	1/D	S	04 to 07
Display On/Off & Cursor	0	0	0	0	1	D	U	B	08 to 0F
Display/Cursor Shift	0	0	0	1	D/C	R/L	x	x	10 to 1F
Function Set	0	0	1	8/4	2/1	10/7	x	x	20 to 3F
Set CGRAM Address	0	1	A	A	A	A	A	A	40 to 7F
Set Display Address	1	A	A	A	A	A	A	A	80 to FF

1/D: 1=Increment*, 0=Decrement
S: 1=Display shift on, 0=Off*
D: 1=Display on, 0=Off*
U: 1=Cursor underline on, 0=Off*
B: 1=Cursor blink on, 0=Off*
D/C: 1=Display shift, 0=Cursor move
R/L: 1=Right shift, 0=Left shift
8/4: 1=8-bit interface*, 0=4-bit interface
2/1: 1=2 line mode, 0=1 line mode*
10/7: 1=5x10 dot format, 0=5x7 dot format*
x = Don't care
* = Initialization settings

Figure 9: LCD command control codes

To issue any of these commands to the LCD, all you have to do is place the command value in the working register, then issue the instruction "Call Send_cmd"

```
;*****************************************************************************************
;                    Explaining the commands and their parameters in the LCD command table
;*****************************************************************************************
```

Clear Display

Moving the value 01 to the working register followed by "call send_cmd" will clear the LCD display, however the cursor will remain at it last position, so any future character writes will start from the last location, to reset the cursor position use the Display and Cursor Home command.

Display and Cursor Home

Resets cursor location to position 00 of the LCD screen (Figure 3), future writes will start at the first location of the first line.

Character Entry Mode

This command has two parameters 1/D and S:
1/D: By default, the cursor is automatically set to move from location 00 to 01 and so on (Increment mode). Suppose now that you are to write from right to left (as in the Arabic language), then you have to set the cursor to the Decrement mode.
S: Accompanies the D/C parameter, explained below

Display On/OFF and Cursor

This command has three parameters:
D: Turns on the display (when you see the black dots on the LCD, it means that it is POWERED on, but not yet ready to operate), this command activates the LCD in order to be ready to use.
U: This displays the cursor (in the form of a horizontal line at the bottom of the character) when it is high and turns the cursor off when it is low
B: If the underline cursor option is enabled, this will blink the cursor if high.

Display/Cursor Shift

All LCDs based on the HD44780 format - whatever their actual physical size is - are internally built in to be 40 characters x 2 lines with the upper row having the display addresses 0-27_H (27_H = 39D → 0-39 =

40 Characters!!) and the lower row from 40_H -67_H. Now suppose you bought an LCD with the physical size of 20 char. x 2 lines, when you start writing to the LCD and the cursor reaches locations 20_D, 21_D, and 22_D …, you will not see them BUT don't worry, they are not lost. They were written in their respective locations but you could not see them because your bought LCD is 20 **visible** Characters wide from the outside and 40 from the inside. All you have to do is shift the display. So all you do is

1. Determine the direction of the shift (R/L)
2. Issue the shift Command D/C

R/L: Determines the direction of the shift, this might be useful if you are writing Arabic characters …
D/C: if this bit has a value of 0, the display is not shifted and the cursor moves the same way it was, if the its value is logic high, the display is shifted once, you might need to issue this command multiple times in order to shift the display by multiple locations!

Function Set

This command has three parameters:
8/4: Eight/Four bits mode
8 – Bit interface: you send the whole command/character (8 bits) in one stage to the D0-D7 lines
4 – Bit interface: you send the command/character in two stages as nibbles to D4-D7 lines.
When to use the 4-bit mode?

1. Interfacing LCD with older devices which have 4-bit wide I/O Bus
2. You don't have enough I/O pins remaining, or you want to conserve the I/O pins for other HW

2/1: Line mode, determines whether you want to use the upper line of the LCD or both lines

10/7: Dot format, based on the LCD built-in characters table, note the following:

* 5x7 format (Default) is used whenever you use the characters found in columns 1 to D
* 5x7 format is also used whenever you use the built in characters in CG-RAM ***(EVEN THOUGH THE CG-RAM CHARACTERs ARE 5X8!!!)***
* 5x10 format is only used when displaying the characters found in columns **E** and **F**

*** *In LCD initialization, we normally set "**Clear Display**", "**Display and Cursor Home**", "**Display On/OFF**" **and "Cursor, and Function Set"**, we place the value of the command then use the call send_cmd instruction.*
****

Set Display Address command

Syntax: 1AAAAAAA
This command allows you to move the cursor to whichever location you want, suppose you want to start writing in the middle of the display (assuming the ***visible*** width of the LCD screen is 20), then from Figure 2 you will observe that location 06 is approximately in the middle so you replace the A's with 06:
1*AAAAAAA* →1*0000110*→0x86
Moreover, suppose you wish to move to the second line which starts at location 40, same as above
1*AAAAAAA* → 1*1000000* → 0xC0
After calculating this value, you place it in the working register and then use the call send_cmd instruction.

```
;*****************************************************************************
;                                   EXAMPLE CODE 1
;*****************************************************************************
;  This code displays on the first "upper" row of the LCD the 26 English letters in alphabetical order
;  The code starts with LCD initialization commands such as clearing the LCD, setting modes and
;  display shifting.
;
; Outputs:
;         LCD Control:
;                               RA1: RS (Register Select)
;                               RA3: E   (LCD Enable)
;         LCD Data:
;                               PORTD 0-7 to LCD DATA 0-7 for sending commands/characters
; Notes:
;         The RW pin (Read/Write) - of the LCD - is connected to RA2
;         The BL pin (Back Light) – of the LCD – is connected to potentiometer
;*****************************************************************************
          include         "p16f877A.inc"
;*****************************************************************************
          cblock          0x20
                                tempChar        ;holds the character to be displayed
                                charCount       ;holds the number of the English alphabet
                                lsd             ;lsd and msd are used in delay loop calculation
                                msd
          endc
;*****************************************************************************
; Start of executable code
                org     0x000
                goto    Initial
;*****************************************************************************
; Interrupt vector
INT_SVC         org     0x0004
                goto    INT_SVC
;*****************************************************************************
; Initial Routine
; INPUT:        NONE
; OUTPUT:       NONE
; RESULT:       Configure I/O ports (PORTD and PORTA as output, PORTA as digital)
;               Configure LCD to work in 8-bit mode, with two lines of display and 5x7 dot format.
;               Set the cursor to the home location (location 00), set the cursor to the visible state
;               with no blinking
;*****************************************************************************
Initial
                Banksel TRISA           ;PORTD and PORTA as outputs
                Clrf    TRISA
                Clrf    TRISD
                Banksel ADCON1          ;PORTA as digital output
                Movlw   07
                mowf    ADCON1
                Banksel PORTA
                Clrf    PORTA
                Clrf    PORTD
                movlw   d'26'
                Movwf   charCount       ; initialize charCount with 26 Number of Characters in the English language
```

```
                Movlw   0x38            ;8-bit mode, 2-line display, 5x7 dot format
                Call    send_cmd
                Movlw   0x0e            ;Display on, Cursor Underline on, Blink off
                Call    send_cmd
                Movlw   0x02            ;Display and cursor home
                Call    send_cmd
                Movlw   0x01            ;clear display
                Call    send_cmd
;*****************************************************************************
; Main Routine
;*****************************************************************************
Main
                Movlw   'A'
                Movwf   tempChar
CharacterDisplay                                ; Generate and display all 26 English Letters
                Call    send_char
                Movf    tempChar ,w             ; 'A' has the ASCII code of 65 decimal (0x41), by
                Addlw   1                       ; adding 1 to it we have 66, which is B. Therefore, by
                movwf   tempChar                ; continuously adding 1 to tempChar we are cycling
                movf    tempChar ,w             ; through the ASCII table (here: alphabets)
                decfsz  charCount
                goto    CharacterDisplay
Mainloop
                Movlw   0x1c                    ;This command shifts the display to the right once
                Call    send_cmd
                Call    delay
                Goto    Mainloop                ; This loop makes the character rotate continuously
;*****************************************************************************
send_cmd
                movwf   PORTD           ; Refer to table 1 on Page 5 for review of this subroutine
                bcf     PORTA, 1
                bsf     PORTA, 3
                nop
                bcf     PORTA, 3
                bcf     PORTA, 2
                call    delay
                return
;*****************************************************************************
send_char
                movwf   PORTD           ; Refer to table 1 on Page 5 for review of this subroutine
                bsf     PORTA, 1
                bsf     PORTA, 3
                nop
                bcf     PORTA, 3
                bcf     PORTA, 2
                call    delay
                return
;*****************************************************************************
delay
                movlw   0x80
                movwf   msd
                clrf    lsd
loop2
                decfsz  lsd,f
```

```
            goto    loop2
            decfsz  msd,f
endLcd
            goto    loop2
            return
;******************************************************************************
            End

```

Set CG-RAM Address command
Syntax: 01AAAAAA

If you give a closer look at Figure 8, you will clearly see that the table only contains English and Japanese characters, numbers, symbols as well as special characters! Suppose now that you would like to display a character not found in the built-in table of the LCD (i.e. an Arabic Character). In this case we will have to use what is called the CG-RAM (Character Generation RAM), which is a reserved memory space in which you could draw your own characters and later display them.

Observe column one in Figure 8, the locations inside this column are reserved for the CG-RAM. Even though you see 16 locations (0 to F), you only have the possibility to use the first 8 locations 0 to 7 because locations 8 to F are mirrors of locations 0 – 7.

So, to organize things, in order to use our own characters we have to do the following:

1. Draw and store our own defined characters in CG-RAM
2. Display the characters on the LCD screen as if it were any of the other characters in the table

Drawing and storing our own defined characters in CG-RAM

As stated earlier, we have eight locations to store our characters in. So how do we choose which location out of these to start drawing and building our characters in?

The answer is quite simple; follow this rule as stated in the datasheet of the HD44780 controller

1. To write (build/store a character in location 00 (crossing of the row and column)), you send the CG-RAM address command as follows: 01*AAAAAA* → 01*000000* → 0x40
2. However, to write in any location from 01 to 07, you have to skip eight locations (WHY?) So the CG-RAM address command will send **0x48** (to store a character in location 1**),** **0x50** (to store a character in location 2) and so on...

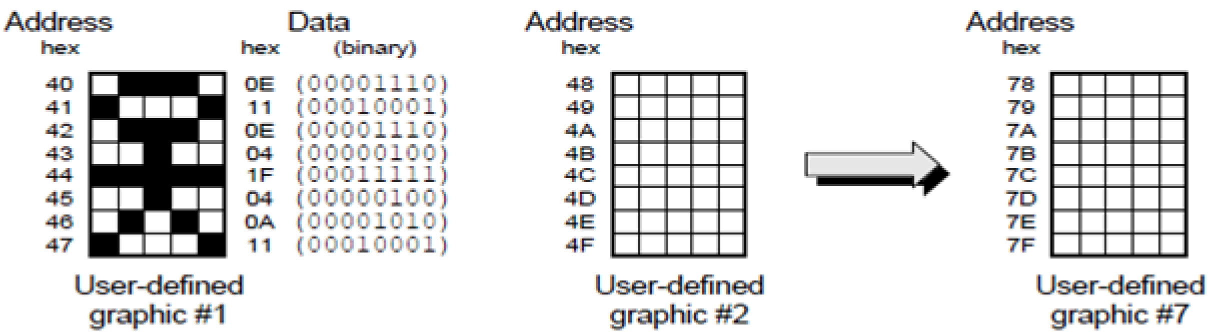
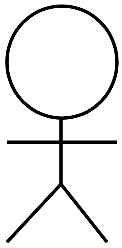


Figure10 Showing how the CGRAM addresses correspond to individual pixels.

So up to this point we have defined **where** to write our characters but not how to build them! This is the fun part☺, draw a 5x8 Grid and start drawing your character inside, then replace each shaded cell with one and not shaded ones with zero. Append three zeros to the left (B5-B7) and finally transform the sequence into hexadecimal format. This is the sequence which you will fill in the CG-RAM SEQUENTIALLY once you have set the CG-RAM Address before.



B4	B3	B2	B1	B0		B7	B6	B5	B4	B3	B2	B1	B0		
						0	0	0	0	1	1	1	0		0x0E
						0	0	0	1	0	0	0	1		0x11
						0	0	0	0	1	1	1	0		0x0E
					→	0	0	0	0	0	1	0	0	→	0x04
						0	0	0	1	1	1	1	1		0x1F
						0	0	0	0	0	1	0	0		0x04
						0	0	0	0	1	0	1	0		0x0A
						0	0	0	1	0	0	0	1		0x11

```
                                        Example
DrawStick1                               Setting the CGRAM address at which we draw the stick man
            Movlw  0x40                 ;Here it is address 0x00 in Figure 8 which transforms into
            Call   send_cmd             ; command 0x40
            Movlw  0X0E                  Sending data that implements the Stick man
            Call   send_char            ; Notice the address where to store the character in CG-RAM
            Movlw  0X11                 ;is a command thus use send_cmd, whereas the
            Call   send_char            ;data bits of the stickman are sent as Data
            Movlw  0X0E                 ;using send_char
            Call   send_char
            Movlw  0X04
            Call   send_char
            Movlw  0X1F
            Call   send_char
            Movlw  0X04
            Call   send_char
            Movlw  0X0A
            Call   send_char
            Movlw  0X11
            Call   send_char
            Return
```

Displaying the user generated (drawn) characters on the LCD screen

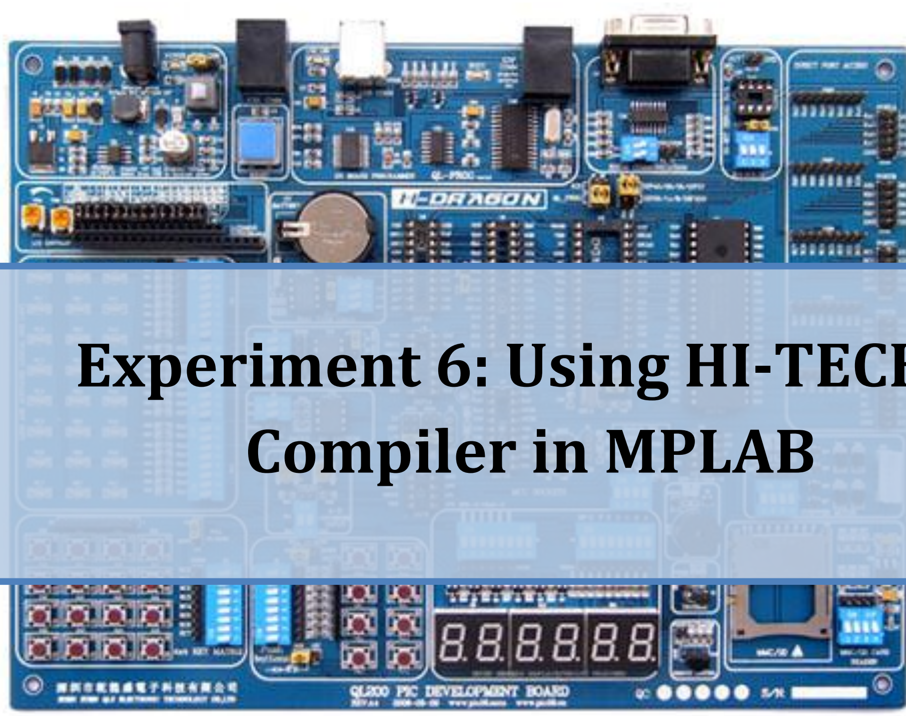
Simply,if we stored our character in location 0, we move 0 to the working register then issue the **"call send_char"** command, if we stored it in location 2, move 2 to the working register and so on ….

```
;*****************************************************************************
;                                             EXAMPLE CODE 2
;*****************************************************************************
;  This code stores two shapes of a stickman, one in location 00 (of Figure 8), and another at location
; 01. The first stickman is written on the leftmost location of the upper line, the second stick man
; shape is also written above the first one, then the first stick man is rewritten on the same location
; that is display: first stickman shape → second stickman shape → first stickman shape and so on ..
; thus the stickman will appear as if it is moving ! ☺
;
; Outputs:
;         LCD Control:
;                                   RA1: RS (Register Select)
;                                   RA3: E   (LCD Enable)
;         LCD Data:
;                                   PORTD 0-7 to LCD DATA 0-7 for sending commands/characters
; Notes:
;         The RW pin (Read/Write) - of the LCD - is connected to RA2
;         The BL pin (Back Light) – of the LCD – is  connected potentiometer
;*****************************************************************************
        include         "p16f877A.inc"
;*****************************************************************************
        cblock          0x20
                                lsd             ;lsd and msd are used in delay loop calculation
                                msd
        endc
;*****************************************************************************
; Start of executable code
                org     0x000
                goto    Initial
;*****************************************************************************
; Interrupt vector
INT_SVC         org     0x0004
                goto    INT_SVC
;*****************************************************************************
; Initial Routine
; INPUT:        NONE
; OUTPUT:       NONE
; RESULT:       Configure I/O ports (PORTD and PORTA as output, PORTA as digital)
;               Configure LCD to work in 8-bit mode, with two lines of display and 5x7 dot format.
;               Set the cursor to the home location (location 00), set the cursor to the visible state
;               with no blinking
;*****************************************************************************
Initial
                Banksel TRISA           ;PORTA and PORTD as outputs
                Clrf    TRISA
                Clrf    TRISD
                Banksel ADCON1          ;PORTA as digital output
                movlw 07
                mowf ADCON1
                Banksel PORTA
                Clrf    PORTA
                Clrf    PORTD
                Movlw   0x38            ;8-bit mode, 2-line display, 5x7 dot format
                Call    send_cmd
```

```
                Movlw  0x0e           ;Display on, Cursor Underline on, Blink off
                Call   send_cmd
                Movlw  0x02           ;Display and cursor home
                Call   send_cmd
                Movlw  0x01           ;clear display
                Call   send_cmd
                Call   DrawStick1     ;The subroutines draw and store the Stick man inside the
                Call   DrawStick2     ;CG-RAM. This DOES NOT mean that the character is
                                      ;displayed on the LCD, it was only stored inside the CG-RAM
                                      ;of the LCD.
                Movlw  0x01           ;the datasheet says you have to clear display command after
                Call   send_cmd       ;storing the characters or the code will not work

;***************************************************************************
; Main Routine
;***************************************************************************
Main
                Movlw  0x00           ;Display character stored in location 00 (Figure 8), which in
                Call   send_char      ;this case is our first stickman in CG-RAM
                Movlw  0x02           ;Cursor Home Command
                Call   send_cmd
                Movlw  0x01           ;Display character stored in location 00 (Figure 8), which in
                Call   send_char      ;this case is our first stickman in CG-RAM
                Movlw  0x02           ;Cursor Home Command
                Call   send_cmd
                Goto   Main           ; This loop makes the character rotate continuously
;***************************************************************************
send_cmd
                movwf  PORTD          ; Refer to table 1 on Page 5 for review of this subroutine
                bcf    PORTA, 1
                bsf    PORTA, 3
                nop
                bcf    PORTA, 3
                bcf    PORTA, 2
                call   delay
                return
;***************************************************************************
send_char
                movwf  PORTD          ; Refer to table 1 on Page 5 for review of this subroutine
                bsf    PORTA, 1
                bsf    PORTA, 3
                nop
                bcf    PORTA, 3
                bcf    PORTA, 2
                call   delay
                return
;***************************************************************************
delay
                movlw  0x80
                movwf  msd
                clrf   lsd
loop2
                decfsz lsd,f
                goto   loop2
```

```
                decfsz  msd,f
endLcd
                goto    loop2
                return
;************************************************************************
DrawStick1                              Setting the CGRAM address at which we draw the stick man
                Movlw   0x40            ; Here it is address 0x00 in Figure 8 which transforms
                Call    send_cmd        ; into command 0x40
                Movlw   0X0E            ;Sending data that implements the Stick man
                Call    send_char
                Movlw   0X11
                Call    send_char
                Movlw   0X0E
                Call    send_char
                Movlw   0X04
                Call    send_char
                Movlw   0X1F
                Call    send_char
                Movlw   0X04
                Call    send_char
                Movlw   0X0A
                Call    send_char
                Movlw   0X11
                Call    send_char
                Return
;************************************************************************
DrawStick2                              ;Setting the CGRAM address at which we draw the stick man
                Movlw   0x48            ;Here it is address 0x01 in Figure 8 which transforms
                Call    send_cmd        ; into command 0x48
                Movlw   0X0E            ;Sending data that implements the Stick man
                Call    send_char
                Movlw   0X0A
                Call    send_char
                Movlw   0X04
                Call    send_char
                Movlw   0X15
                Call    send_char
                Movlw   0X0E
                Call    send_char
                Movlw   0X04
                Call    send_char
                Movlw   0X0A
                Call    send_char
                Movlw   0X0A
                Call    send_char
                Return
;************************************************************************
                End
```

University of Jordan
Faculty of Engineering and Technology
Department of Computer Engineering
Embedded Systems Laboratory 0907334



6

Experiment 6: Using HI-TECH C Compiler in MPLAB

Objectives

The main objectives of this experiment are to familiarize you with:

- Writing PIC programs in C
- Setting up MPLAB IDE projects to use the HI-TECH C compiler
- Becoming familiar with HI-TECH C primitives, built-in function in use with 10/12/16 MCU Family

Prepared by Eng. Enas Ja’ra

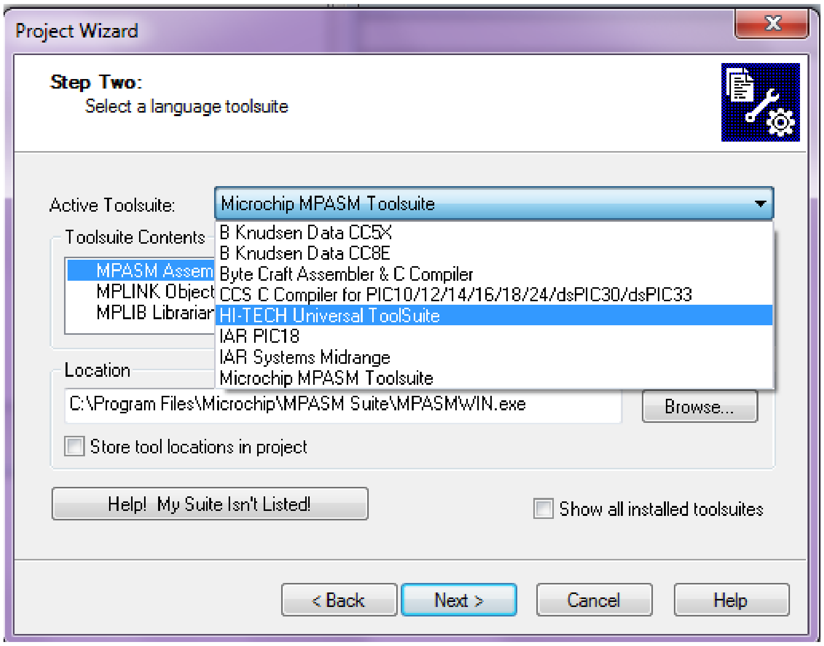
INTRODUCTION

So far in this lab course, PIC assembly programming has been introduced, however, in practice, most of the industrial and control codes are written in High Level Languages (abbreviated as HLL) the most common of which is the C programming language. The use of high level languages is preferred due to their simplicity which allows for faster program development (especially for large and very complex programs), easier debugging, and for easier future code maintainability, this will provide developers with shorter time to market advantages in a world where competition is at its prime to introduce new commercial products. On the other hand, HLLs assembled codes are often longer (due to inefficient compilers, aggressive and advanced optimizing compilers are often used to yield better results). Longer codes are at a disadvantage since memory space is limited in microcontrollers not to mention that longer codes take more time to execute. Expert assembly programmers can rewrite certain pieces of code in a very optimized and short fashion such that they execute faster, this is very important especially when real time applications are concerned. This direct use of assembly language requires that the programmer knows the problem in hand very well and that one is experienced in both software and target microcontroller hardwrae limitations. Often, programmers combine in between the use of C and Assembly language in the same developed source code.

There are many C compilers available commercially, such as mikroC, CCS and HI-TECH among others. This experiment introduces the "free" Lite version C compiler from HI-TECH software bundled with MPLAB, in contrast to the Pro versions of compilers commercially available from HI-TECH and others, the compiler and assembler don't use aggressive techniques and the resultant assembly codes are larger in size.

THIS PART ASSUMES YOU HAVE ALREADY SAVED A FILE WITH A C EXTENSION AND YOU HAVE ALREADY INSTALLED THE HI-TECH C PRO FOR THE PIC10/12/16 MCU FAMILY COMPILER

Create a project in MPLAB in the same steps as was shown in Exp 0, the only difference is in the step of selecting a language toolsuite; " Active Toolsuite" dialog box:

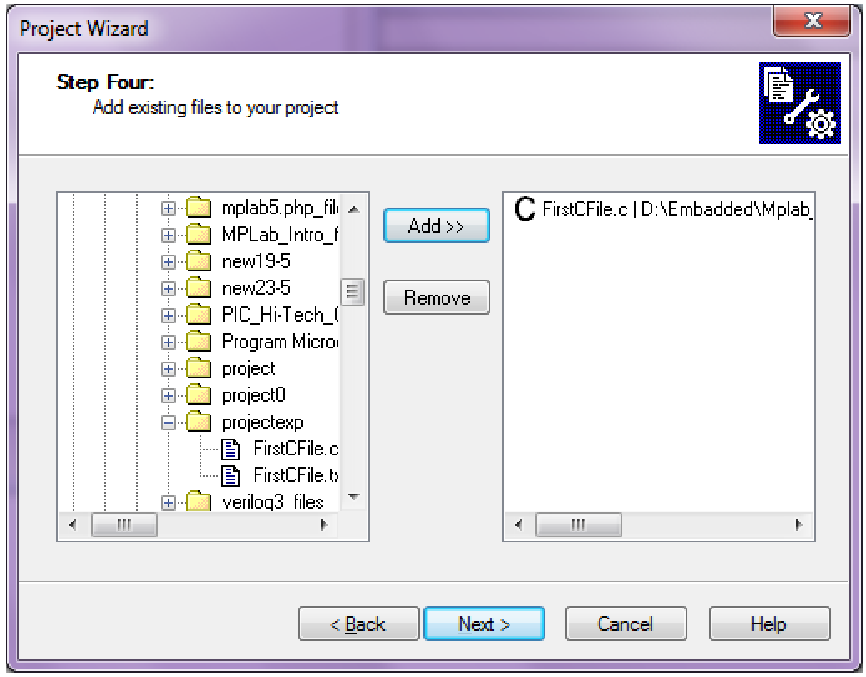


In this step where you get to specify the toolsuite associated with the project, you are not associating the project with the MPASM compiler as previously done, but instead we will be using the HI-TECH C compilers for Microchip devices

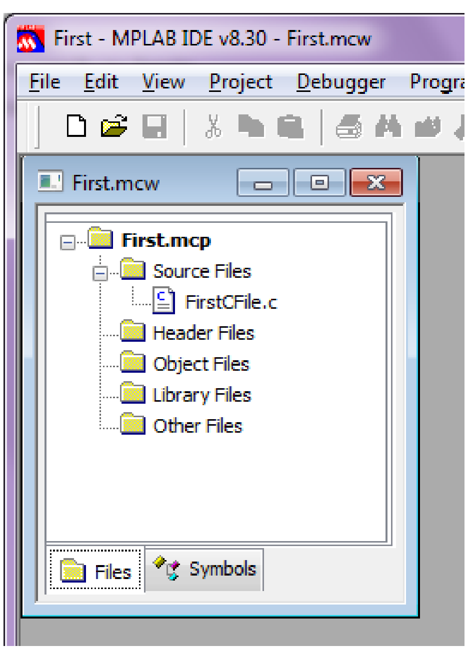
In the Active Toolsuite drop down menu, select HI-TECH Universal Toolsuite → Click next.

The next steps will proceed as usual:

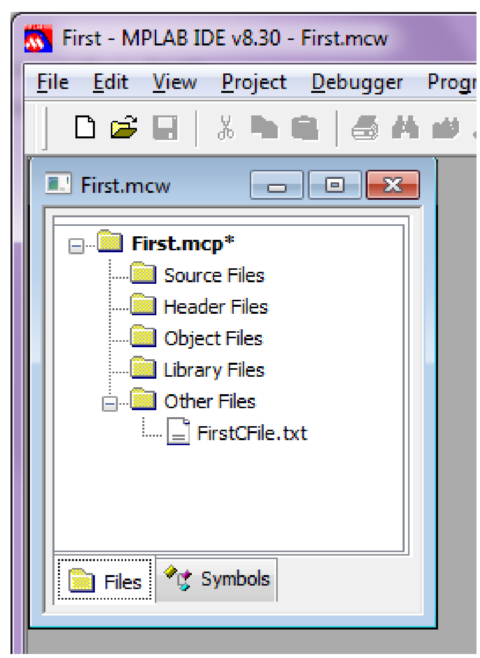
Browse to the directory where you saved your C file. Give your project a name → Save → Next. If you navigated correctly to your file destination you should see it in the left pane otherwise choose back and browse to the correct path. When done Click add your file to the project (here: FirstCFile.c). Make sure that the letter C is beside your file and not any other letter → Click next →Click Finish.



As before, you should see your C file under *Source file* list, now you are ready to begin.
Double click on the FirstCFile.C file in the project file tree to open. This is where you will write your programs, debug and simulate them.



CORRECT



WRONG

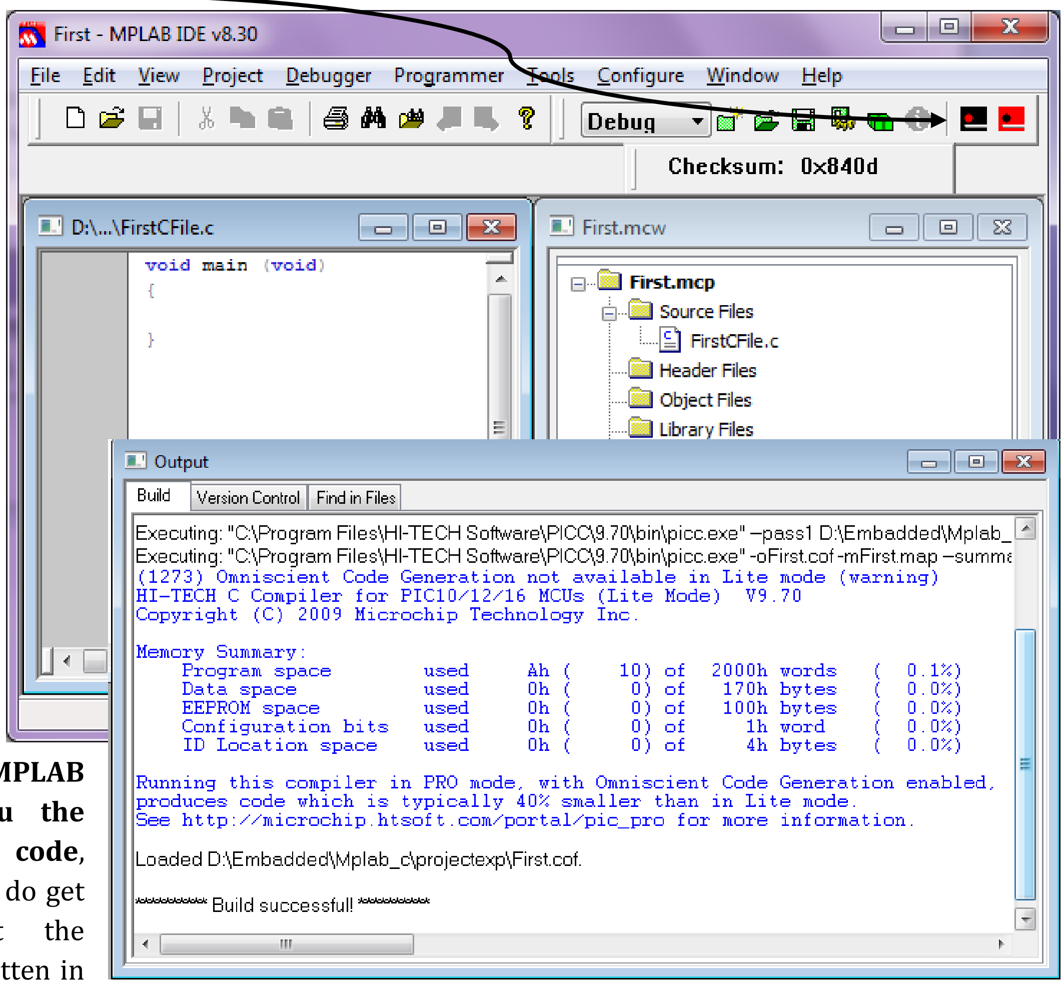
The proceeding parts assume that you have basic knowledge of programming in C. We will present the C language in general context then we'll introduce it within the contest of use in PIC programming. The following discussion attempts to write and simulate a simple C program in MPLAB and check the results

In MPLAB, inside your newly created project from above, write the following:

```
#include <htc.h>
void main(void)    // every C program you write needs a function called main.
{

}
```

Notice that comments are indicated with // instead of ';'

After writing the above EMPTY program we should build the code to ensure that MPLAB IDE and HI-TECH C are properly installed. Select Build from the Project menu, or choose any of MPLAB IDE's shortcuts to build the project — you can, for instance, click on the toolbar button that shows the HI-TECH "ball and stick" logo, as shown in the figure below. You will notice that the Project menu has two items: Build and Rebuild.



An output window should show with BUILD SUCCEDDED

The compiler has produced memory summary and there is no message indicating that the build failed, so we have successfully compiled the project. If there are errors they will be printed in Build tab of this window. **You can double-click each error message and MPLAB IDE will show you the offending line of code**, where possible. If you do get errors, check that the program is as it is written in this document. BUILD SUCCEED DOES NOT MEAN THAT YOUR PROGRAM IS CORRECT, IT SIMPLY MEANS THAT THERE ARE NO **SYNTAX** ERRORS FOUND, SO WATCH OUT FOR ANY LOGICAL ERRORS YOU MIGHT MAKE.

Quick Review of the Basic Rules for Programming in C

1. Comments for only ONE line of code start with 2 slashes: //
   // This is a one line comment
   Remember to always document your code through the use of functional comments!
2. Comments for more than one line start with /* en end with */
   /*
   This is a comment.
   This is another comment.
   */
3. At the end of each line with some instruction, a semi-colon (;) has to be placed.
   a=a+3;
4. Parts of the program that belong together (functions, statements, etc.), are between { and }.
   void main(void) //Function
   {
   //Add code
   }

The Basic Structure of a C Program

The ***ordered*** structure of a program in C is as follows:

- Libraries
- Global Variables
- Function Prototypes
- Main Function
- Functions

❖ Adding libraries (the initial few lines of any C program)

Syntax: #include <filename.h>

Libraries such as "htc.h", "math.h" and "stdlib.h" include many references to built-in variables and functions to be used in programs, if the header files are not included, the built-in functions and variables if used will not be defined which will result in build errors.

The htc.h file will be included in all our C programs which use the HI-TECH compiler, other compilers have different header files, refer to their documentation when needed.

❖ Declaring "global" variables.

Define and declare the variables to be used throughout the program, this is in contrast to "local" variables discussed later on.

❖ Defining prototypes of the functions.

A C program has a main function and possibly other functions as well which might be written below the main function. If we are to call any of the other functions from inside the main subroutine, the build will fail and indicate that the function is undefined. This is because the code is compiled line by line and at the moment the compiler attempts to compile "call function", it still has not known of the existence of this function because it is declared later in the code "after main". One solution is to *place all the functions before the main function*. Another preferred method is the use of function *prototype*. A prototype of a function ensures that the function can be called anywhere in the program**. It is simply copying only the header of the function, placing it before the main subroutine and ending it with a semicolon ';'**

- Main function.

  This is the function that will be called first when starting your microcontroller. From there, other functions are called. ***Every C program must have a main function.***
- Functions.

  Functions are a grouping of instructions which perform a certain task. They are the unit of modularity and are very useful to make it easy to repeat tasks. They have input and output variables.

Syntax: type identifier function name (type identifier identifier1, type identifier identifier2 ….)

```
{
        //The body of the function
        return identifier        //only when return type is not void
}
```

Type identifier: could be int, long, short, char, void ….. etc

The output variable type precedes the function's name, input variables follow the function name and are placed in between brackets, a function can take as many input variables as needed but it only returns one output variable.

testFunction1 has two input parametres of type integer (x,y) but has no output, all processing is local inside the function and it returns no values	testFunction2 has one input parameter of type integer (x), it returns an output which is the square of the input number. Notice, that value returning functions end with a return statement, omitting of which will result in an error	testFunction3 takes no input or outputs.

void testFunction1(int x, int y)
{
int k;
k = x;
y = 2 + x;
}
	int testFunction2 (int x)
	{
	return x*x;
	}
		void testFunction3 (void)
		{
		//some code
		}

How to call function: Examples		
testFunction1(75,99)	A = testFunction2(5) Since this type of functions returns a value, the value need be stored in a previously defined variable. The variable must be defined as the same return output type of the function, if the function returns an integer, A must be defined as integer, if the function returns a character, A need be defined as character …	testFunction() Note that the brackets are left empty when no arguments are passed

Example Program 1: Typical Program Layout

```
// ExampleProgram1.c
        #include <htc.h> //Always include this library when using HI-TECH C compiler
//Declaring global variables
        int     a, b, c;
        char    temp;
//Defining prototypes
        int     calc (int p);
//Main function
        void main(void)
        {
                a=calc(3); //write main body code
        }
//Functions
        int calc (int p)
        {
                p=p+1;    //write function body code
                return p;
        }
```

More on Variables

Variables can be classified into two main types depending on their scope:

Global Variables

These variables can be accessed (i.e. known) by any function comprising the program. They are implemented by associating memory locations with variable names. They do not get recreated if the function is recalled. *In Example Program 1, (a, b, c, and temp) are* ***GLOBAL VARIABLES***

Local Variables

These variables only exist inside the specific function that creates them. They are unknown to other functions and to the main program. As such, they are normally implemented using a stack. Local variables cease to exist once the function that created them is completed. They are recreated each time a function is executed or called. *In Example Program 1, (p) is a* ***LOCAL VARIABLE***

Variable Types

The following table lists all possible variable types in C, the size they take up in memory and the range of each.

Type	Memory usage	Possible values
bit	1 bit	0, 1
char	8 bits	-128…127
unsigned char	8 bits	0…255
signed char	8 bits	-128…127
int	16 bits	-32k7…32k7
unsigned int	16 bits	0…65k5
signed int	16 bits	-32k7…32k7
long int	32 bits	-2G1…2G1
unsigned long int	32 bits	0…4G3
signed long int	32 bits	-2G1…2G1
float	32 bits	± 10^(±38)
double	32 bits	± 10^(±38)

Default Input Is Decimal

Example Program 2:

```
#include <htc.h>
char           Ch;
unsigned int   X;
signed int     Y;
int            Z, a, b, c; // Same as "signed int"
unsigned char  Ch1;
bit            S, T;

void main (void)
{
        Ch = 'a';
        X  = -5;
        Y  = 0x25;
        Z  =-5;
        Ch1='b';
        T   = 0;
        S   = 81;       //S=1 When assigning a larger integral type to a bit variable,
                        //only the Least Significant bit is used.
        a   = 15;
        b  = 0b00001111;
        c   = 0x0F;
        // a, b, c will all have the same value which is 15

}
```

C Operators

❖ Relational and bit operators

>	Greater than
>=	Greater than or similar to
<	Less than
<=	Less than or similar to
==	Equal to
!=	Not equal to

~	Bitwise NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
<<	Shift to left
>>	Shift to right

❖ Arithmetic operators

x--;	This is the same as x = x – 1;
x++;	This is the same as x = x + 1;

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (remainder after division)

Operators Precedence Chart

Operator precedence describes the order in which C reads expressions. For example, the expression a=4+b*2 contains two operations, an addition and a multiplication. Does the HI TECH compiler evaluate 4+b first, then multiply the result by 2, or does it evaluate b*2 first, then add 4 to the result? The operator precedence chart contains the answers. Operators higher in the chart have a higher precedence, meaning that the HI TECH compiler evaluates them first. Operators on the same line in the chart have the same precedence, and the "Associativity" column on the right gives their evaluation order.

Operator Precedence Chart		
Operator Type	**Operator**	**Associativity**
Primary Expression Operators	**()**	left-to-right
Binary Operators	*** / %**	left-to-right
	+ -	
	>> <<	
	< > <= >=	
	== !=	
	&	
	^	
	|	

Example Program 3: Fibonacci series: 0, 1, 1, 2, 3, 5

```
#include <htc.h>                                            // Library
unsigned int Fib (unsigned int Num1, unsigned int Num2);   // Prototype
unsigned int    F1, F2, F3, F4, F5, F6;                     // Global Variables

void main (void)                                            // Main function
{
        F1 = 0;
        F2 = 1;
        F3 = Fib (F1, F2);
        F4 = Fib (F2, F3);
        F5 = Fib (F3, F4);
        F6 = Fib (F4, F5);
}
 unsigned int Fib (unsigned int Num1, unsigned int Num2) //Function
{
        return Num1 + Num2;
}
```

Preparing for Simulation

1. Start a new MPLAB session, add the file *ExampleProgram3.c* to your project
2. Build the project
3. Select **Debugger** ⇴ **Select Tool** ⇴**MPLAB SIM**
4. Go to View Menu → Watch (From the drop out menu choose the variables watch F1 through F6 we want to inspect during simulation and click ADD Symbol for each one)

From the **Debugger Menu** → choose **Select Tool** → then **MPLAB SIM**
After the following buttons appears in the toolbar:

5. Press the "Step into" button one at a time and check the Watch window each time an instruction executes; keep pressing "Step into" until you all the six terms of the series are generated.
6. Reset the simulation, do step 5 above but this time use "Step Over", note the difference
7. Reset the simulation, do step 5 above, this time place a break point at the last instruction in main, press run. Inspect the variables in watch window.

Notes about simulating a code written in C in MPLAB

Stepping into codes written in C is not as direct as one would imagine, different compilers translate the C code into assembly differently, a single line of code might be translated into multiple assembly lines, for example a simple assignment statement "X = 5" where X has been defined as integer will be translated into four assembly instructions.

```
Movlw 05
Movwf 0x70   //GPR address 0x70 chosen by compiler
Movlw 00
Movwf 0x71
```

Since X is an integer which reserves 2 bytes in memory (16 bits as specified in the table in page 7), it need be saved as 0x0005, so two instructions are needed to load the first byte into location 0x70 and another two to move the rest of the number into location 0x71.

If a simple one statement instruction was assembled like this, imagine how would complex statements be translated like for loops and if statements. Moreover, some compilers are more efficient than others, which give you optimized shorter assembly codes which might not be easy to understand.

Moreover, function placement spans through multiple pages in program memory, the code might not be placed in consecutive order into memory by the compiler; further overhead instructions to switch between pages are common.

In addition, the use of built-in library functions will further complicate stepping through assembly codes line by line as these functions are often provided as a black box for the developer to use with no interest in their details.

For this, it might be difficult for the inexperienced to understand the assembly code generated by compilers, and stepping into assembly code one instruction at a time might be a headache. ***It is often advised to place breakpoints at points of interest and run the program till it halts at the required breakpoints and analyze the outputs in the watch window.***

Control and Repetition Statements

- IF...ELSE statements

```
if (expression1)
{
  statement 1;
        .
        .
  statement n;
}
else
{
  statement 1;
        .
        .
  statement n;
}
```

Example Code 5:

```
if (a==0) //If a is equal to 0
{
        b++; // increase b and c by 1
        c++;
}
else
{
        b--; //decrease b and c by 1
        c--;
}
```

- WHILE loop

```
while (expression)
{
  statement 1;
  statement 2;
        .
        .
  statement n;
}
```

Example Code 6:

```
while (a>=1 ) && (a <=10) //As long as 1<=a <= 10
{
        b = b + 3;
        c = a%b;
}
```

- FOR loop

```
for (expr1; expr2; expr3)
{
  statement 1;
  statement 2;
        .
        .
  statement n;
}
```

Example Code 7:

```
for (i = 0 ; i < 100 ; i++) //loop 100 times
{
        B = B + i + A%i;
}
```

C for PIC

The preceding discussion introduced the C language in a broad concept. Now, we will draw an example of how to use C with the PIC microcontroller. Actually, it is fairly simple where besides user defined variables, the PIC registers are also used in the context of programs.

The microcontroller is completely controlled by registers. All registers used in MPLAB HI-TECH have exact the same name as the name stated in the datasheet. Registers can be set in different ways, following are few examples:

```
TRISB   = 0b00000000;  //TRISB is output
PORTC  = 255;          //All pins of PORTC are made high
PORTD  = 0xFF;         //All pins of PORTD are made high
PORTB  = 170;          //Pin B7 on, B6 off, B5 on, B4 off, etc.
TRISB   = 0b11110010;  //Pin RB7, RB6, RB5, RB4 and RB1 are input, other bits are outputs.
OPTION=0xD4            //PSA assigned to TMR0, Prescalar = 32, TMR0 clock source is the internal instruction cycle
                       //clock, External interrupt is on the rising "refer to datasheet"
```

To set or reset one single bit in a register (one of the 8 bits), the pin name is used and, the names of the bits are also as specified and used in the datasheet.
Some examples:

```
RB0 = 1 //Pin B0 on
RB7 = 0 //Pin B7 off
```

Example Program 8: Periodically switch a LED connected to RD0 on and off

```
#include <htc.h>
// if the whole function is placed before the main function, there is no need for a prototype
void Wait()
{
        unsigned char i;
        for(i=0; i<100; i++)
        _delay(60000);        //built in function .. more info next page
}

void main()
{
  //Initialize PORTD -> RD0 as Output
        TRISD=0b11111110;

  //Now loop forever blinking the LED.
        while(1)
        {
           RD0 = 1;    //LED on
           Wait();

           RD0 = 0;    //LED off
           Wait();
        }
}
```

To simulate the above example code, you can either select PORTD from the ADD SFR drop down menu or choose _PORTDbits from the ADD SYMBOL drop list, click on the + sign to expand and see the individual bits.

Place your break points on both Wait() instructions and run the code.

BUILT IN LIBRARY FUNCTIONS

The C standard libraries contain a standard collection of functions, such as string, math and input/output routines. The declaration or definition for a function is found in the htc.h and other libraries files which are to be included whenever necessary. Some of these functions are listed below, the syntax of each and a brief description follows.

Delay functions

_DELAY

Synopsis

```
#include <htc.h>
void _delay(unsigned long cycles);
```

Description

This is an inline function that is expanded by the code generator. The sequence will consist of code that delays for the number of cycles that is specified as argument. The argument must be a literal constant.
An error will result if the delay period requested is too large. **For very large delays, call this function multiple times.**

```
//Example

#include <htc.h>
int A;


void main (void)
{
        A = A | 0x7f;
        _delay(10); // delay for 10 cycles
        A = A & 0x85;
 }
```

__DELAY_MS, __DELAY_US

Synopsis

```
__delay_ms(x) // request a delay in milliseconds
__delay_us(x)  // request a delay in microseconds
```

Description

As it is often more convenient request a delay in time-based terms rather than in cycle counts, the macros __delay_ms(x) and __delay_us(x) are provided. These macros simply wrap around _delay(n) and convert the time based request into instruction cycles based on the system frequency.
These macros require the prior definition of preprocessor symbol _XTAL_FREQ. This symbol should be defined as the oscillator frequency (in Hertz) used by the system.

```
        //Example

        #include <htc.h>
        int A;
        #define _XTAL_FREQ  4000000

        void main (void)
{
        A = A | 0x7f;
         __delay_ms(10); // delay for 10 ms
        A = A & 0x85;
}
```

Arithmetic functions

In addition to the htc.c library, other libraries such as Standard Library <stdlib.h> or C Math Library <math.h> need be included in the project for making use of many useful built-in functions. Make sure you include the appropriate header files for each library before making use of its functions or else build errors will be present.

ABS, POW, LOG, LOG10, RAND, MOD, DIV, CEIL, FLOOR, NOP, ROUND, SQRT are required.. refer to the datasheet for the documentation of the others

ABS ***Synopsis*** #include <stdlib.h> int abs (int j) ***Description*** The abs() function returns the absolute value of the passed argument j.	**POW** ***Synopsis*** #include <math.h> double pow (double f, double p) ***Description*** The pow() function raises its first argument, f, to the power p.
LOG, LOG10 ***Synopsis*** #include <math.h> double log (double f) double log10 (double f) ***Description*** The log() function returns the natural logarithm of f. The function log10() returns the logarithm to base 10 of f.	**RAND** ***Synopsis*** #include <stdlib.h> int rand (void) ***Description*** The rand() function is a pseudo-random number generator. It returns an integer in the range 0 to 32767, which changes in a pseudo-random fashion on each call.

Trigonometric functions

SIN, COS, TAN, COS, ASIN, ATAN …… refer to the data sheet for the others

<table>
<tr>
<td>

- **SIN**

Synopsis

#include <math.h>
double sin (double f)

Description

This function returns the sine function of its argument.it is very important to realize that C uses radians, not degrees to perform these calculations! If the angle is in degrees you must first convert it to radians.

</td>
<td>

- **COS**

Synopsis

#include <math.h>
double cos (double f)

Description

This function yields the cosine of its argument, which is an angle in radians. The cosine is calculated by expansion of a polynomial series approximation.

</td>
</tr>
<tr>
<td colspan="2">

```
// Example:
#include <htc.h>
#include <math.h>
#include <stdio.h>
#define C 3.141592/180.0
double X,Y;
void main (void)
{
double i;
X=0;
Y=0;
for(i = 0 ; i <= 180.0 ; i += 10)
{X= sin(i*C);
Y= cos(i*C);
}
}
```

</td>
</tr>
</table>

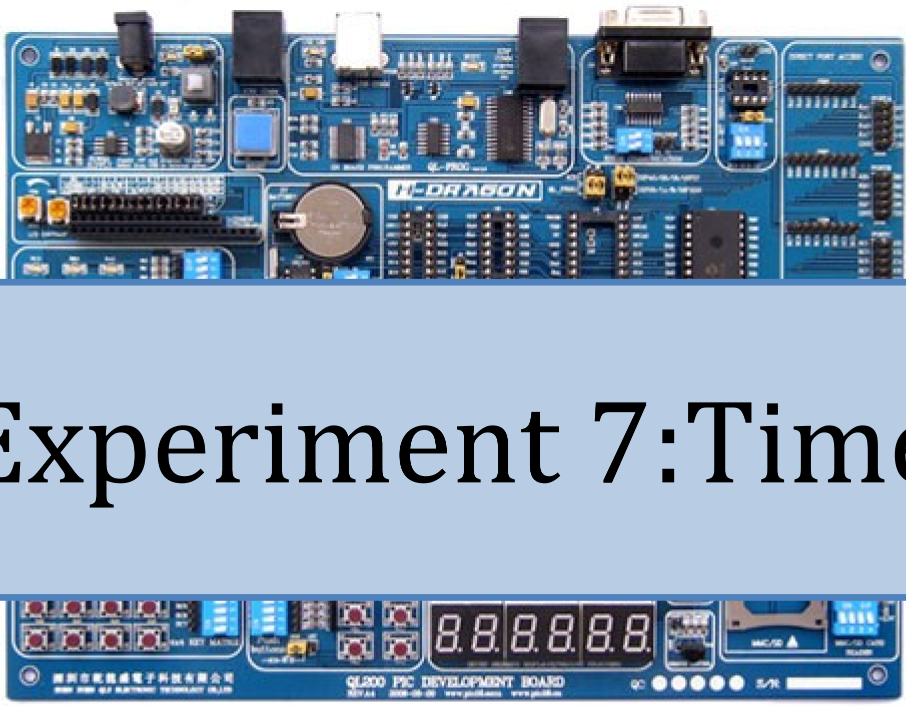
- define directive

You can use the **#define** directive to give a meaningful name to a constant in your program.

```
#define identifier constant
```

Example:
#define COUNT 1000

University of Jordan
Faculty of Engineering and Technology
Department of Computer Engineering
Embedded Systems Laboratory 0907334



7 Experiment 7:Timers

Objectives

- To become familiar with hardware timing modules provided by the PIC 16F877A
- To become familiar with the concept of 7 segment multiplexing

Written by Eng. Ashraf Suyyagh and Eng. Enas Jaara |

Pre-lab

You are required to review the following in order to be fully prepared for the experiment, refer back to both your text book and the Microchip PIC datasheets whenever you find it necessary.

- The operation of the Timer0 Module and the related OPTION_REG settings
- The Operation of Timer2 Module and its associated PR2 and T2CON registers
- The External interrupt on RB0.
- Context saving and retrieval while using interrupts.

DO NOT COME TO THE LAB UNPREPARED

The Idea behind the Code

The code is simply a 2-digit stopwatch (Max. 60 seconds) which has its output in decimal format shown on 2 Seven segments displays, it simply does the following:

1. Initially, when the system is display 00 on the 2 seven segments displays.
2. The stopwatch remains in this condition until a (Start/Stop) button is pressed, after which you will observe the following count:

   00, 01, 02,03,04,05... 58, 59, 00, 01…

3. The stopwatch will count this way indefinitely until the (Start/Stop) button is pressed again where the count display will remain as is (Hold). When the (Start/Stop) button is pressed another time it will continue counting from its last count.

   Counting Example:

   00, 00 (Start/Stop), 01, 02, 03, 04, 05, (Start/Stop), 05, 05, 05, (Start/Stop), 06, 07, 08 ……

How did we write this code?

- In this experiment we will use PIC16877A microcontroller and an oscillator with a value of 4 MHz
- We made the decision to use TMR0 to count time (1 second), and to use the external interrupt RB0 as the Start_Stop button.
- We have also defined a register: Start_Stop , which if it has the value 0x00, then the stopwatch will stop, if it has the value 0xFF then the stopwatch will count.
- Now, the first problem, if Fosc is 4 MHz, then the instruction cycle is 1µs, at this speed the maximum count of TMR0 at maximum pre-scalar settings is 256 x 256 x 1µs = 65.536 **ms** which is far below the one second time needed (1,000,000 µs).

So what do we do now?

Since we need to count 1,000,000 µs, use your mind, calculator, sheet of paper, pencil and luck ☺ to find three numbers X, Y, Z (all under 255, maximum register width) where *X* x *Y* x *Z* = 1,000,000 µs and with the condition that one of the numbers should satisfy 2^N (one of the values of TMR0 pre-scalar)

We have found that 250 x 32 x 125 = 1,000,000 (notice that $32 = 2^5$) so we do the following:

- Let 32 be the pre-scalar
- 250 be TMR0 count. (that is TMR0 will be initialized to 256 – 250 = 6)

  So each interrupt, TMR0 will count 32 x 250 = 8000 µs = 8 **ms**.

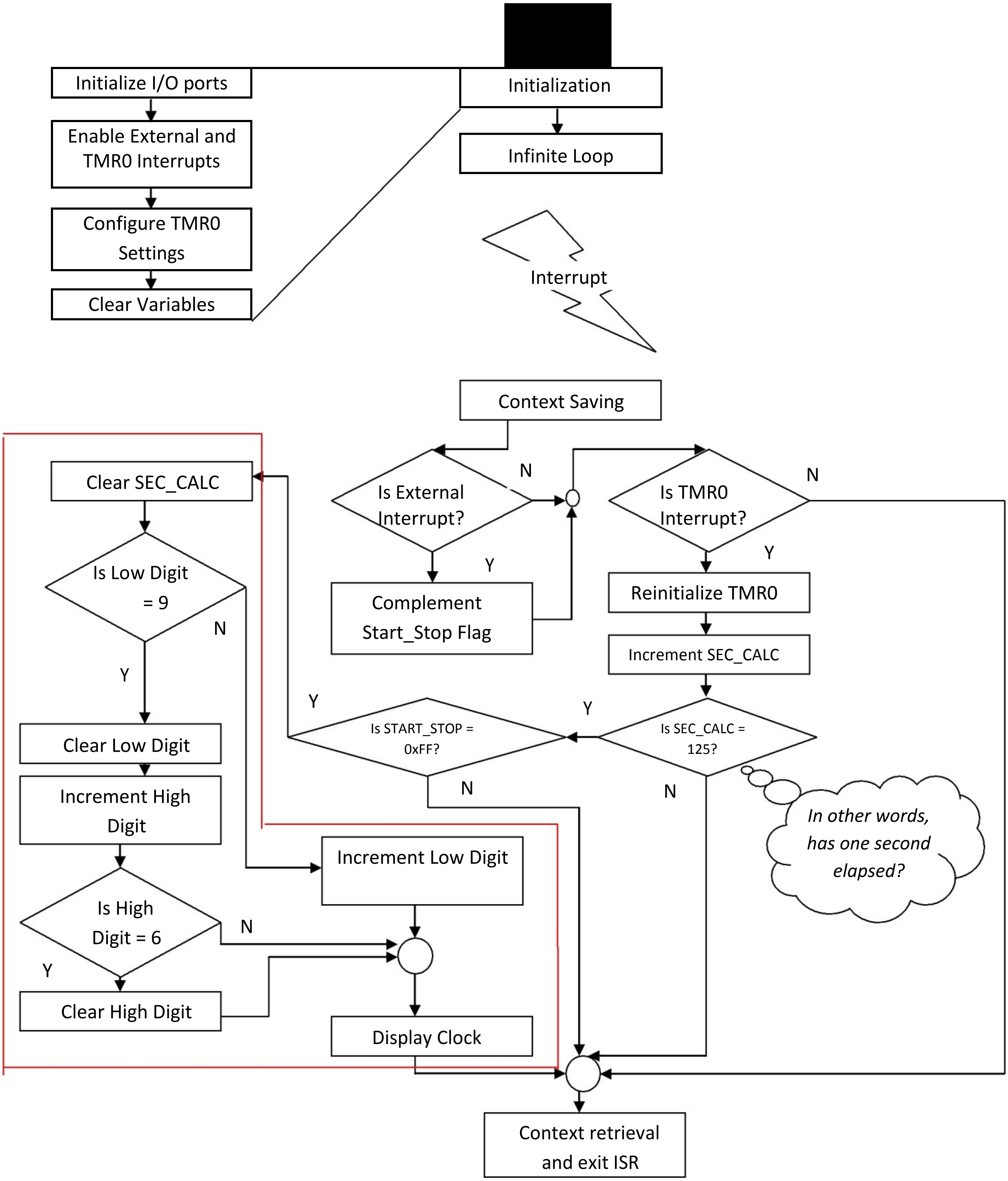
- Each interrupt, a register which we defined: SEC_CALC will be incremented, nd it will be checked for the value 125 to know whether we reached 1 second or not.

SEC_CALC	No. of TMR0 Interrupts	Total Time elapsed
0	0	0 ms
1	1	8 ms
2	2	16 ms
3	3	24 ms
4	4	32 ms
5	5	40 ms
--		
124	124	992 ms
125	125	1000 ms (1 second)
0	Cleared in order to count the next second correctly	
1	126	0 ms
2	127	8 ms

Notice in the flow chart below that in order for the clock digits to update two conditions should be satisfied:

1. One second has elapsed (SEC_CALC = 125)
2. The clock should be in the counting mode (START_STOP = 0xFF)

If either condition fails, the clock will not count but hold its previous count on the display unchanged

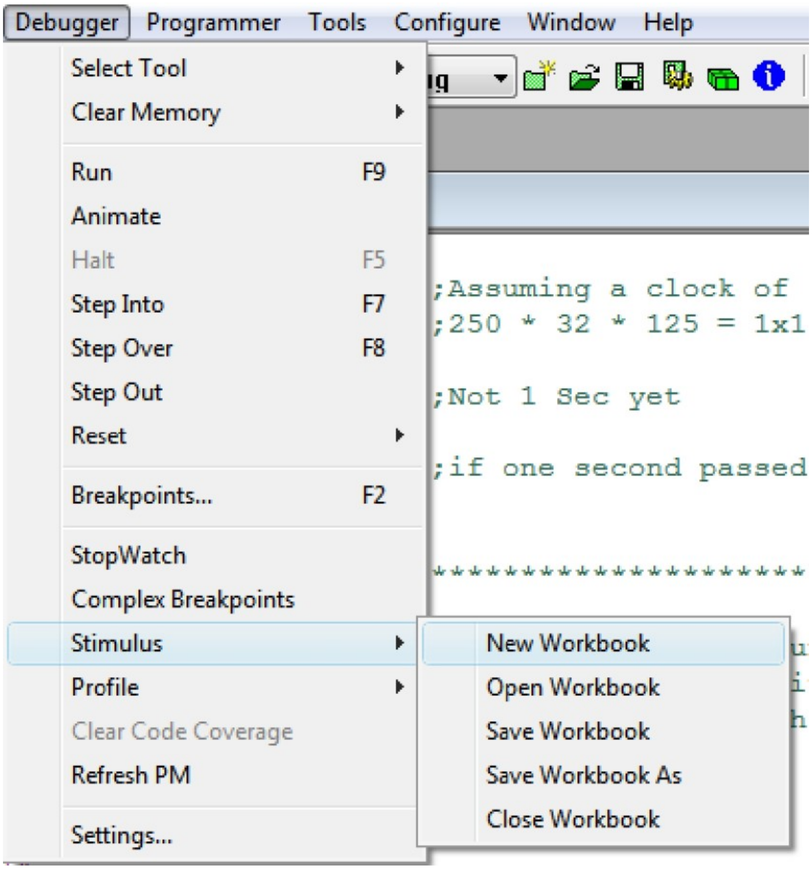


[redacted]
Initialize I/O ports
Enable External and TMR0 Interrupts
Configure TMR0 Settings
Clear Variables
Initialization
Infinite Loop
Interrupt
Context Saving
Is External Interrupt?
N
Y
Complement Start_Stop Flag
Is TMR0 Interrupt?
N
Y
Reinitialize TMR0
Increment SEC_CALC
Is SEC_CALC = 125?
Y
N
In other words, has one second elapsed?
Is START_STOP = 0xFF?
Y
N
Clear SEC_CALC
Is Low Digit = 9
N
Y
Clear Low Digit
Increment High Digit
Is High Digit = 6
N
Y
Clear High Digit
Increment Low Digit
Display Clock
Context retrieval and exit ISR

Do not forget to set the speed to 4MHz for this code in MPLAB

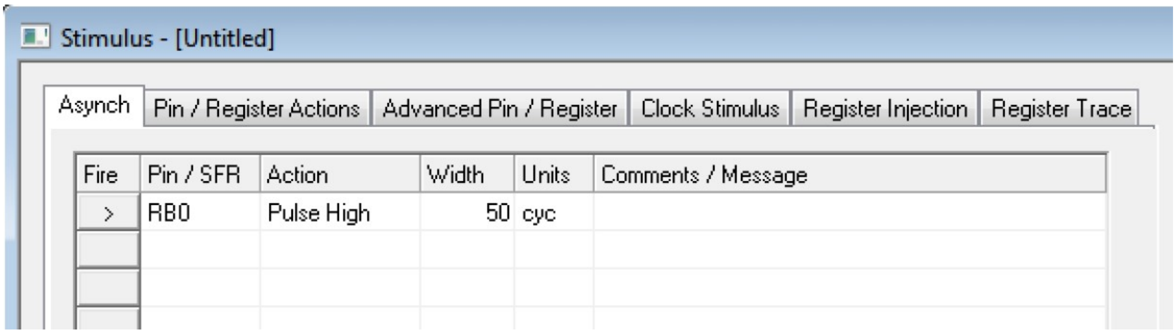
How to simulate this code in MPLAB?

You have learnt so far that in order to simulate inputs to the PIC, you usually entered them through the Watch window. However, this is only valid and true when you are dealing with internal memory registers. In order to simulate external inputs to the PIC pins, we are to use what is called a Stimulus.



There are multiple actions which you can apply to an input pin, choose whatever you see as appropriate to simulate your program. Here we have chosen to simulate the button press as a pulse.

1. Add Low_Digit, High_Digit and Start_Stop to the watch window.
2. Place a break point at line 79 (Instruction return). This will allow us to see the change to Start_Stop, if 0xFF the stopwatch counts, else it stops.
3. Place another breakpoint at line 105 (Instruction return), this will allow us to observe how Low_Digit and High_Digit change
4. Run your code, you will observe nothing except that the values in the watch window are all zeros.



5. Now Press “Fire”, the arrow next to the RB0 in the Stimulus pin, what do you observe?
6. Now, press “run” again, observe how the values of Low_Digit and High_Digit change whenever you reach the breakpoint.
7. Press “fire” again, how do the values in Low_digit and High_Digit change now?

Example Code

```
;**********************************************************************
; Connections:
;                   Input:
;                          Pushbutton : RB0
;                   Output:
;                          7-segment A-G:  PortD 0-6
; hardware requests : S6 set ON, first and second set ON,S1 ON ,S12 and S13 OFF
__CONFIG_DEBUG_OFF&_CP_OFF&_WRT_HALF&_CPD_OFF&_LVP_OFF&_BODEN_OFF&_PWRTE_O
FF&_WDT_OFF&_XT_OSC
;**********************************************************************
INCLUDE "P16F877A.INC"
;**********************************************************************
; CBLOCK Assignments
;**********************************************************************
                CBLOCK          0X20
                     Delay_reg
                     STATUSTEMP
                     LOW_DIGIT         ; holds the digit to be displayed on first 7-segment
                     HIGH_DIGIT        ; holds the digit to be displayed on second 7-segment
                     SEC_CALC          ; used in calculating the elapse of one second
                     START_STOP        ; user defined flag which if filled with 1's the stop watch
                                       ;counts, else halts
                ENDC
 ;**********************************************************************
                 ORG  0X000
                 GOTO         MAIN
                 ORG  0X004
                 GOTO         ISR
 ;**********************************************************************
 MAIN
                 CALL         INITIAL
MAINLOOP
                 CALL         DisplayClock
                 GOTO         MAINLOOP
 ;**********************************************************************
 INITIAL
                 BANKSEL      TRISA
                 CLRF         TRISA           ;TRISA and TRISD as outputs
                 CLRF         TRISD
                 MOVLW        01
                 MOVWF        TRISB           ;RB0 as input (External Interrupt enabled), RB1-RB7
                                              ; as outputs
                 BSF          INTCON, GIE     ;TMR0 and External Interrupts Enabled, their
                                              ; flags cleared
                 BSF          INTCON, INTE
                 BSF          INTCON, TMR0IE
                 BCF          INTCON, INTF
                 BCF          INTCON, TMR0IF
                 MOVLW        0XD4    ;PSA assigned to TMR0, Prescalar = 32, TMR0 clock source
                                      ;is the internal
                 MOVWF        OPTION_REG      ;instruction cycle clock, External interrupt is on the
                                              ;rising egde
```

```
            BANKSEL     ADCON1
            MOVLW       06H
            MOVWF       ADCON1      ;set PORTA as general Digital I/O PORT

            BANKSEL     TMR0        ;TMR0 to update 256 – 6 = 250
            MOVLW       0X06
            MOVWF       TMR0
            CLRF        LOW_DIGIT   ;Initially, the number to be displayed is 00
            CLRF        HIGH_DIGIT
            CLRF        SEC_CALC    ;0 ms has passed
            CLRF        START_STOP  ;stopwatch is initially stopped
            MOVLW       0FFH
            MOVWF       PORTD       ;close all display
            RETURN
;*************************************************************************
ISR
            BTFSC       INTCON, INTF        ;External Interrupt has higher priority
            CALL        START_STOP_SUB
            BTFSC       INTCON, TMR0IF
            CALL        TMR0_CODE
            RETFIE
;*************************************************************************
START_STOP_SUB
            BCF         INTCON, INTF        ;clear external interrupt flag
            COMF        START_STOP, F       ;thus halting or starting the stopwatch
            RETURN
;*************************************************************************
TMR0_CODE
            BCF         INTCON, TMR0IF      ;Clear TMR0 Flag
            MOVLW       0X06                ;Reinitialize TMR0
            MOVWF       TMR0
            INCF        SEC_CALC, F
            MOVLW       .125                ;Assuming a clock of 4MHZ, we need
            SUBWF       SEC_CALC, W         ; 250 * 32 * 125 = 1×106 µs = 1 sec
            BTFSS       STATUS, Z
            GOTO        ENDTMR0
            BTFSC       START_STOP, 0
            CALL        UPDATE_DIGITS       ;if one second passed, update digits
ENDTMR0
            RETURN
;*************************************************************************
UPDATE_DIGITS
            CLRF        SEC_CALC        ;Cleared so as to count the next 1 sec correctly
            MOVF        LOW_DIGIT, W    ; If previous low digit is not 9, increment low digit
                                        ;by one
            SUBLW       0X09            ; else, increment high digit by one and clear low digit
            BTFSC       STATUS, Z
            GOTO        UPDATE_HIGH_DIGIT
            GOTO        UPDATE_LOW_DIGIT
END_UPDATE
            CALL        DisplayClock    ; Update clock display
```

```
                RETURN
;***********************************************************************************
UPDATE_LOW_DIGIT
                INCF        LOW_DIGIT, F
                GOTO        END_UPDATE
UPDATE_HIGH_DIGIT
                CLRF        LOW_DIGIT
                INCF        HIGH_DIGIT, F
                MOVF        HIGH_DIGIT, W
                SUBLW       6           ; if high digit reaches 6 (that is number = 60, 1 Minute),
                                        ;reset
                BTFSC       STATUS, Z
                CLRF        HIGH_DIGIT
                GOTO        END_UPDATE
;***********************************************************************************
DisplayClock              ;7 segment digit  multiplexing  ; see appendix 3
                MOVF        LOW_DIGIT,W
                CALL        Look_TABLE
                MOVWF       PORTD
                BCF         PORTA,1             ;enable first 7_segment Display
                CALL        DELAY
                BSF         PORTA,1
                MOVF        HIGH_DIGIT,W
                CALL        Look_TABLE
                MOVWF       PORTD
                BCF         PORTA,0             ;enable second 7_segment Display
                CALL        DELAY
                BSF         PORTA,0
                RETURN
;***********************************************************************************
Look_TABLE
                ADDWF     PCL , 1
                RETLW     B'11000000'     ;'0'
                RETLW     B'11111001'     ;'1'
                RETLW     B'10100100'     ;'2'
                RETLW     B'10110000'     ;'3'
                RETLW     B'10011001'     ;'4'
                RETLW     B'10010010'     ;'5'
                RETLW     B'10000010'     ;'6'
                RETLW     B'11111000'     ;'7'
                RETLW     B'10000000'     ;'8'
                RETLW     B'10010000'     ;'9'
;********************** delay subprogram ***************************************
Delay
                MOVLW      0FFH
                MOVWF      Delay_reg
L1              DECFSZ     Dealy_reg,F
                GOTO       L1
                RETURN
END
```

Appendix 1 - Timer2 Module

Prepared by Eng. Enas Ja'ra

Timer2 is an 8-bit timer with a prescaler and a postscaler, , it is connected only to an internal clock - (FOSC/4) and it has Interrupt on overflow feature.

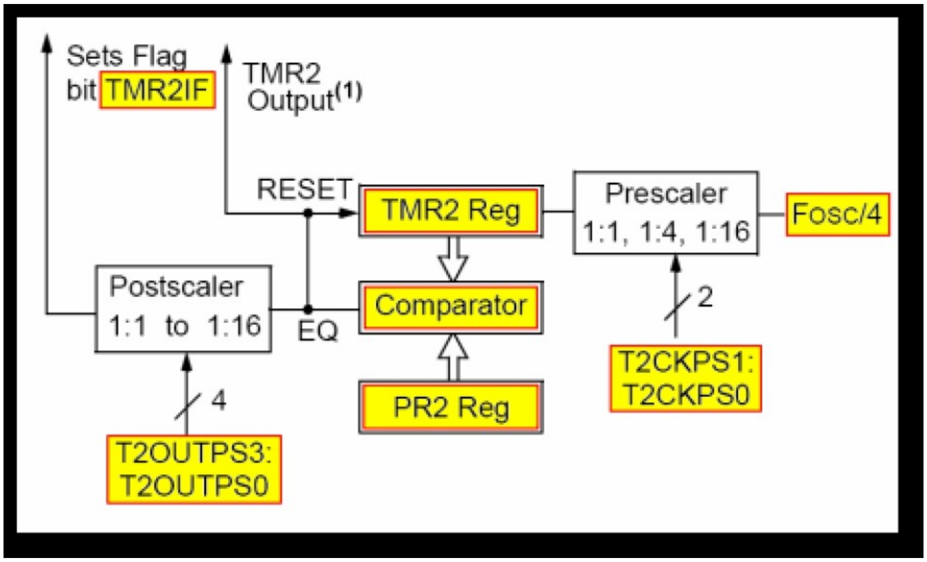
Timer2 has 2 count registers: TMR2 and PR2. The size of each registers is 8-bit in which we can write numbers from 0 to 255.The TMR2 register is readable and writable and is cleared on any device Reset. PR2 is a readable and writable register and initialized to FFh upon Reset.

Register TMR2 is used to store the "initial" count value (the value from which it begins to count). Register PR2 is used to store the "ending" count value (the maximum value we need/want to reach). ie: using Timer2 we can determine the started count value, the final count value, and the count will be between these two values. The Timer2 increments from 00h until it matches PR2 and then resets to 00h on the next increment cycle.

Prescaler and Postscaler :

Each allows making additional division of the frequency clock source.

- *Prescaler* divides the frequency clock source BEFORE the counting takes place at the register TMR2, thus the counting inside the TMR2 register is performed based on the divided frequency clock source by the Prescaler.
- *Postscaler* divides the frequency that comes out of the Comparator again for the last time. The match output of TMR2 goes through a 4-bit postscaler (which gives a 1:1 to 1:16 scaling) to generate a TMR2 interrupt if enabled (TMR2IF (PIR1 register bit no 1)).



TIMER2 BLOCK DIAGRAM

All the necessary settings are controlled from with T2CON Register

T2CON: TIMER2 CONTROL REGISTER (ADDRESS 12h)

U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0
bit 7							bit 0

T2CKPS1:T2CKPS0: Timer2 Clock Prescale Select bits

T2CKPS1:T2CKPS0 (T2CON<1:0>).
00 = Prescaler is 1
01 = Prescaler is 4
1x = Prescaler is 16

TMR2ON: Timer2 On bit

TMR2ON (T2CON<2>)
1 = Timer2 is on
0 = Timer2 is off

TOUTPS3:TOUTPS0: Timer2 Output Postscale Select bits

TOUTPS3:TOUTPS0 (T2CON<6:3>).

0000 = 1:1 postscale
0001 = 1:2 postscale
0010 = 1:3 postscale
•
•
•
1111 = 1:16 postscale

Appendix 2- Watchdog Timer
Prepared by Eng. Enas Ja'ra

A watchdog timer (abbreviated to WDT) is a part of hardware that can be used to automatically detect software anomalies and reset the processor if any occur. A watchdog timer can get a system out of a lot of dangerous situations.

A watchdog circuit is a resistor/capacitor network inside the PIC. This provides a unique clock, which is independent of any external clock that you provide in your circuit. Now, when the Watchdog Timer is enabled, a counter starts at 00 and increments by 1 until it reaches FF. When it goes from FF to 00 (which is FF + 1) then the PIC will be reset, irrespective of what it is doing. The only way we can stop the WDT from resetting the PIC is to periodically reset the WDT back to 00 throughout our program. Now you can see that if our program does get stuck for some reason, the WDT will then reset the PIC, causing our program to restart from the beginning.

In order to use the WDT, we need to know three things. First, how long have we got before we need to reset the WDT, secondly how do we clear it. Finally, we have to tell the PIC programming software to enable the WDT inside the PIC.

WDT Times

The PIC data sheet specifies that the WDT has a period from start to finish of 18mS. This is dependant several factors, such as the supply voltage, temperature of the PIC etc. The reason for the approximation is because the WDT clock is supplied by an internal RC network. The time for an RC network to charge depends on the supply voltage. It also depends on the component values, which will change slightly depending on their temperature. So, for the sake of simplicity, just take it that the WDT will reset every 18mS. We can, however, make this longer by Prescaler. We can program this prescaler to divide the RC clock. The more we divide the RC clock by, the longer it takes for the WDT to reset.

The prescaler is located in the OPTION register, bits 0 to 2 inclusive. Below is a table showing the bit assignments with the division rates and the time for the WDT to time out, Remember these times are irrespective of your external clock frequency.

By default the prescaler is assigned to the other internal timer" TIMR0" . This means that we have to change the prescaler over to the WDT.

Bit 2,1,0	*Rate*	*WDT Time*
0,0,0	1:1	18mS
0,0,1	1:2	36mS
0,1,0	1:4	72mS
0,1,1	1:8	144mS
1,0,0	1:16	288mS
1,0,1	1:32	576mS
1,1,0	1:64	1.1Seconds
1,1,1	1:128	2.3Seconds

Example:

Suppose we want the WDT to reset our PIC after about half a second as a failsafe.

From table the nearest we have is 576mS, or 0.576 seconds.

- We have to reset the "TMR0" to 0.
- reset the WDT and prescaler
- Assign the prescaler to the WDT.
- Select the appropriate prescaler.

```
Banksel   TMR0           ; make sure we are in bank 0
clrf      TMR0           ; TMR0=0;
Banksel   OPTION         ;switch to bank 1
clrwdt                   ;reset the WDT and prescaler
movlw      b'00001101'   ;Select the new prescaler value
movwf     OPTION         ; and assign it to WDT
```

The CLRWDT instruction is used to clear the WDT before it resets the PIC. So, all we need to do is calculate where in our program the WDT will time out, and then enter the CLRWDT command just before this point to ensure the PIC doesn't reset. If your program is long, bear in mind that you may need more than one CLRWDT. For example, if we use the default time of 18mS, then we need to make sure that the program will see CLRWDT every 18mS.

> The CLRWDT instruction clears the WDT and the prescaler, if assigned to the WDT, and prevent it from timing out and generating a device RESET condition.

Example:

This subroutine lights one LED on an 8-LED-row and continuously moves back and forth in this fashion.

```
;**************************************************************
            include   "p16f917.inc"
;**************************************************************
COUNT1      equ     20H             ; DELAY Loop register.
COUNT2      equ     21H             ; DELAY Loop register.
COUNT       equ     22H
;******************************************************************
ORG 0x00
            goto initial
;****************************************************************
initial
            clrf   TMR0                  ;Clear TMR0
            Banksel TRISB
            clrwdt                       ;reset the WDT and prescaler
            movlw b'00001011'            ;Select the prescaler value and assign
            movwf   OPTION_REG           ;it to WDT,WDT time to reset 144mS
            bsf STATUS,RP0
            movlw    00H
            movwf   TRISB
            bcf STATUS,RP0
            movlw 8
            movwf COUNT

MAIN
            movlw    01H
            movwf   PORTB

Rotate_Left                              ; Move the bit on Port B left, then right.
            call    DELAY
            rlf         PORTB, F
            btfss       STATUS, C
            goto        Rotate_Left
Rotate_Right
            call        DELAY
            rrf                     PORTB, F
            btfss       STATUS, C
            goto        Rotate_Right
            goto        Rotate_Left
; ****************************************************
 ; Subroutine to give a delay between bit movements.
 ;Total of 42.7 mS
; ****************************************************
DELAY
  MOVLW             0X6F
            MOVWF   COUNT2
L11         MOVLW   0X7F
            MOVWF COUNT1

LOOP2
            DECFSZ  COUNT1,F
            GOTO     LOOP2
LOOP1

            DECFSZ  COUNT2,F
            GOTO     L11
            CLRWDT                 ; This simply resets the WDT.
             return                ;  Return from our original DELAY subroutine

 END
```

- The instruction at Line 59 resets the WDT, Comment out or removes this command to see the WDT in action. It should reset the PIC.
- If you comment out, or remove the CLRWDT command, you will find that the PIC will not go past lighting the fifth LED. This is because the WDT is resetting the PIC. With the CLRWDT in place, the program works as it should.

Appendix 3- 7 Segment Multiplexing

Some kits like QL 200 development kit provide multiplexed multi 7 segment digit displays in single packages; **Multiplexed displays** are electronic displays where the entire display is not driven at one time. Instead, sub-units of the display are multiplexed.

In multiplexed 7 segment applications (see Figure 1) the LED segments of all the digits are tied together so if you send date to any one of the segment , it will displayed on both segments to prevent that the common pins of each digit are turned ON separately by the microcontroller. When each digit is displayed only for several milliseconds, the eye cannot tell that the digits are not ON all the time. This way we can multiplex any number of 7-segment displays together. For example, to display the number 24, we have to send 2 to the first digit and enable its common pin. After a few milliseconds, number 4 is sent to the second digit and the common point of the second digit is enabled. When this process is repeated continuously, it appears to the user that both displays are ON continuously.

- The display can be controlled from the microcontroller as follows

- Send the segment bit pattern for digit 1 to segments **a** to **g**
- Enable digit 1.
- Wait for a few milliseconds.
- Disable digit 1.
- Send the segment bit pattern for digit 2 to segments **a** to **g**
- Enable digit 2
- Wait for a few milliseconds
- Disable digit 2.
- Repeat the above process continuously

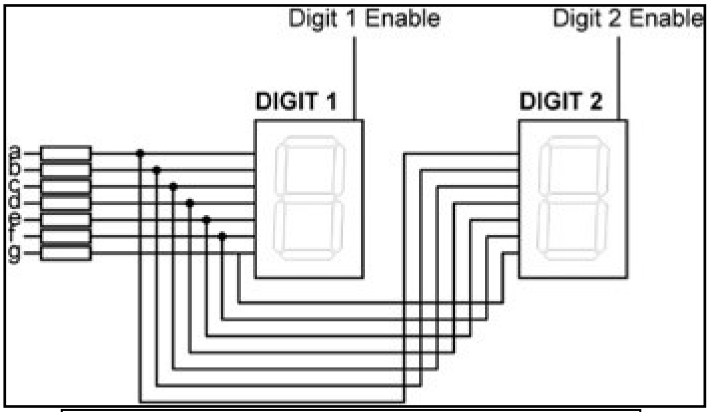
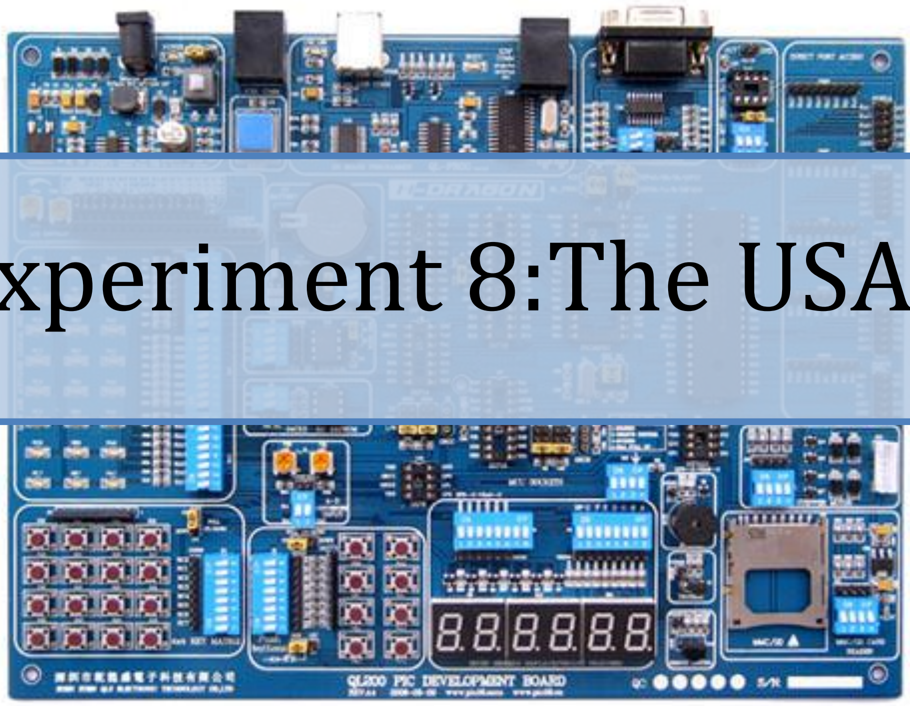


Figure 1: Two multiplexed 7-segment displays

University of Jordan
Faculty of Engineering and Technology
Department of Computer Engineering
Embedded Systems Laboratory 0907334

8 Experiment 8:The USART



Objectives

- Introduce the USART module of the PIC 16series through an industrial example.
- To become familiar with the serial communications using PIC and RS232 Protocol.
- Become familiar with serial communication testing techniques either in software and hardware

Pre-lab

You are required to review the following in order to be fully prepared for the experiment, refer back to both your text book and the Microchip PIC datasheets whenever you find it necessary.

- The general operation of the USART in asynchronous mode.
- Familiarize yourself with the following registers and their individual bit functions: TXSTA, RCSTA, TXREG, RCREG, and SPBRG.
- Calculating baud rate speeds.
- The PIE and PIR registers in the 16F877A.

The Idea behind the Code

In a certain factory, a modern computerized machine is serially connected to a control computer. Once the machine is powered on, it sends a message to the control room indicating that it is ready to receive commands. After reading the message, an operator sends commands to the machine through the control computer. In this experiment, since the there is no physical machine to carry out the commands, the commands will be simply displayed on 7 segments display.

The simple flow of the program is:

- ➢ Initialize I/O, enable interrupts, configure USART settings: baud rate, transmitter and receiver settings
- ➢ Send message to control computer
- ➢ Wait continuously "Loop" until commands are received from control computer, when received display them on 7 segments display

STEP 1: INITIALIZE I/O, ENABLE INTERRUPTS …

- RC6 is reserved by Microchip design specifications for serial data transmission, therefore configure as o/p
- RC7 is reserved by Microchip design specifications for serial data reception, therefore configure as i/p
- PORTD will be connected to the 7 segments display, so configure as o/p
- Baud rate is agreed to be 9600 bps, review datasheet or do hand calculations to find that SPBRG has to be filled by 25 and high baud rate will be enabled (BRGH = 1) in order to achieve this speed.
- Enable serial port (SPEN = 1), enable receiver (CREN = 1), enable transmitter (TXEN = 1)
- Since we want to use asynchronous mode (SYNC = 0).
- We have agreed to use receiver interrupt to know whether the machine received commands from the control station or not, so (GIE = 1), (PEIE = 1) and (RCIE = 1).

STEP 2: SEND MESSAGE TO CONTROL COMPUTER

The machine status message which reads "Machine ready to receive commands" has a length of 33 characters and is sequentially stored in a look up table. Where the first entry in the table is the letter "M", the second is "a", third is "c" and so on … To send the message, the look-up table is to be accessed 33 times with the first time adding 0 to PCL to retrieve "M", the second time adding 1 to PCL to retrieve "a", the third time adding 2 to PCL to retrieve "c" and so on … The message length is stored in a variable which is decremented each time the look up table is accessed and is checked to see if this variable reached 0 or not to indicate end of message.

After each message character is retrieved from the look-up table it is sent to TXREG, assuming the USART is configured properly, the character will be serially sent at the designated speed.

We can't send the next character immediately to TXREG while there is data still being transmitted or residing in the transmitter's TXREG, this will overwrite the data to be transmitted and therefore be lost. In consequence, we have four ways to detect if transmission of the previous frame has finished or not before sending the next one:

1. Use Interrupts (when transmission is finished, program flow will be interrupted and you can send the next character inside ISR)

2. Poll the TXIF interrupt flag found in PIR1 register

3. Poll the TRMT flag found in TXSTA register **<u>(which is the method employed in this experiment)</u>**

4. Insert a time delay calculated to be larger than the delay time needed to transmit the character frame Ex. If speed is 9600 bps, this means the time needed to send a frame asynchronously is:

9600   1,000,000µs (1s)
10   X

X = 1041 µs = 1.041ms so insert a delay larger than this value before transmitting the next frame.

After the whole message is sent, the code goes into an infinite loop waiting to receive commands.

STEP 3: COMMANDS ARE RECEIVED FROM CONTROL COMPUTER

When characters are received from a control computer, the character frame will reside in the RCREG register and the RCIF flag will be set high (Remember that interrupt flags are set high whenever their event occurs regardless whether the sources were enabled or not). But how do we know the moment the command is received and ensure that we get all commands without losing any of them?

Similar to what has been discussed above. We have three methods to ensure data is read at **sufficient time periods without any data loss:**

1. Use Interrupts (**<u>which is the method employed in this experiment</u>**, when a command is received, the program flow will be interrupted and you can read RCREG inside ISR)

2. Poll the RCIF flag found in PIR1 register

3. Periodically read RCREG at sufficient time intervals.

Another important issue is how to check if the date received is erroneous or not? There are two types of errors in serial data communications which the PIC can detect and flag:

1. Framing errors occur due to the difference in the speed of communication between the transmitter and receiver (not correctly set to match each other). This error is detected when a stop bit is received as CLEAR and the framing error bit (FERR) in the RCSTA register is set to indicate occurrence. The FERR pin is set/cleared for every frame received to indicate if there is speed mismatch! Therefore, the FERR value will be updated with every coming frame and it is necessary to read RCSTA value before RCREG to check if we are receiving the data correctly.

2. Overrun errors: The receiver module has a two-level deep buffer in which the received data is stored. Data received in the RSR register ultimately fill the buffer. However, if the two buffer locations are already occupied, and a third frame of data is being shifted into the RSR, once it is complete, it will not be stored in the buffer and thus be lost, and hence an overrun error occurs. Flag OERR in the RCSTA register is set to indicate this error occurrence. Once this OERR bit is set, no further data is received! The FIFO buffer is cleared by reading data in the RCREG, that is, it needs two RCREG reads to empty the buffer! Furthermore, once set, the OERR bit can only be cleared in software by clearing and setting the CREN bit. To avoid overrun errors, the user should always make sure to read data at appropriate speeds such that the buffers won't become full!

3. Parity Errors: used to detect odd number of erroneous bit transmissions. This is done by enabling the 9^{th} bit mode in the RCSTA register "RX9 bit". However, no hardware is present to calculate and check for parity, therefore, the sender should write appropriate code to calculate desired parity (odd/even) and place the result in the TX9D pin in the TXSTA register before sending the frame. An equivalent code should read the received parity RX9D from the RCSTA register calculate parity and check for a match!

Code Example

```
Function
In a certain factory, a modern computerized machine is serially connected to a control computer. Once the machine is powered on, it sends a message to the control room indicating that it is ready to receive commands. After reading the message, an operator sends commands to the machine through the control computer. In this experiment, since the there is no physical machine to carry out the commands, the commands will be simply displayed on 7 segments display.
;
The simple flow of the program is:
    1.  Initialize I/O, enable interrupts, configure USART settings: baud rate, transmitter and receiver settings
    2.  Send message to control computer
    3.  Wait continuously "Loop" and wait until commands are received from control computer, when received display them on 7 Segments Display

;Hardware Connections
            Inputs
                        RC7: USART Receiver pin
            Outputs
                        RC6: USART Transmitter pin
                        PORTD 0 -6: 7 segment display
                        RA0 is connected to 7-Segment Digit Enable
;************************************************************************************************
        include "p16f9877a.inc"
;************************************************************************************************
; User-defined variables
        cblock  0x20
                WTemp                   ; Must be reserved in all banks
                StatusTemp              ; reserved in bank0 only
                Counter
                BLNKCNT
                MSG
        endc
        cblock  0x0A0
                WTemp1
        endc
        cblock  0x120
                WTemp2
        endc
        cblock  0x1A0
                WTemp3
        endc
;************************************************************************************************
; Macro Assignments
push    macro
        movwf       WTemp           ;WTemp must be reserved in all banks
        swapf       STATUS,W        ;store in W without affecting status bits
        banksel     StatusTemp      ;select StatusTemp bank
        movwf       StatusTemp      ;save STATUS
        endm


pop     macro
        banksel     StatusTemp              ;point to StatusTemp bank
        swapf       StatusTemp,W            ;unswap STATUS nibbles into W
```

```
        movwf       STATUS              ;restore STATUS (which points to where W was
                                        ;stored)
        swapf       WTemp,F             ;unswap W nibbles
        swapf       WTemp,W             ;restore W without affecting STATUS
        endm
;**********************************************************************************
; Start of executable code
        org         0x00                ; Reset Vector
        goto        Main
        org         0x04                ; Interrupt Vector
        goto        IntService
;**********************************************************************************
; Main program
; After Initialization, this code sends the message: "Machine ready to receive commands" then
; goes into an infinite loop during which, the program is interrupted if data is received.
;**********************************************************************************
Initial
        movlw       D'25'               ; This sets the baud rate to 9600
        banksel     SPBRG               ; assuming BRGH=1 and Fosc = 4.000 MHz
        movwf       SPBRG

        banksel     RCSTA
        bsf         RCSTA, SPEN         ; Enable serial port
        bsf         RCSTA, CREN         ; Enable Receiver

        banksel     TXSTA
        bcf         TXSTA, SYNC         ; Set up the port for Asynchronous operation
        bsf         TXSTA, TXEN         ; Enable Transmitter
        bsf         TXSTA, BRGH         ; High baud rate used

        banksel     PIE1                ; Enable Receiver Interrupt
        bsf         PIE1,RCIE
        banksel     INTCON
        bsf         INTCON, GIE         ; Enable global and peripheral interrupts
        bsf         INTCON, PEIE
        banksel     TRISD
        clrf        TRISD               ; PORTD is used to display the received commands
        clrf        TRISA
        bcf         TRISC, 6            ; Configuring pins RC6 as o/p, RC7 as i/p for
        bsf         TRISC, 7            ; serial communication
        movlw       06
        movwf       ADCON1

        banksel     PORTD
        clrf        PORTD
        clrf        PORTA
        return
;**********************************************************************************
Main
        Call        Initial
    MainLoop                            ; Prepare to send first character in the message MSG = 0
        Clrf        MSG                 ;  then incremented by on to access every character in
                                        ;.look up table
```

```
SEND
        movf        MSG, W
        call        Message
        movwf       TXREG
TX_not_done
        banksel     TXSTA               ; Polling for the TRMT flag to check
        btfss       TXSTA, TRMT         ; if TSR is empty or not
        goto        TX_not_done
        banksel     MSG
        incf        MSG, F              ; Move to next character in string
        movlw       .33                 ; Check if the whole message has been sent
        subwf       MSG, W              ; "Message length = 33"
        btfss       STATUS, Z
        goto        SEND
Loop
        Goto        Loop                ; When whole message is sent, loop and wait
                                        ; for receiver interrupts.
;*************************************************************************************
; Interrupt Service Routine
IntService
        push
        btfsc       PIR1, RCIF                  ; Check for RX interrupt
        call        RX_Receive
        pop
        retfie
RX_Receive
        bcf         PIR1, RCIF                  ;Pass the value of RCREG to PORTD
;***************************************************************************
; Uncomment the following piece of code if error detection is required. Note that it is
; recommended to detect for serial transmission errors
;***************************************************************************
        ;banksel RCSTA
        ;btfsc       RCSTA, FERR                ;Check for framing error
        ;goto        FramingError
        ;btfsc       RCSTA, OERR                ;Check for Overrun error
        ;goto        OverrunError
        banksel     RCREG
        movf        RCREG, W
        banksel     PORTD
        CALL        Look_TABLE
        movwf       PORTD
        return

Look_TABLE
        ADDWF       PCL , 1
        RETLW       B'11000000'         ;'0'
        RETLW       B'11111001'         ;'1'
        RETLW       B'10100100'         ;'2'
        RETLW       B'10110000'         ;'3'
        RETLW       B'10011001'         ;'4'
        RETLW       B'10010010'         ;'5'
        RETLW       B'10000010'         ;'6'
        RETLW       B'11111000'         ;'7'
        RETLW       B'10000000'         ;'8'
        RETLW       B'10010000'         ;'9'
```

```
Message
            addwf       PCL, F
            retlw       A'M'
            retlw       A'a'
            retlw       A'c'
            retlw       A'h'
            retlw       A'i'
            retlw       A'n'
            retlw       A'e'
            retlw       A' '
            retlw       A'r'
            retlw       A'e'
            retlw       A'a'
            retlw       A'd'
            retlw       A'y'
            retlw       A' '
            retlw       A't'
            retlw       A'o'
            retlw       A' '
            retlw       A'r'
            retlw       A'e'
            retlw       A'c'
            retlw       A'e'
            retlw       A'i'
            retlw       A'v'
            retlw       A'e'
            retlw       A' '
            retlw       A'c'
            retlw       A'o'
            retlw       A'm'
            retlw       A'm'
            retlw       A'a'
            retlw       A'n'
            retlw       A'd'
            retlw       A's'
            END
;************************************************************************************
```

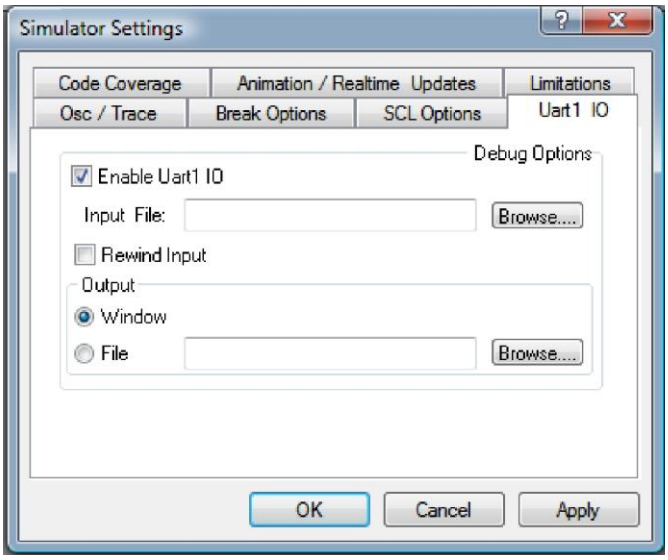
CODE TESTING

At first glance, you might think that you cannot test your code unless you have a physical control PC and a machine at home!! Surely this is not feasible. Therefore we will now introduce you to testing USART serial communication in MPLAB IDE.

MPLAB TRANSMITTER TESTING

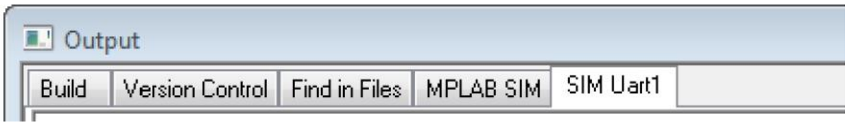
After Building your project in MPLAB do the following procedure:

1. *Debugger → Select Tool → MPLAB SIM*
2. *Debugger → Settings → Uart1 IO*
3. *The following screen will show up:*
4. *Select Enable Uart1 IO*
5. *Select the output to be shown in Window*
6. *Click Ok*

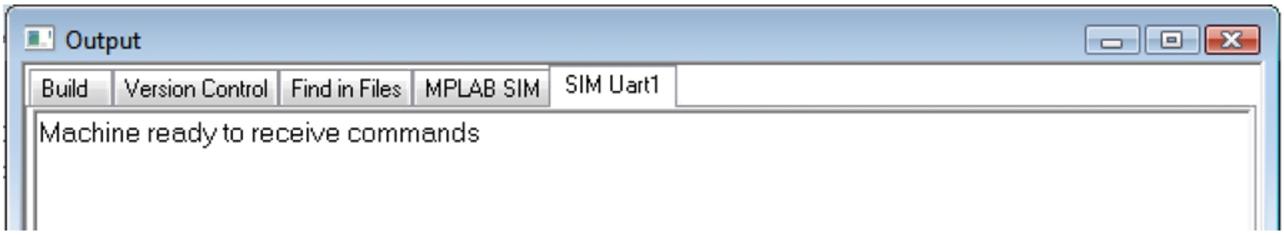


Now, if the output window is not already shown, go to View → Output

Notice that a new tab (SIM Uart1) has shown up as shown below:



Now run the program, you will see that the message has appeared in the Uart1 IO window which we have already enabled. See screenshot below:

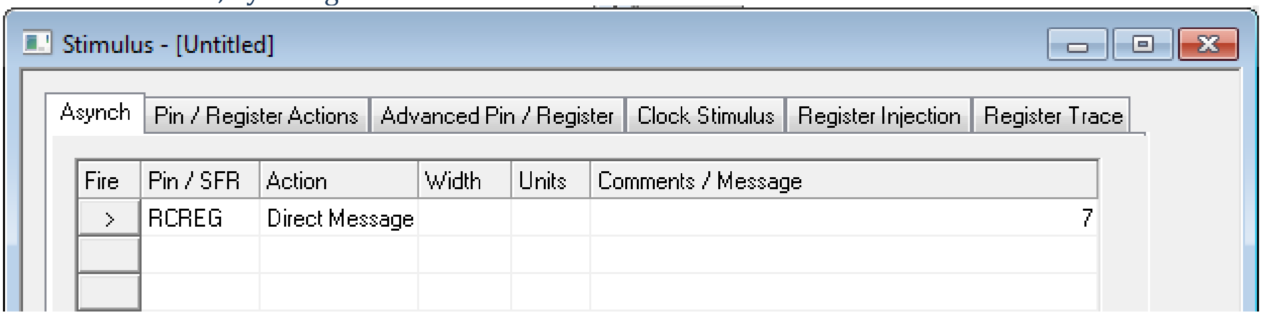


MPLAB RECEIVER TESTING

We will test the receiver the same way we used to test for external inputs: using stimulus.

The procedure will be revisited here again:

1. Debugger → Stimulus →New Workbook
2. In the *Async* tab choose ***RCREG***, and the action as ***Direct Message,*** in the Message field type in the character you wish to send.
3. Press fire, by doing so the character *"7"* will be received in RCREG



4. Place a break point at instruction <u>**goto IntService .**</u>
3. Since the received character is displayed on 7 segment display which are connected to PORTD, use the watch window, check if "11111000" has been actually sent to PORTD

IN-LAB TESTING PHASE 2

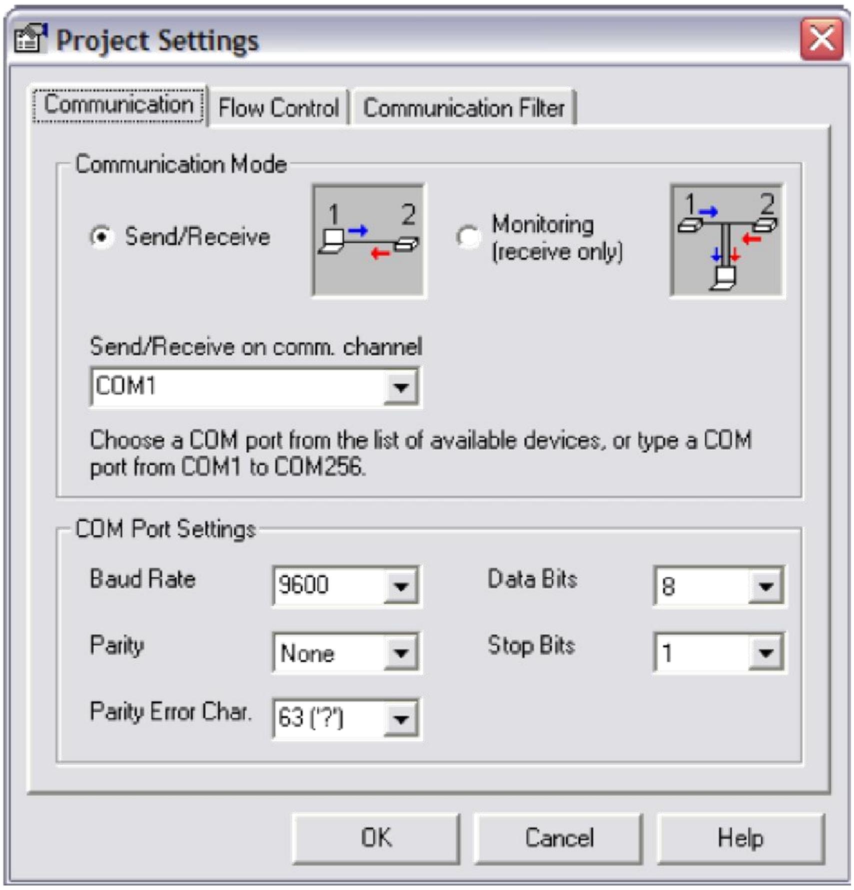
✓ <u>**TRANSMITTER TESTING**</u>

- Double click on the *"Docklight"* program icon found on the desktop.

  The *"Docklight"* program is a tool through which we can establish serial communication between two PC's or a PC with other devices. We can send, receive and view data in different formats: ASCII, hex and decimal.

- Press Ok to the message that appears then ***"Start with a blank project"***
- In *"Docklight", go to Tools → Project Settings*

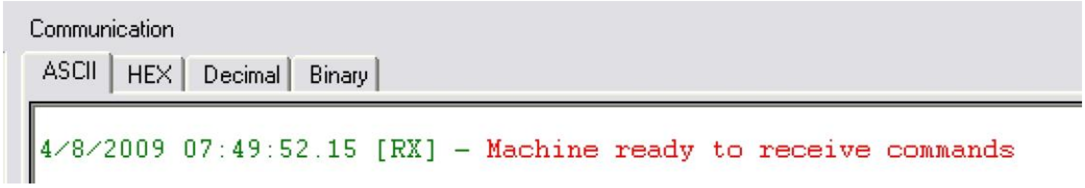
A window will appear through which we can set the communication session settings, set the com port used to COM1, configure the baud rate speed to match that used by the other device/PC, set number of stop bits used and so on. Check the settings we will use in this experiment as seen below:



- Press Ok to save your settings.
- Now we have to start the communication session between the PC and the kit, press the **play** button ▶ or press **F5** to open communication port.

- Download the program to the PIC16F877A;
- Connect the kit to the PC through a serial RS232 cable. Be sure to connect the cable to COM1 of the PC. In this scenario, the kit will act as the machine and the PC will act as the control computer which will receive machine status and send commands.

- Now the PC (control room) is configured properly to receive status and send commands
- Switch back to *"Docklight",* make sure that the window format is ASCII; you will be able to read the message which has been sent by the PIC to the PC. WOW ☺

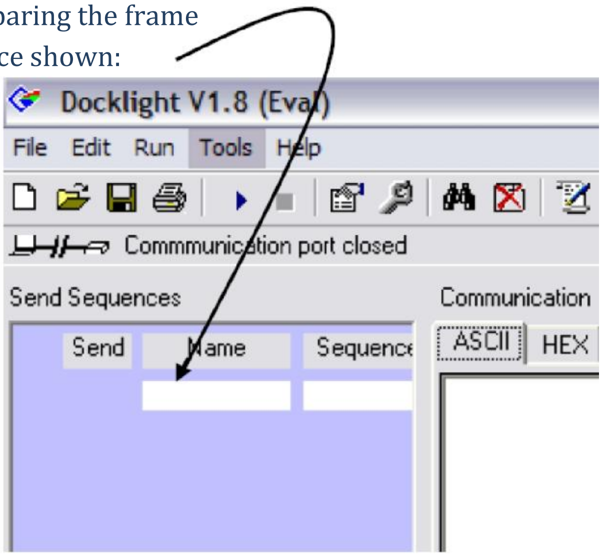


RECEIVER TESTING

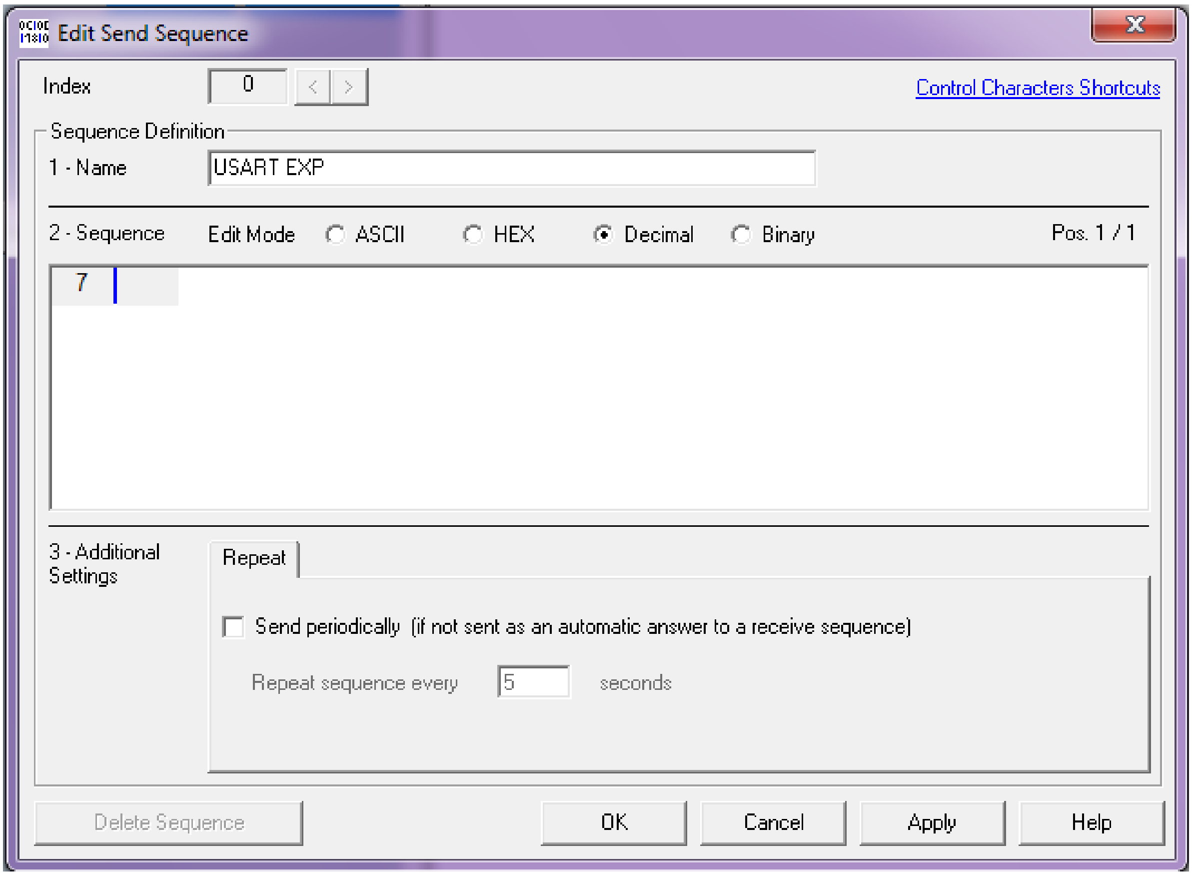
To send data to the kit (machine) start with the following:

We will start with preparing the frame

- Double click on the space shown:

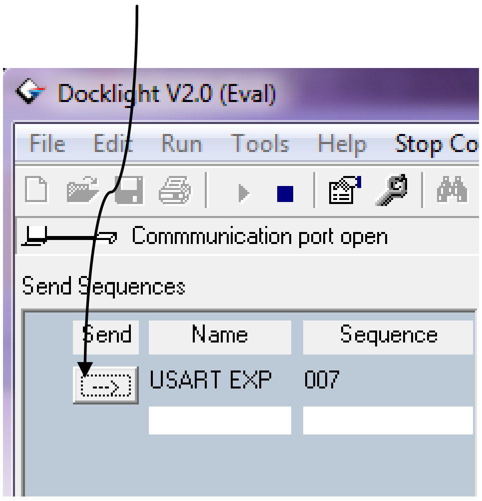


- Give the data sequence you want to send a name (optional)
- Choose the data format you want to see (decimal)
- Fill in the data you want to send then click ***OK***



- So far we have not yet transmitted the data to do so

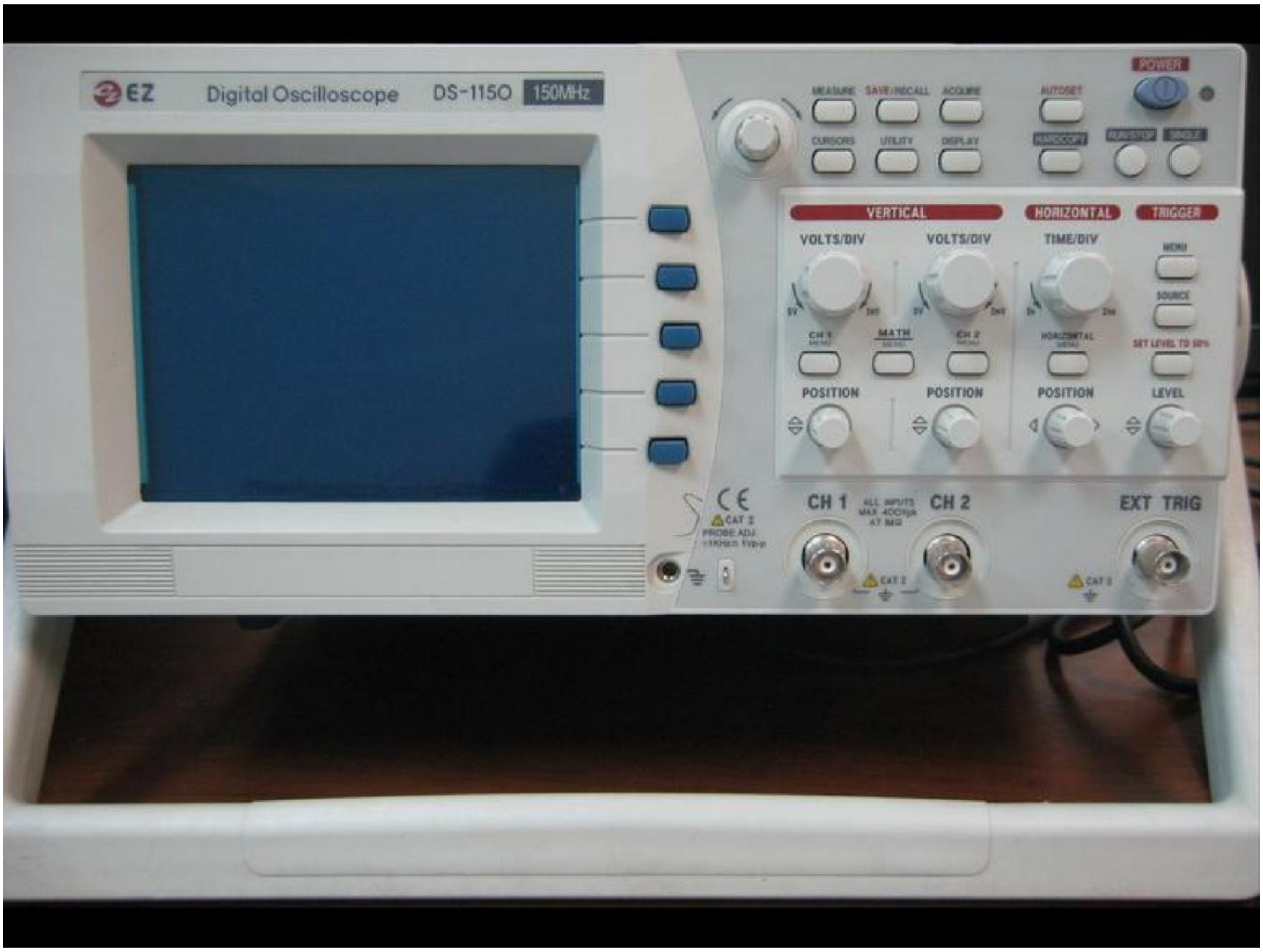
Click on the fire button



Capturing the frame sent/received by the USART using a Digital Oscilloscope

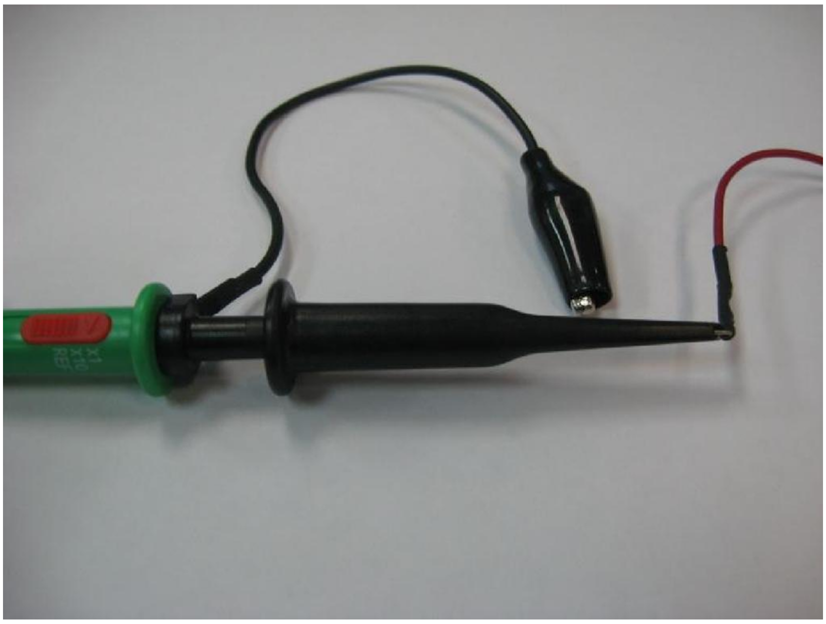
Digital oscilloscopes provide an easy way to capture signals using the “AutoSet” function provided with most models. However, this function is not feasible for use with non periodic signals especially those that are at high frequencies which is the case in this experiment; we are to capture and view a transmitted or received frame at baud rates of 9600 or more. Even using manual setting and pressing the “Stop” button will not be that easy as transmission and reception speed increases. Therefore, we are to use the trigger function which modern oscilloscopes offer.

The trigger event is usually the input waveform reaching some user-specified threshold voltage in the specified direction (going positive or going negative). Trigger circuits allow the display of non-periodic signals such as single pulses or pulses that don't recur at a fixed rate.

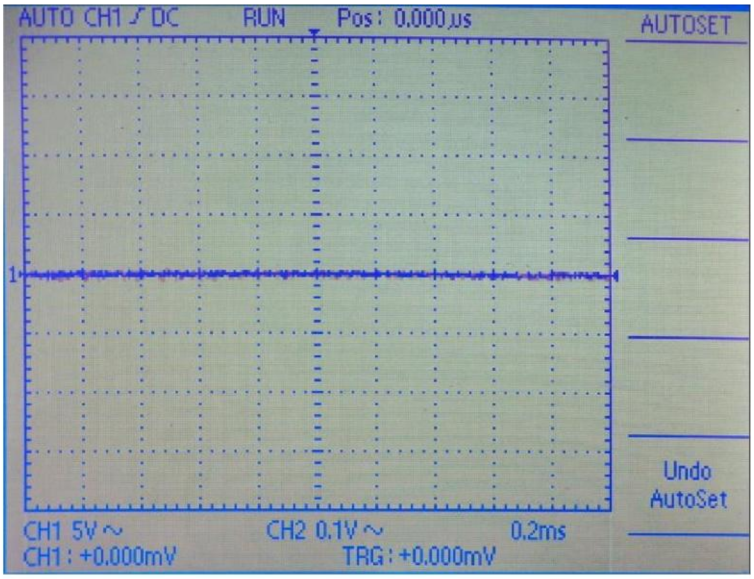


The DS1150 Digital Oscilloscope

1. In this experiment, connect the oscilloscope probe to CH1 and use the hook at the other end to connect to RC7 pin (Receiver) through a wire. Connect the probe GND to that of the Mechatronics board (Optional). – See figure below!

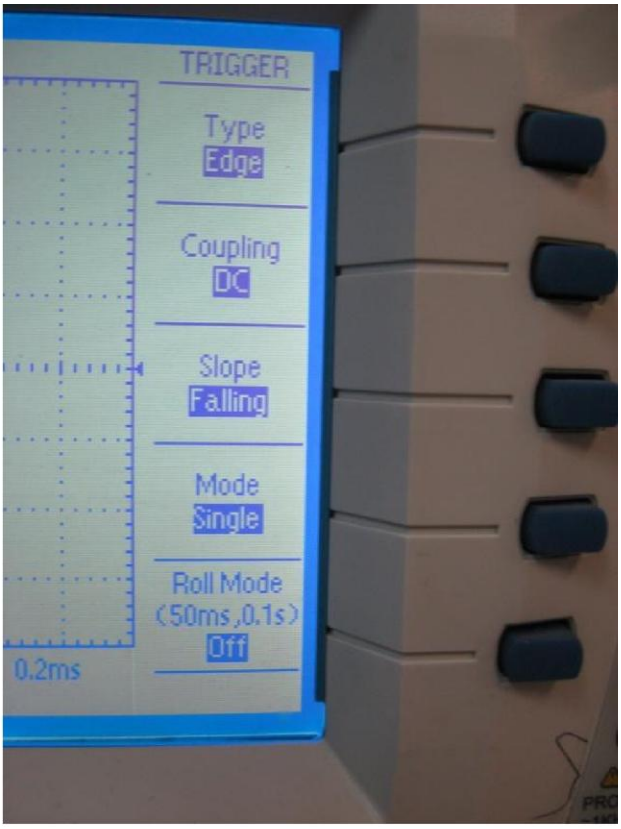


2. Make sure that the orange slider of the oscillator’s probe is at X1 option.
3. Power on the oscilloscope
4. Press Autoset (if the probe is not connected to the circuit, this resets the oscilloscope)
5. Set Voltage/Div value on CH1 to 5 Volts using the knob.
6. Set the time division to 0.2 ms (remember that we have calculated above that the whole frame will take 1.041 ms to be sent, therefore we need a smaller time division in order to see the whole frame fit on the oscilloscope screen).



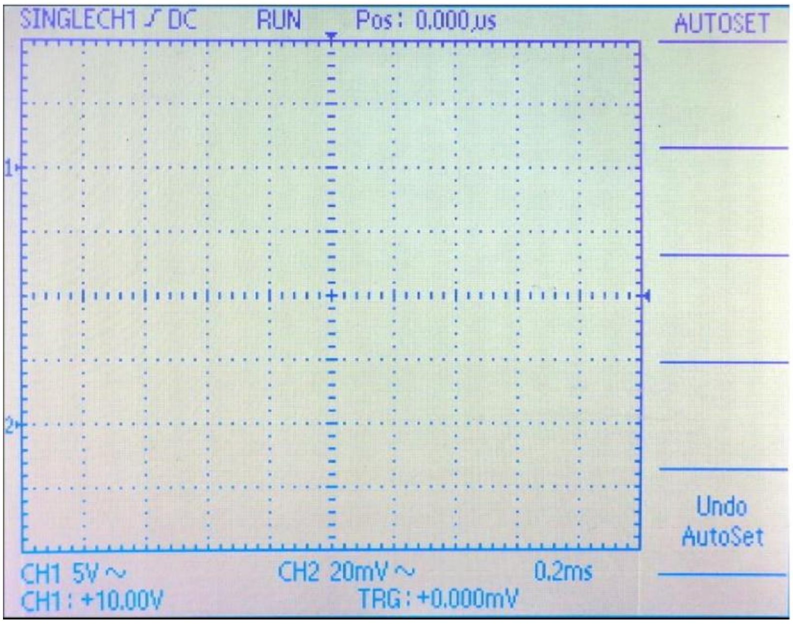
Oscilloscope Screenshot after settings

7. On the right side of the oscilloscope you will see a set of trigger buttons, press the "Source" button as many times until you see that the trigger is on CH1 (Upper left corner of the screen)
8. Press the trigger's "Menu" button then select the following options using the 5 blue buttons to the right of the oscilloscope's screen:



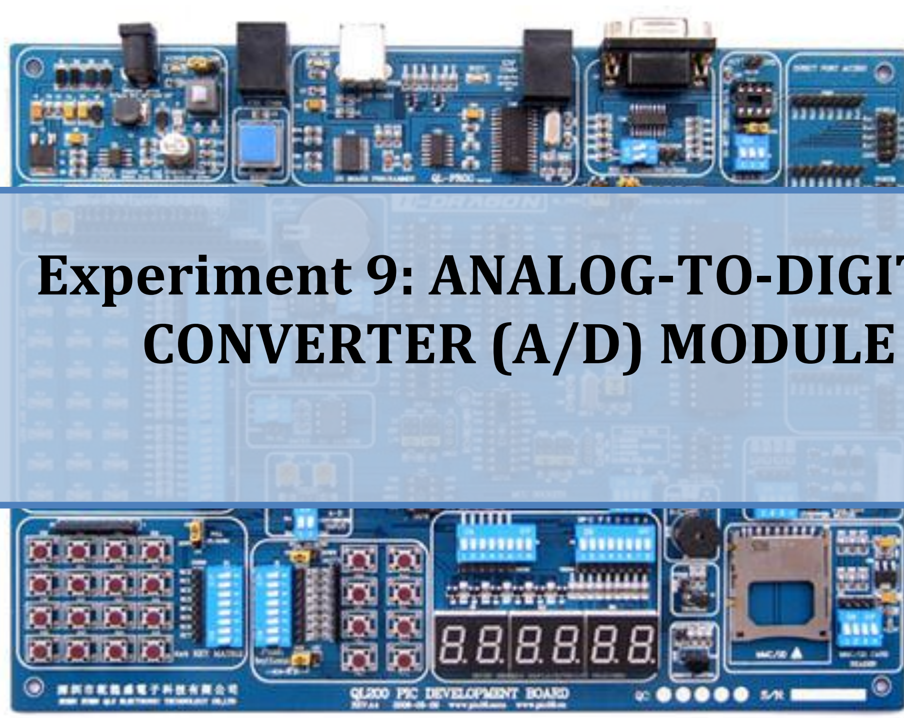
   - Select **CH1** (other options include CH2, Line and EXTernal), you will see your selection at the upper left corner of the oscilloscope's screen.
   - Change the coupling to "**DC**",
   - Edge to "**Falling**" (since we are to detect the beginning of the frame, which is a transition from idle state to start bit state (Logic 1 to Logic 0 at pin RC7)
   - Finally set the Mode to "**Single**" since we are to detect only one frame.

9. Make sure that all your connections are correct and firmly fixed, review your oscilloscope and Docklight settings, after which use the Docklight program to send the hex value 0x65 as an example.
10. The frame should now appear on the screen, draw it here:



11. Now, you will notice that the screen has frozen to show this frame, to view other frames, press the STOP/RUN button, now the oscilloscope is ready to receive and display new frames.

University of Jordan
Faculty of Engineering and Technology
Department of Computer Engineering
Embedded Systems Laboratory 0907334



9

Experiment 9: ANALOG-TO-DIGITAL CONVERTER (A/D) MODULE

Objectives

- To familiarize you with the built-in A/D hardware module.

Pre-lab requirements

- Review the PIC16F877A datasheet section on the AD module.

Appendix A quickly reviews the AD module

Overview

An analog to digital converter converts analog voltages to digital information that can be used by a computer. In almost in all digital systems, there is a frequent need to convert analog signals generated by peripheral devices such as microphones, sensors, and etc. into digital values that can be stored and processed. As an example, temperature and brightness are changing continuously. This experiment will focus on A/D conversion by using the PIC16F877A Analog-To-Digital Converter.

The idea behind the code

Select RAO as input connected to potentiometer, get the result of a A/D conversion, convert the result into the BCD format and finally the result (the only low 8 bits) will be displayed on three 7-segment displays, The 7 segments display will use Time Division Multiplexing to display a 3-digit values.

A Detailed View of the Interworking of the System

Based on the above discussion, we will further elaborate how this system works.

1. Initially, the system should be initialized as follows:
    - We need to connect an analogue signal to the PIC, we shall use either one of PORTA or PORTE, since both offer analogue input interfacing to the PIC. We will specify which PORT and which exact pin of the port to be used as analogue or digital through the use of the **ADCON1** register. In this experiment we chose RA0 as the analogue input (corresponding to channel 0 "AN0" of the AD module)
    - We will configure the AD module as follows, power on the module (set ADON), and choose the analogue channel 0 "AN0" as the analogue input of the AD module by setting CH2, CH1 and CH0 as zeros. We will set the voltage references to be between 0 and 5 volts (why?) and finally the result is to be right justified, that is the lower 8-bits will reside in ADRESL and the higher 2 bits will reside in ADRESH. In this program, we will choose to ignore ADRESL and only deal with the 8-bit digitized value to simplify program development.
    - We chose a conversion speed of Fosc/8, therefore ADCON1 will have the value of 0x8E
    - We implemented the code such that the main functionality is to convert analogue signals into digital ones and save them into ADRESL in a continuous fashion such that we will always have updated and recent values of the potentiometer, this is the code of the main subroutine will have all other actions: CHANGE _To_BCD ,this subroutine is used to convert the result of the conversion into BCD values (Units , Tens , Hundreds), then display the result on the 7 segment display , Time Division Multiplexing used to display a 3-digit values(Units , Tens , Hundreds).

2. As stated above, the main subroutine is to continuously update ADRESL register with a recent digitized value of the potentiometer. The routine starts by starting the conversion process (bsf ADCON0, GO), the value of ADRESL is not read until we are sure that the conversion process has truly finished. This is done through polling the ADIF flag (remember that we have not enabled the interrupt for AD, yet the flags of interrupts are set and cleared no matter whether they were enabled or not, this is why polling is possible). When the conversion is finished, the value of ADRESL is copied into TEMP register in order to display it on the 7 segment display!

The steps should be followed for doing an A/D Conversion:

- ***Port configuration***

  The I/Os pin should be configured as analog by setting the associated TRIS and PCFG3:PCFG0 bits.

- ***ADC module configuration***

  - Channel selection

    The CHS bits of the ADCON0 register.
  - ADC voltage reference selection

    The PCFG bits of the ADCON1 register.
  - Results formatting

    The ADFM bit of the ADCON1 register controls the output format.
  - Turn on ADC module

    To enable the ADC module, the ADON bit of the ADCON0 register must be set to a '1'.
  - ADC conversion clock source

    The source of the conversion clock is software selectable via the ADCS bits of the ADCON1 and ADCON0 registers.

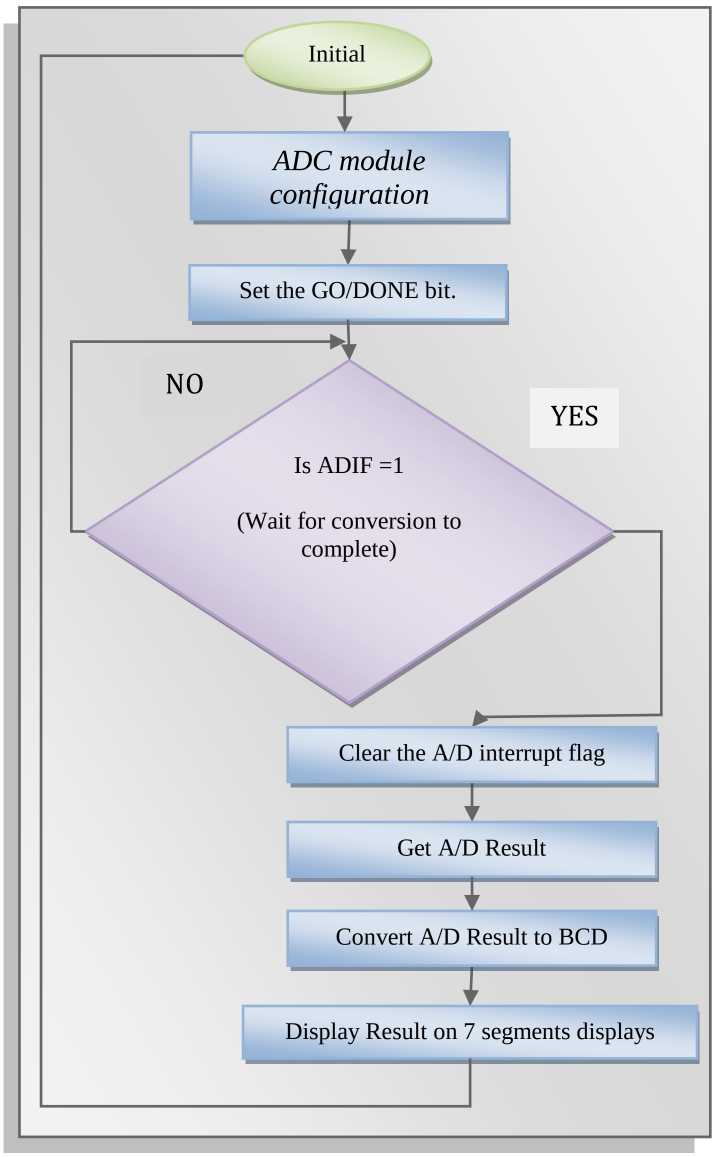
- ***Interrupt control***

  The ADC module allows for the ability to generate an interrupt upon completion of an Analog-to-Digital conversion, but we have chosen to use the ADC without interrupts and use polling instead.

;;;Main Subroutine;;;

This subroutine shows theA/D Conversion Procedure.

Start conversion by setting the GO/DONE bit. Poll the AD interrupt flag ADIF (interrupts disabled) to check whether conversion has finished or not. Clear the ADC interrupts flag (required). Finally Read ADC Result found in ADRESH and/or ADRESL. Convert Result into BCD Format and display it on the 7 segments displays.




```
Main
 MOVLW    8EH            ;A/D data right justified
 MOVWF    ADCON1         ; RA0 is analogue input
 Banksel  PORTA          ;BANK 0
 MOVLW    41H            ;A/D enabled
 MOVWF    ADCON0         ;select CLOCK is fosc/,
 CALL     DELAY
 BSF      ADCON0,GO      ;startup ADC divert
WAIT
 BTFSS    PIR1,ADIF      ;Is the convert have finished?
 GOTO     WAIT           ; wait for the convert finished
 bcf      PIR1, ADIF     ; Clear the A/D flag
 Banksel  TRISA
 MOVF     ADRESL,W       ;read the result of convert
 Banksel  PORTA
 MOVWF    TEMP           ; keep in temporary register
 CALL     CHANGE_To_BCD  ; call result convert subr.
 CALL     DELAY
 CALL     DISPLAY        ; call display subroutine
 CALL     DELAY
 GOTO     Initial        ; Do it again
```

```
;****************************************************
;Code Function:Select RAO as input connected to potentiometer,
;get the result of a A/D conversion ,convert the result into the BCD format
; and finally the result (the only low 8 bits) will be displayed on 7-segment displays.

#INCLUDE<P16F877a.INC>

TEMP        EQU   20H    ;temporary register
hundreds    EQU   21H    ;the hundred bit of convert result
tens        EQU   22H    ;the ten bit of convert result
units       EQU   23H    ;the ones bit of convert result
;****************************************************
        ORG     00H
        NOP
        GOTO    Initial

;******************Initial subroutine****************************
Initial
                CLRF     hundreds
                CLRF     tens
                CLRF     units
                Banksel  TRISA                  ;select bank 1
                MOVLW    01H                    ;PORTA bit Number0 is INPUT
                MOVWF    TRISA
                CLRF     TRISD                  ;All of the PORTD bits are outputs
;**********************MAIN program*********************
Main
                MOVLW    8EH            ;A/D data right justified
                MOVWF    ADCON1         ;only select RA0 as ADC PORT,the rest are data PORT
                Banksel  PORTA          ;BANK 0
                MOVLW    41H
                MOVWF    ADCON0         ;select CLOCK is fosc/8,A/D enabled
                CALL     DELAY          ;call delay program,ensure enough time to sampling
                BSF      ADCON0,GO      ;startup ADC divert
WAIT
                BTFSS    PIR1,ADIF      ;is the convert have finished?
                GOTO     WAIT           ;wait for the convert finished
                Bcf      PIR1, ADIF     ; Clear the A/D flag
                Banksel  TRISA
                MOVF     ADRESL,W       ;read the result of convert
                Banksel  PORTA
                MOVWF    TEMP           ;keep Result in temporary register
                CALL     CHANGE_To_BCD  ;call result convert subroutine
                CALL     DELAY
                CALL     DISPLAY        ;call display subroutine
                CALL     DELAY
                GOTO     Initial        ;Do it again
;***********************Convert subroutine*******************
CHANGE_To_BCD
gen_hunds
                MOVLW    .100           ;sub 100,result keep in W
                SUBWF    TEMP,0
                BTFSS    STATUS,C       ;judge if the result biger than 100
                GOTO     gen_tens       ;no,get the ten bit result
                MOVWF    TEMP           ;yes,result keep in TEMP
                INCF     hundreds,1     ;hundred bit add 1
                GOTO     gen_hunds      ;continue to get hundred bit result
gen_tens
                MOVLW    .10            ;sub 10,result keep in W
                SUBWF    TEMP,0
                BTFSS    STATUS,C       ;judge if the result biger than 10
                GOTO     gen_ones       ;no,get the Entries bit result
                MOVWF    TEMP           ;yes,result keep in TEMP
                INCF     tens,1         ;ten bit add 1
                GOTO     gen_tens       ;turn  to continue get ten bit
```

```
gen_ones
                MOVF      TEMP,W
                MOVWF     units               ;the value of Entries bit
                RETURN

;*************************Display subroutine*******************
        DISPLAY
         MOVF     hundreds,W                  ;display Hundreds bit
         CALL     TABLE
         MOVWF    PORTD
         BCF      PORTA,3
         CALL     DELAY
         CALL     DELAY
         BSF      PORTA,3

         MOVF     tens,W                      ;display Tens bit
         CALL     TABLE
         MOVWF    PORTD
         BCF      PORTA,4
         CALL     DELAY
         CALL     DELAY
         BSF      PORTA,4

         MOVF     units,W                     ;display Units bit
         CALL     TABLE
         MOVWF    PORTD
         BCF      PORTA,5
         CALL     DELAY
         CALL     DELAY
         BSF      PORTA,5
         RETURN

;*********************************************************
TABLE
                ADDWF   PCL,     1
                RETLW   B'11000000'           ;'0'
                RETLW   B'11111001'           ;'1'
                RETLW   B'10100100'           ;'2'
                RETLW   B'10110000'           ;'3'
                RETLW   B'10011001'           ;'4'
                RETLW   B'10010010'           ;'5'
                RETLW   B'10000010'     ;'6'
                RETLW   B'11111000'           ;'7'
                RETLW   B'10000000'           ;'8'
                RETLW   B'10010000'           ;'9'

;**************************Delay subroutine*********************
DELAY
        MOVLW   0xFF
        MOVWF   TEMP
L1      DECFSZ  TEMP,1
        GOTO    L1
        RETURN

;*********************************************************
 END              ;program end
```

Appendix A

Analog-to-Digital Conversion (ADC)

An analog-to-digital converter, or simply ADC, is a module that is used to convert an analog signal into a digital code. In the real world, most of the signals sensed and processed by humans are analog signals. Analog-to-digital conversion is the primary means by which analog signals are converted into digital data that can be processed by Microcontroller for various purposes.

Sensors signals is an analog quantity, and digital systems often use signals to implement measurement, control, and protection functions so it is the necessary to convert the analog signal to digital information.

There's generally a lot of confusion about using the A/D inputs, but it's actually really very simple - it's just a question of Extraction the information you need out of the datasheets.

There are four main registers associated with using the analogue inputs; these are summarized in the following table:

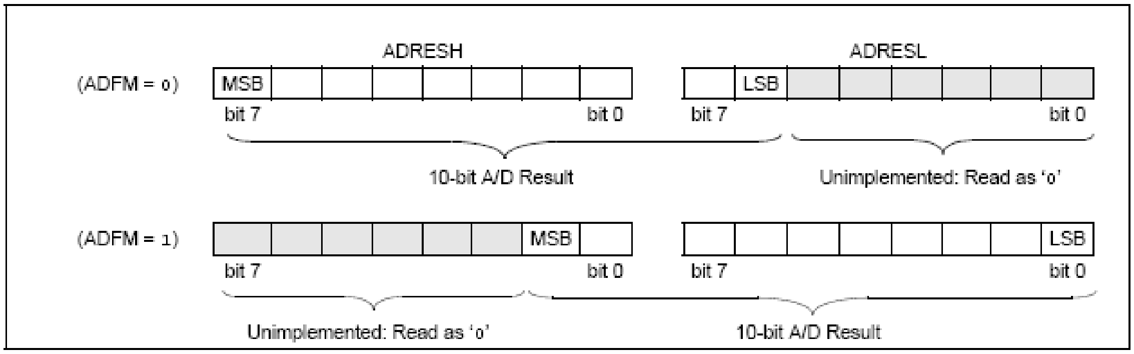
Main registers used for Analog-to-Digital Conversion.

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ADRESH	A/D Result Register - High Byte							
ADRESL	A/D Result Register - Low Byte							
ADCON0	ADCS1	ADCS0	CHS2	CHS1	CHS0	GO/DONE	-	ADON
ADCON1	ADFM	ADCS2	-	-	PCFG3	PCFG2	PCFG1	PCFG0

- **ADCON0** and **ADCON1** are the registers that control the A/D conversation process.
- **ADRESH** and **ADRESL** are the registers that return the 10-bit result of the analogue to digital conversion, the only slightly tricky thing about them is that they are in different memory banks.

RESULT FORMATTING:

The 10-bit A/D conversion result can be supplied in two formats, left justified or right justified. The desired formatting is chosen by sitting the ADFM bit in the ADCON0 register.



ADCON0 Details

ADON (bit 0), turns the A/D On (when = 1) or off (when = 0), thus saving the power it consumes.

GO/DONE (bit 2), this bit has a dual function, the first is that by setting the bit it initiates the start of the analogue to digital conversion process, the second is that when the bit is automatically cleared when the conversion is complete, it can be polled to check if conversion has ended before initiating a subsequent conversion.

CHS2, **CHS1** and **CHS0 (bits 3 - 5)**, the channel selection bits, choose one channel among the available eight AD analogue channels and specify which one is to be used as an input for the AD module for digitization. Be careful that the first five channels AN0-AN4 map to pins (RA0-RA3, RA5). Further notice that AN4 uses digital pin RA5, not RA4 as you would expect. And the remaining three channels AN5-AN7 map to pins (RE0-RE2). See adjacent figure.

CHS2	CHS1	CHS0	Channel	Pin
0	0	0	Channel0	RA0/AN0
0	0	1	Channel1	RA1/AN1
0	1	0	Channel2	RA2/AN2
0	1	1	Channel3	RA3/AN3
1	0	0	Channel4	RA5/AN4
1	0	1	Channel5	RE0/AN5
1	1	0	Channel6	RE1/AN6
1	1	1	Channel7	RE2/AN7

ADCS1 and ADCS0 (bits 6 - 7): A/D Conversion Clock Select bits (see **ADCS2)**

ADCON1 Details

ADFM (bit 7), the **Result Format Selection Bit**, selects if the output is Right Justified (bit set) or Left Justified (bit cleared). For full digitization precision, the whole 10 bits are to be used.

ADCS2 (bit 6), which set the clock frequency used for the analogue to digital conversion, this clock is divided down from the system clock (or can use an internal oscillator), bit 4 and bit 5 Unimplemented: Read as '0'.

ADCON1 ADCS2	ADCON0 <ADCS1:ADCS0>		A/D Conversion Clock Select bits.
0	0	0	Fosc/2
0	0	1	Fosc/8
0	1	0	FOsc/32
X	1	1	FRC (clock derived from a dedicated Internal oscillator = 500 kHz max.)
1	0	0	Fosc/4
1	0	1	Fosc/16
1	1	0	Fosc/64

PCFG3:PCFG0 (bit 3:0): A/D Port Configuration Control bits

Example

If we make ADCON1 = 0x80, then we have 8 analog channels, and Vref+ = VDD, and Vref- = Vss.

PCFG <3:0>	AN7	AN6	AN5	AN4	AN3	AN2	AN1	AN0	VREF+	VREF-	C / R
0000	A	A	A	A	A	A	A	A	VDD	VSS	8 / 0
0001	A	A	A	A	VREF+	A	A	A	AN3	VSS	7 / 1
0010	D	D	D	A	A	A	A	A	VDD	VSS	5 / 0
0011	D	D	D	A	VREF+	A	A	A	AN3	VSS	4 / 1
0100	D	D	D	D	A	D	A	A	VDD	VSS	3 / 0
0101	D	D	D	D	VREF+	D	A	A	AN3	VSS	2 / 1
011x	D	D	D	D	D	D	D	D	—	—	0 / 0
1000	A	A	A	A	VREF+	VREF-	A	A	AN3	AN2	6 / 2
1001	D	D	A	A	A	A	A	A	VDD	VSS	6 / 0
1010	D	D	A	A	VREF+	A	A	A	AN3	VSS	5 / 1
1011	D	D	A	A	VREF+	VREF-	A	A	AN3	AN2	4 / 2
1100	D	D	D	A	VREF+	VREF-	A	A	AN3	AN2	3 / 2
1101	D	D	D	D	VREF+	VREF-	A	A	AN3	AN2	2 / 2
1110	D	D	D	D	D	D	D	A	VDD	VSS	1 / 0
1111	D	D	D	D	VREF+	VREF-	D	A	AN3	AN2	1 / 2

A = Analog input D = Digital I/O
C / R = # of analog input channels / # of A/D voltage references