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.

Create asm file using MPLAB

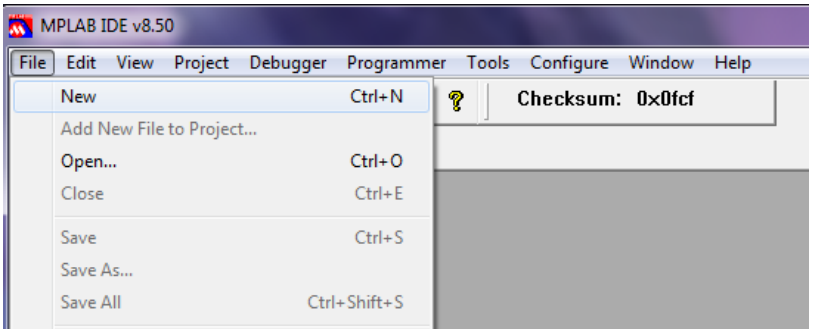
a) Double click on the "**MPLAB**" program icon found on the desktop.

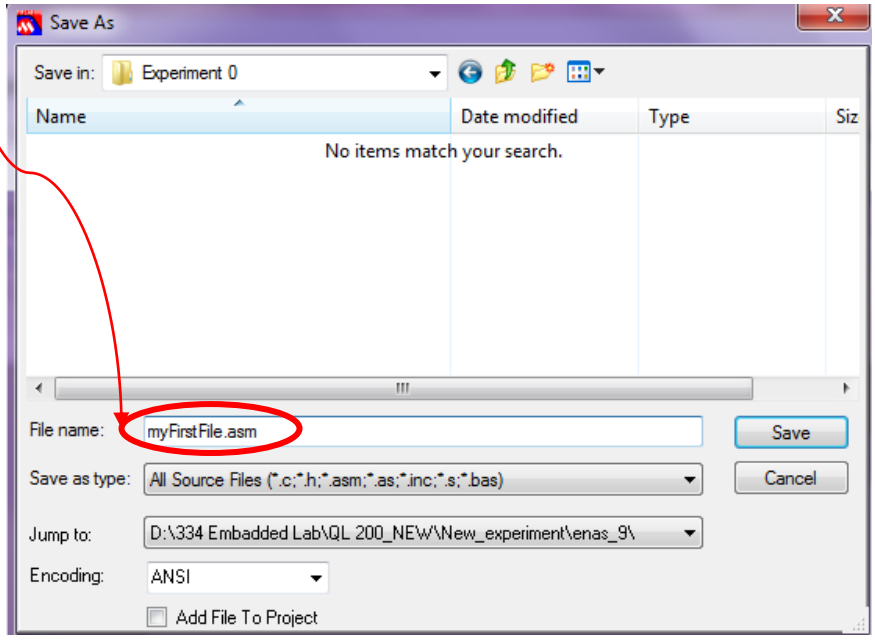
Note: All programs written, simulated and debugged in MPLAB should be stored in files with .asm extension.

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.



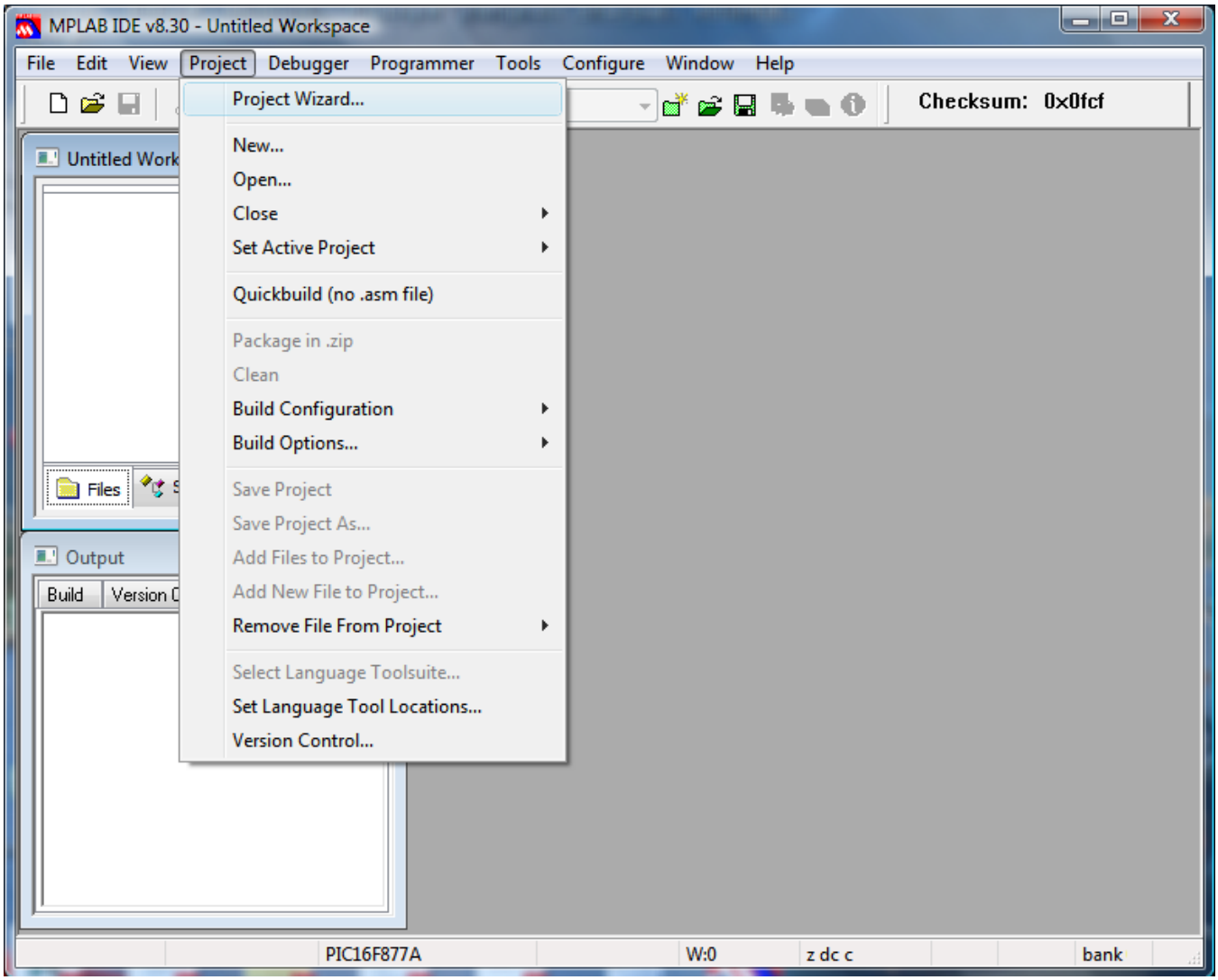


NOTE: All your files should be stored in a short path:

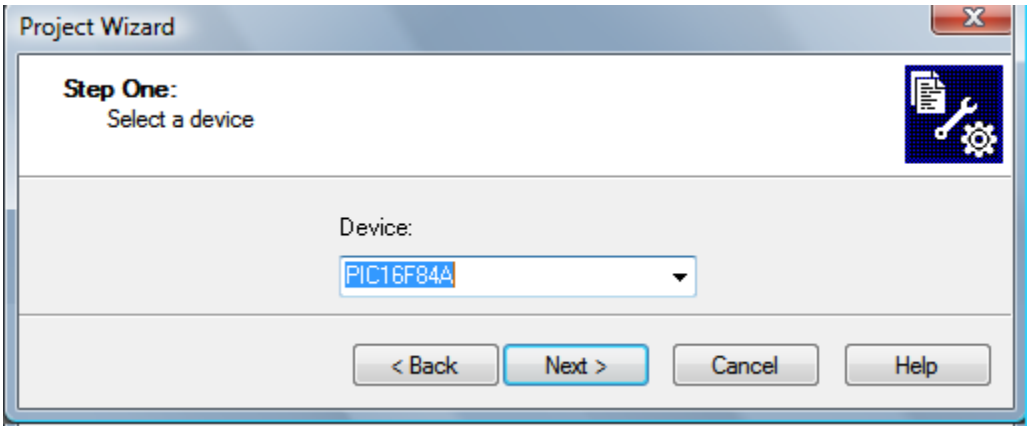
The total number of characters in a path should not exceed 64	Char No.	
C:\ or D:\ or …	3	✓
D:\Embedded\	12	✓
D:\Embedded\Lab	15	✓
D:\Engineer\Year_Three\Summer_Semester\Embedded_Lab\Experiment_1\MyProgram.asm	78	×
Any file on Desktop		×

Create a project in MPLAB by following these simple steps:

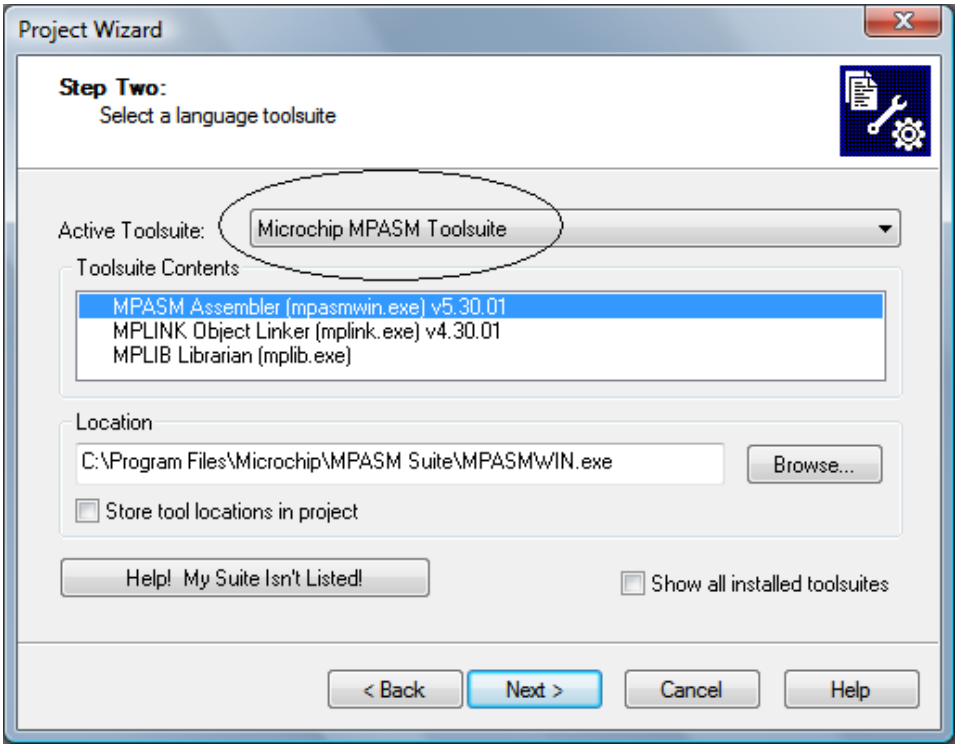
1. Select the Project → Project Wizard menu item → Next



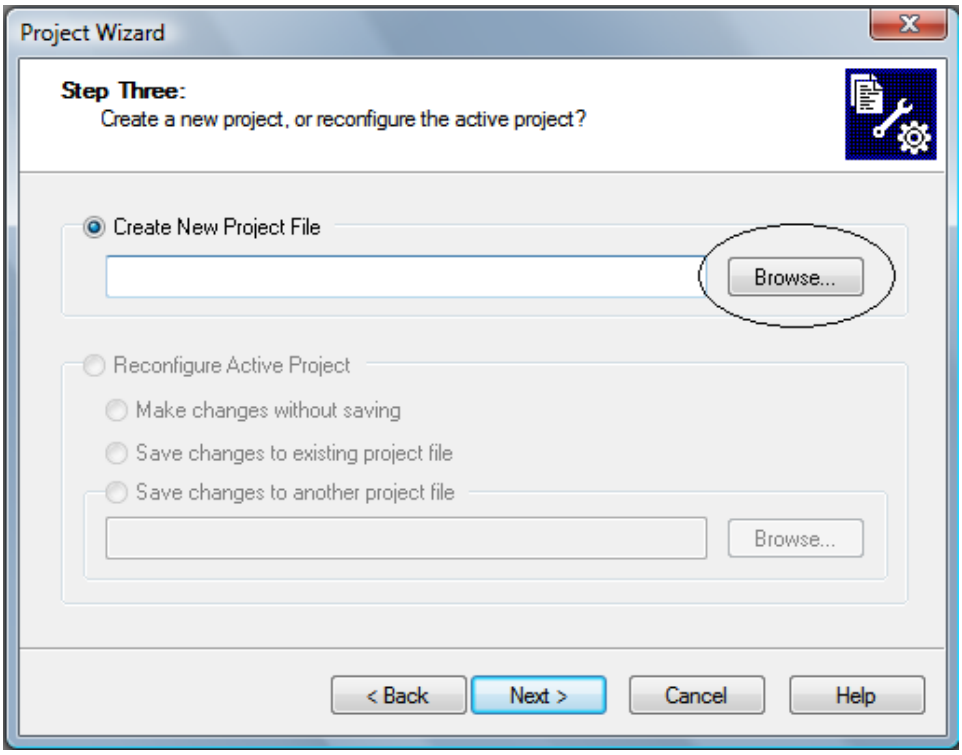
2. In the device selection menu, choose 16F84A (or your target PIC) → Next



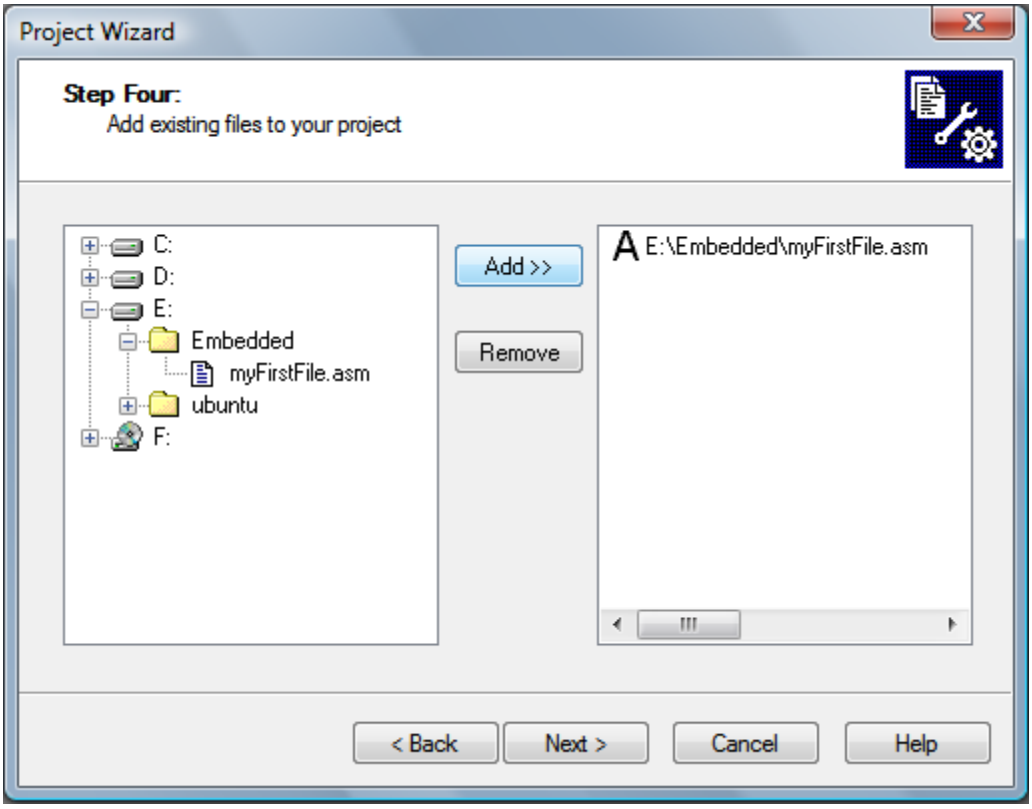
3. In the Active Toolsuite, choose Microchip MPASM Toolsuite → Click next.
   DO NOT CHANGE ANYTHING IN THIS SCREEN



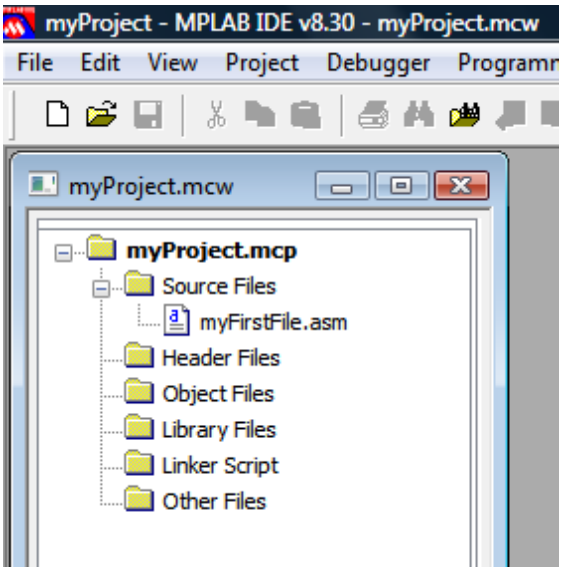
4. Browse to the directory where you saved your ASM file. Give your project a name → Save → Next.



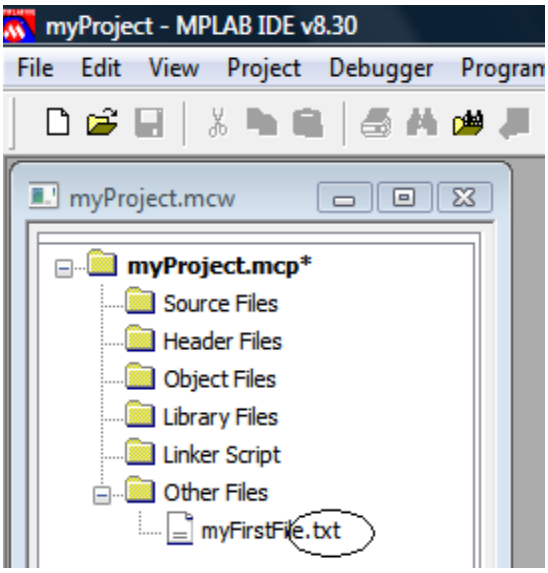
5. If, in Step 4, you navigated correctly to your file destination you should see it in the left pane otherwise choose back and browse to the correct path. When done Click add your file to the project (here: myFirstFile.asm). Make sure that the letter A is beside your file and not any other letter →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



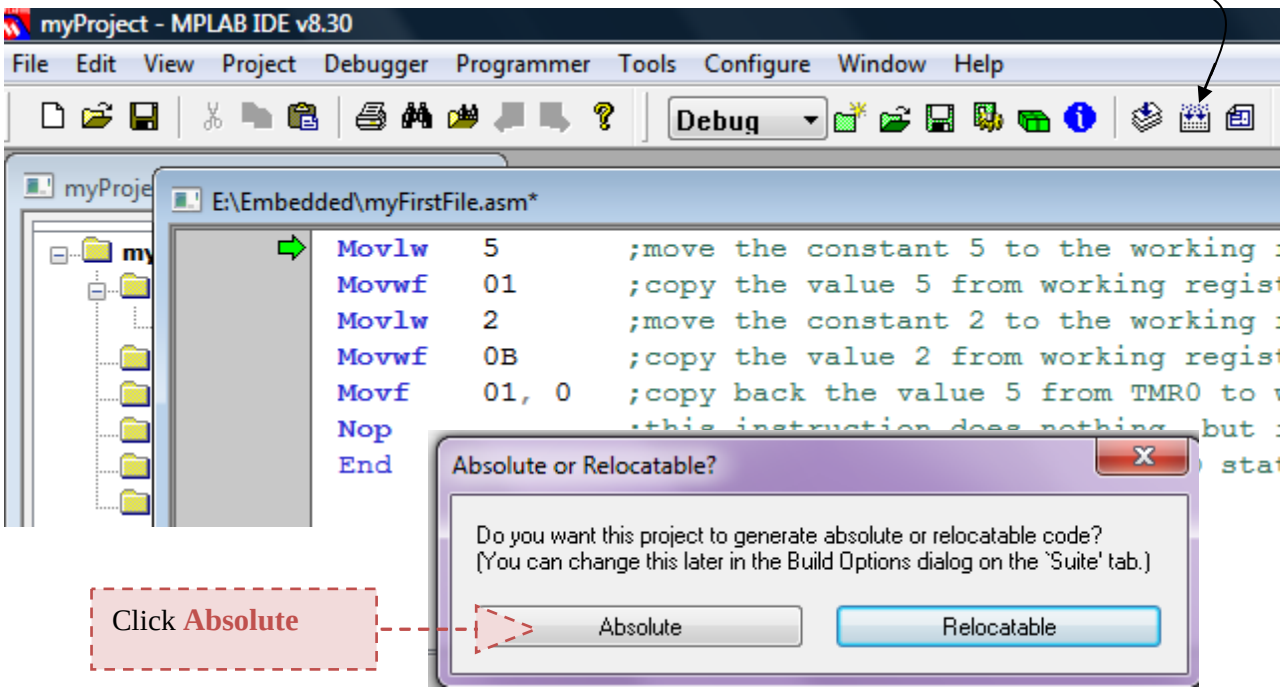
WRONG

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
```

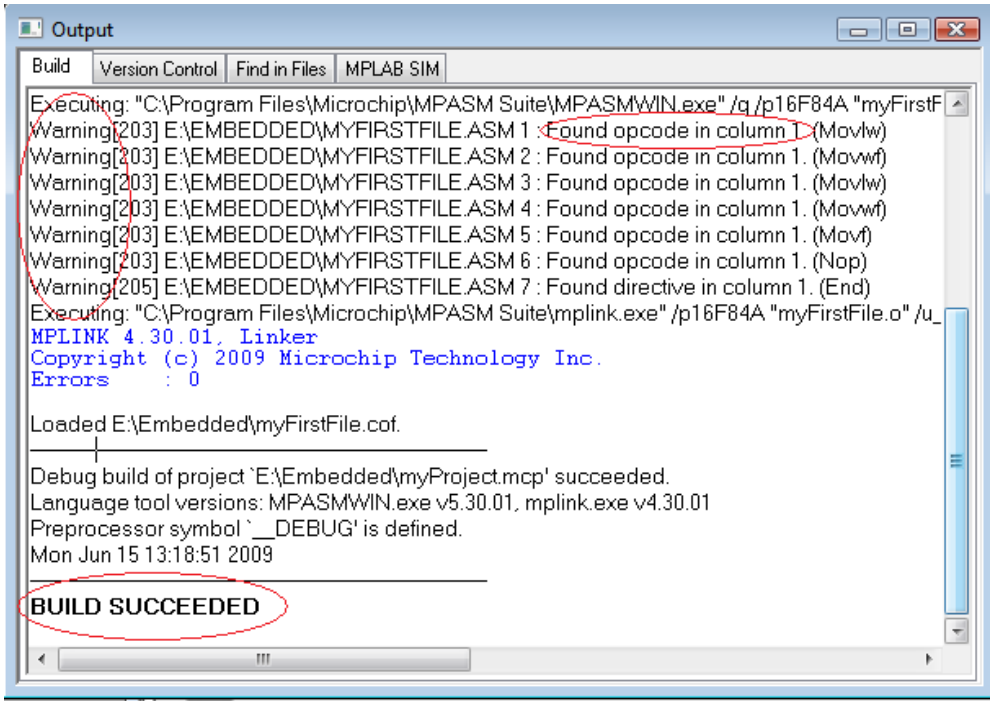
After writing the above instructions we should build the project, do so by pressing ***build***



Click **Absolute**

An output window should show:

BUILD SUCCEEDED

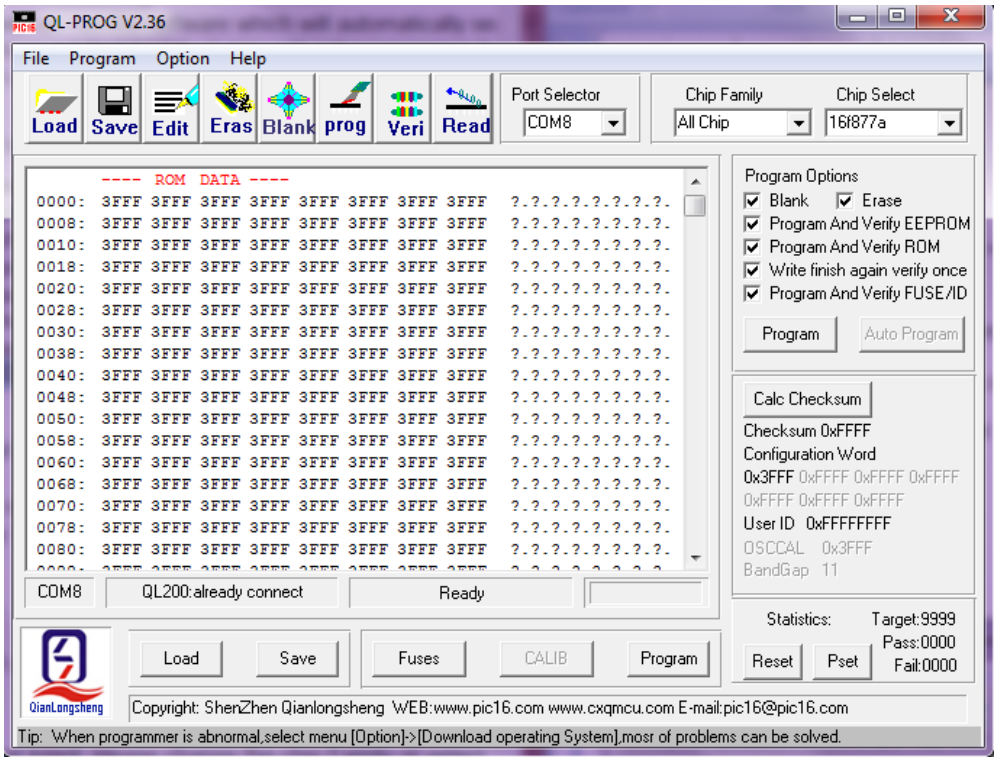


QL-PROG – How to Program

Prepared by Eng. Enas Jaara

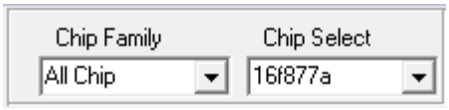
After installation, shortcut of this software will appear on desktop.

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**

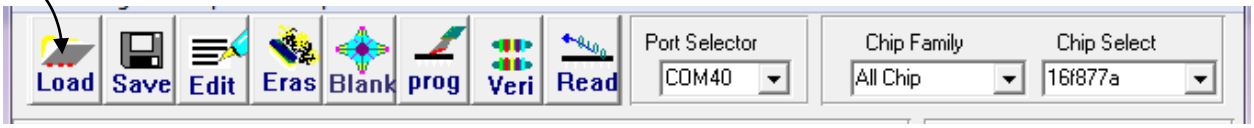
   Choose All Chip from the chip family and choose 16F877A from the chip select



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

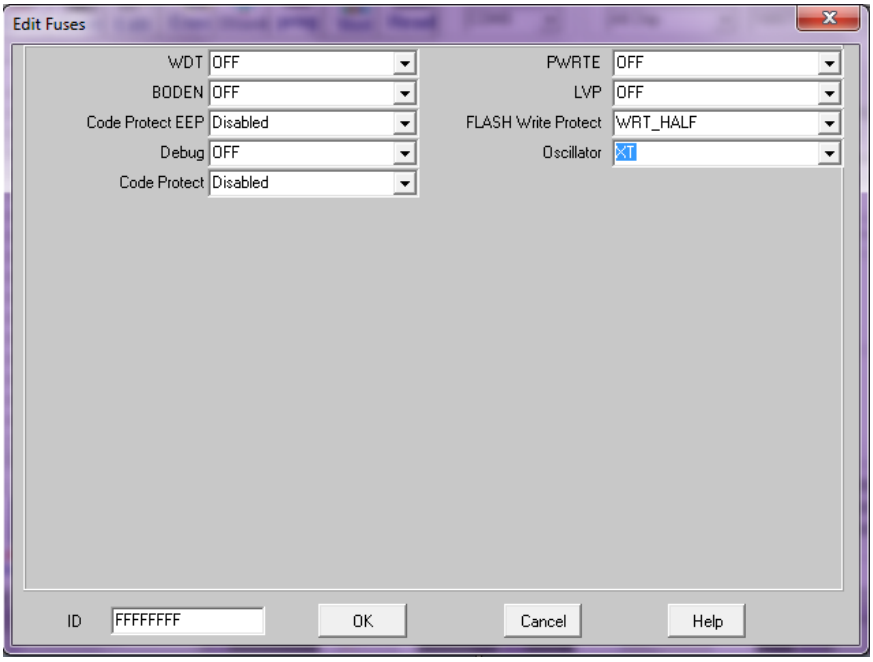
4. **Load File to Program**

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

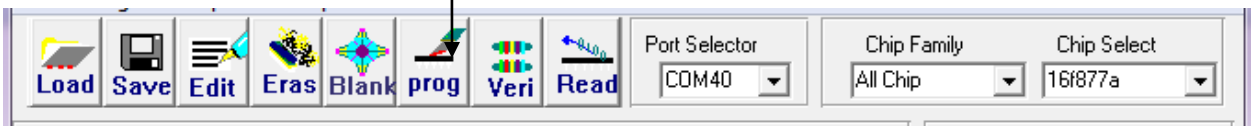
You may set or change the configuration bit of chip by running pressing "**Fuses**" button on software panel. After running the command software, pop-up window to set configuration bit will appear as shown in below diagram. Set the options according to your requirement and click "OK" button.

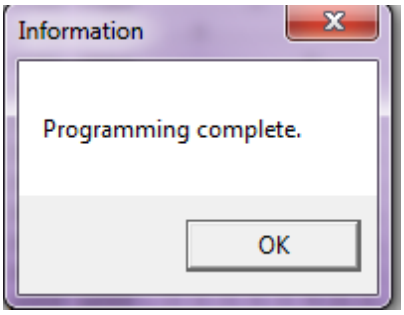


If any of the above option differs, it is because you have chosen the wrong PIC, so go to **chip select** and choose your appropriate PIC.

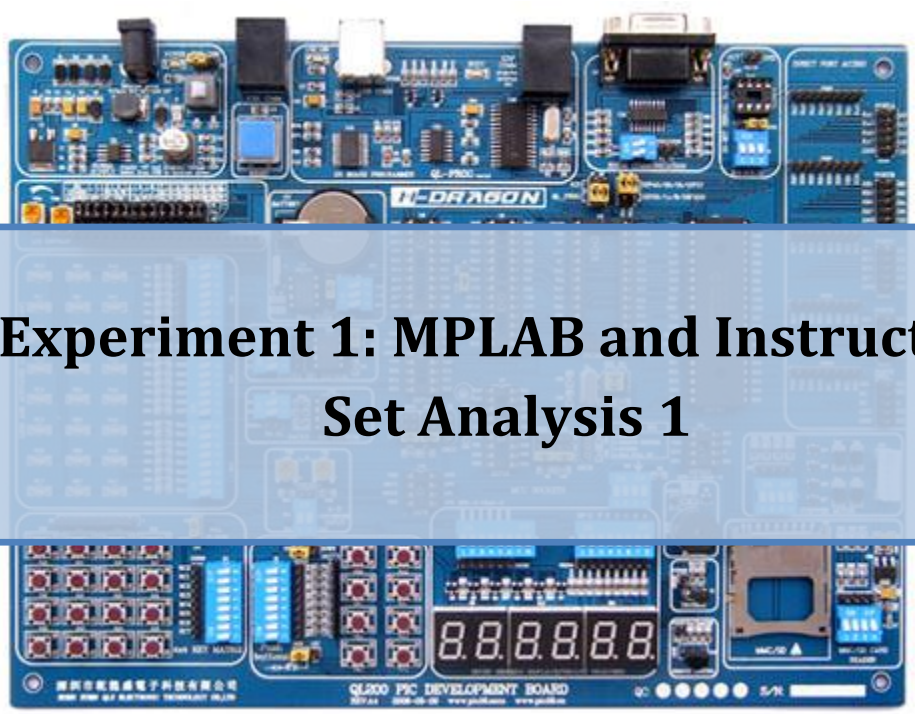
6. Program the PIC

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





University of Jordan
School of Engineering
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. You are required to install the MPLAB version that is available in the Lab.

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, while the other operand might be either a constant or a memory location. Many operations store their result in the working register; therefore, we usually need the following movement operations to perform arithmetic and logic instructions between two values:

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)

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

- **MOVLW**: 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:

NOTE: The DEFAULT BASE in MPLAB IS HEXADECIMAL

Examples:

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.

- **MOVWF**: 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.

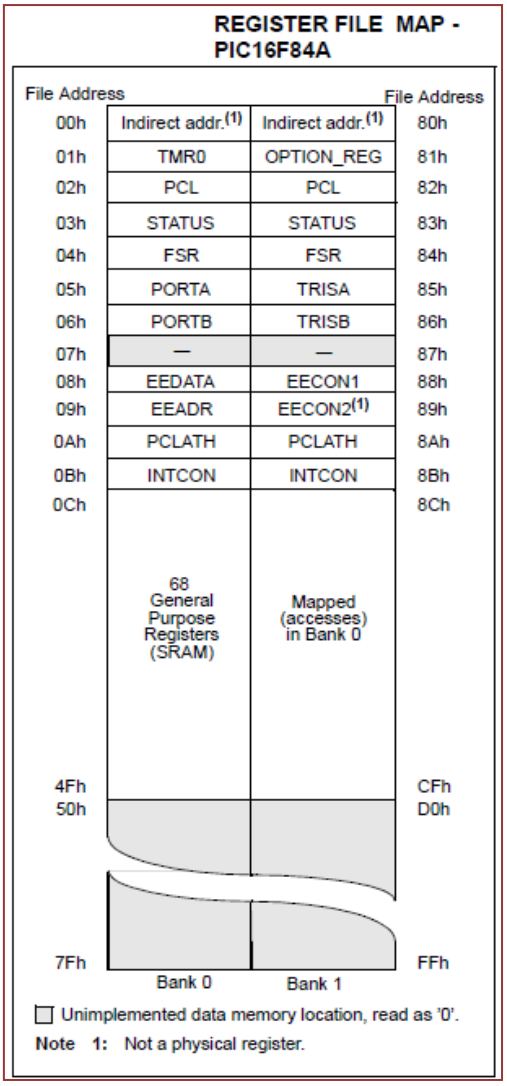
So what is the memory map?

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 (as shown in the figure below):

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.

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

REGISTER FILE MAP - PIC16F84A



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:

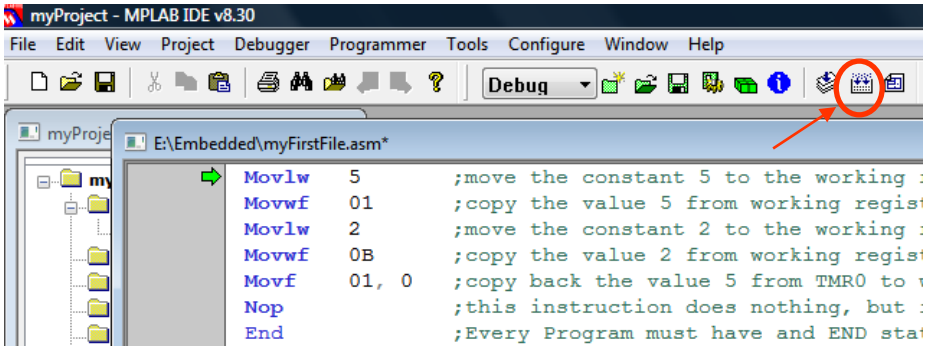
1. MOVF 05, 0 : copies the content of PORTA to the working register
2. MOVF 2D, 0 : copies the content of the GPR 2D the working register
3. MOVF 05, 1 : copies the content of PORTA to itself
4. MOVF 2D, 1 : copies the content of the GPR 2D to itself

Writing and Compiling Programs

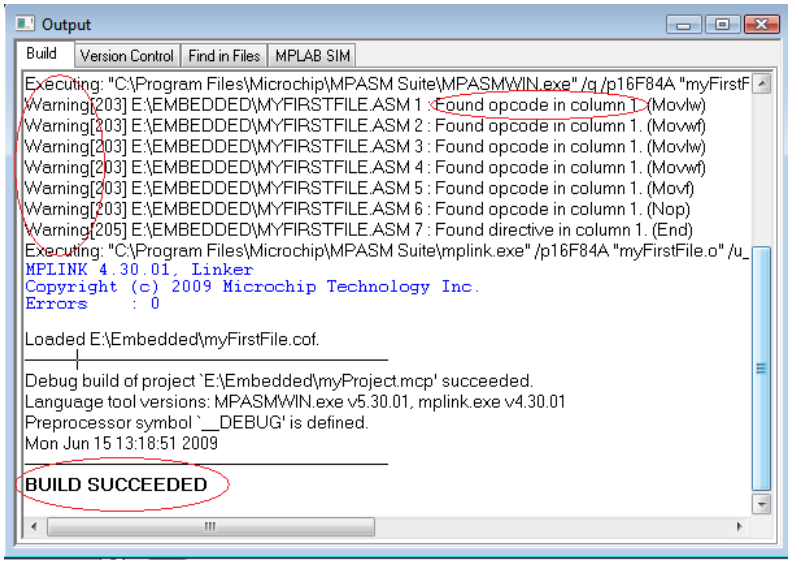
Now, let's try to use MPLAB to write a compile a simple program. In MPLAB write the following program.

```
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*** button.



An output window should show: BUILD SUCCEDDED

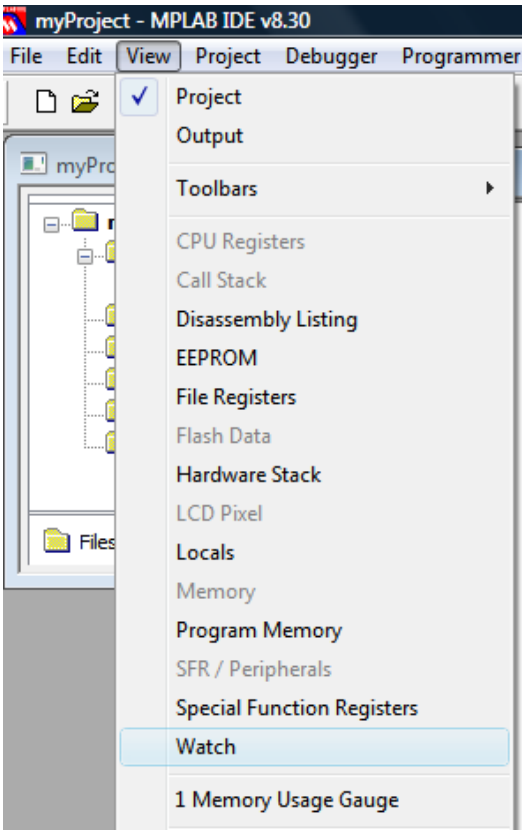


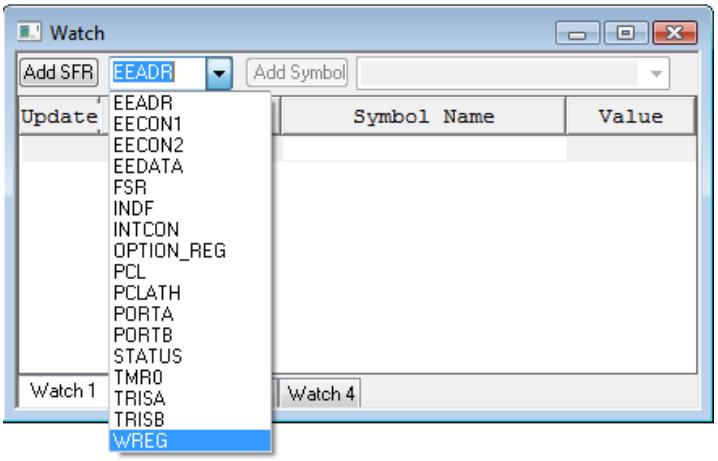
Notes on building programs:

- Build succeeded does not mean that the logic of your program is correct. It means that there are no SYNTAX errors.
- The warnings that you see 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.

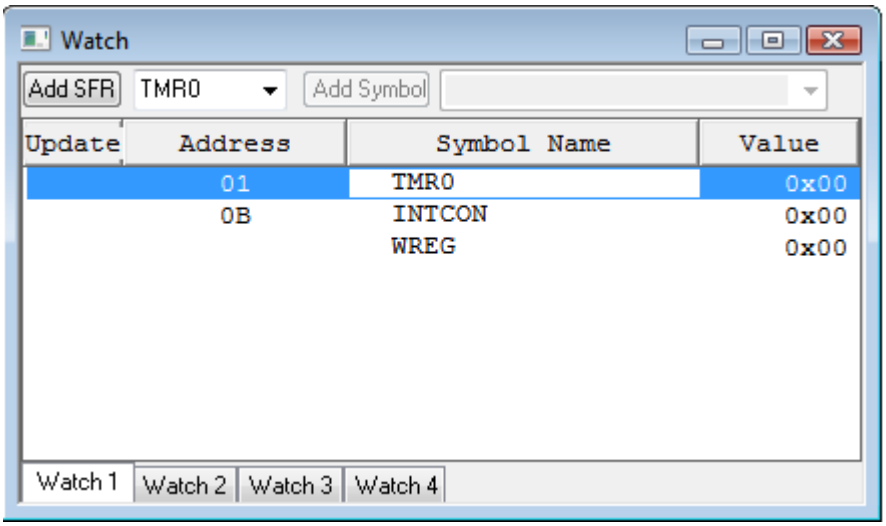
Preparing for Simulation

1. Go to View Menu → Watch

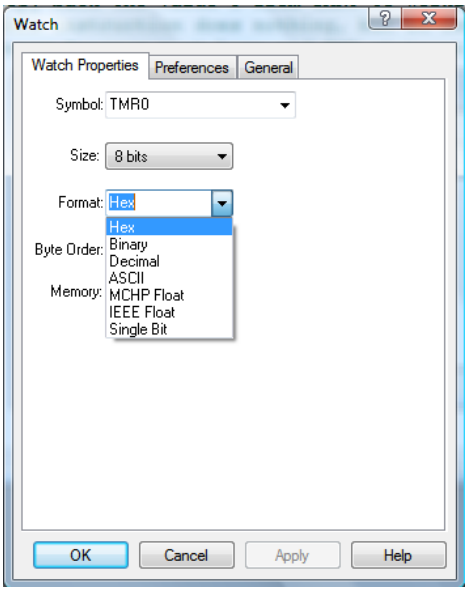




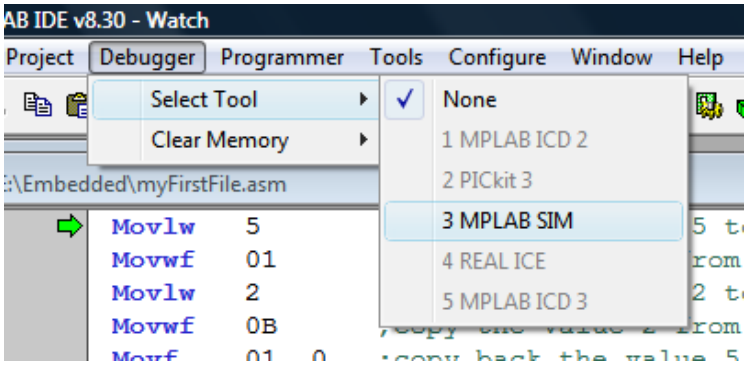
2. From the drop out menu choose the registers we want to watch during simulation and click ADD SFR for each one. Add the following registers: WREG, TMR0 and INTCON. You should have the following:



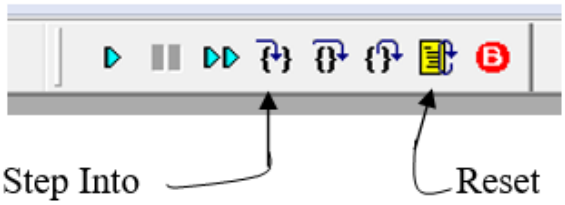
3. Notice that the default format is in hexadecimal, to change it (if you need to) simply right-click on the row → **Properties** and choose the new format you wish.



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

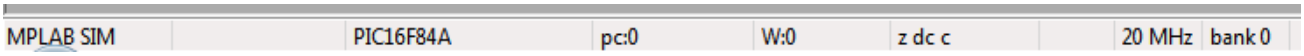


Now new buttons will appear in the toolbar as shown below.



5. 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**:**



6. 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" directive is a command 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.

❖ The "EQU" Directive

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 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.

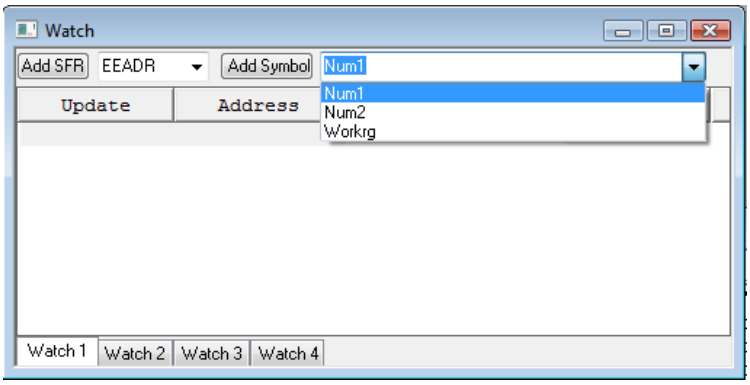
NOTE: DIRECTIVES THEMSELVES **ARE NOT CASE-SENSITIVE** BUT THE **LABELS** YOU DEFINE **ARE**. SO YOU MUST USE THE NAME AS YOU HAVE DEFINED IT SINCE IT IS CASE-SENSITIVE.

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 of registers 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 changes 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.

The 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.

Arithmetic and Logic Instructions

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 that.

	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.
The order is important in the subtract operation		
Examples (assuming you are using the include	xorlw 0BB W = W ^ 0BB	Andwf TMR0, W W = TMR0 & W

statement and appropriate equ statements for defining GPRs)	sublw .85
	W = 85_d – 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 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

```
1   include "p16f84a.inc"
2   Fib0    equ 20          ;At the end of the program the Fibonacci series numbers from 0 to 5 will
3   Fib1    equ 21          ;be stored in Fib0:Fib5
4   Fib2    equ 22
5   Fib3    equ 23
6   Fib4    equ 24
7   Fib5    equ 25
8
9   Clrw                    ;This instruction clears the working register, W = 0
10  clrf    Fib0            ;The clrf instruction clears a file register specified, here Fib0 = 0
11  movf    Fib0, w         ;initializing Fib1 to the value 1 by adding 1 to Fib0 and storing it in Fib1
12  addlw   1
13  movwf Fib1
14
15  movf    Fib0, W         ; Fib2 = Fib1 + Fib0
16  addwf   Fib1, W
17  movwf Fib2
18
19  movf    Fib1, W         ; Fib3 = Fib2 + Fib1
20  addwf   Fib2, W
21  movwf Fib3
22
23  movf    Fib2, W         ; Fib4 = Fib3 + Fib2
24  addwf   Fib3, W
25  movwf Fib4
26
27  movf    Fib3, W         ; Fib5 = Fib4 + Fib3
28  addwf   Fib4, W
29  movwf Fib5
30  nop
31  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

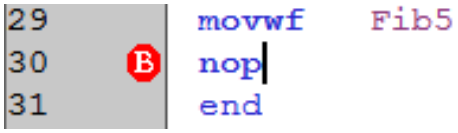
- A **capital letter** means that the value of the flag is **one**; meanwhile a **small letter** means a value of **zero**. In this case, the result is not zero; however, digit carry and a carry are present.

Run and Breakpoints

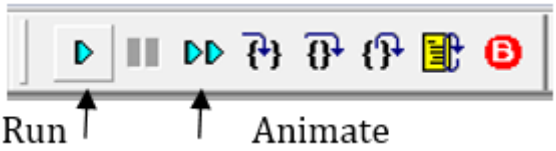
Many times you will need to make some changes to your code, additions, omissions and bug fixes. It is not then flexible to step into your code step by step to observe the changes you have made especially when your program is large. It would be a good idea to execute your code all at once or up to a certain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



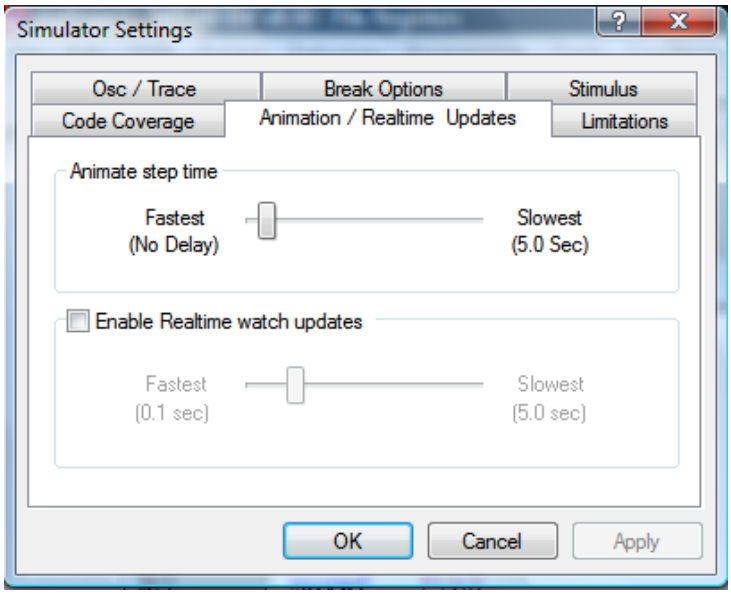
2a. **Now press the run button**



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:

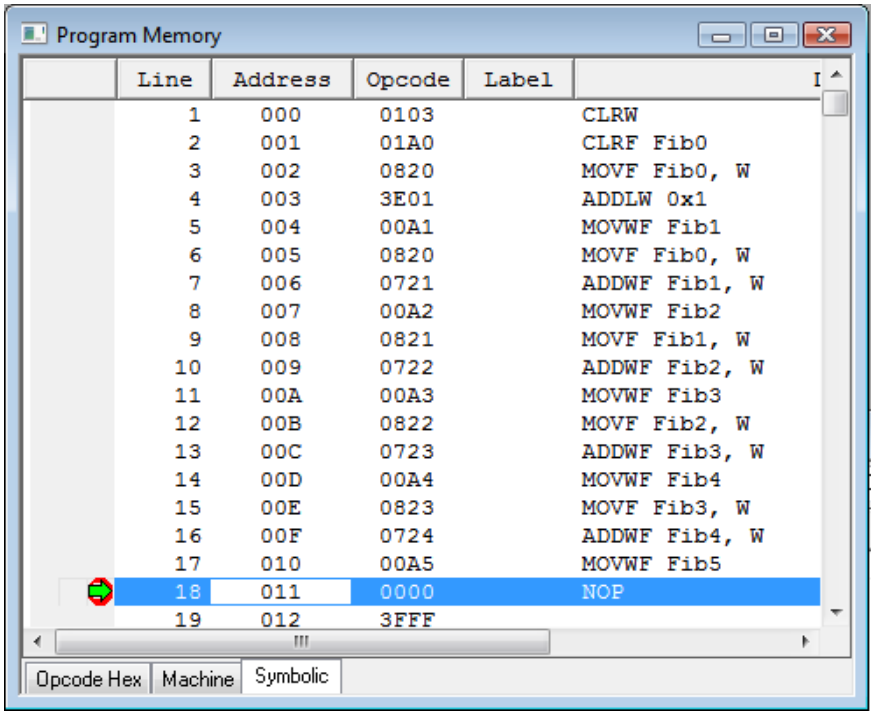
1. **Debugger** ↳ **Settings** ↳ **Animation/ Real time Updates**
2. Drag the slider to set the speed of simulation you find convenient



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



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

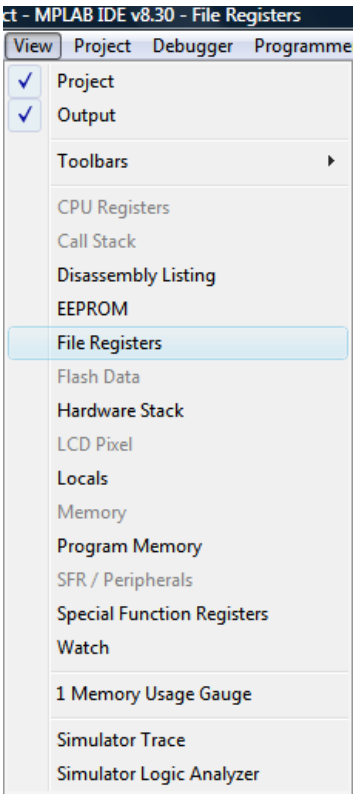
```
1   include "p16F84A.inc"
2
3   cblock  0x25
4           VarX
5           VarY
6           VarZ
7           Result
8   endc
9
10          org     0x00
11  Main                            ;loading the truth table
12          movlw B'01010101'       ;ZYX    Result
13          movwf VarX              ;000    0       (bit7_VarZ, bit7_VarY, bit7_VarX)
14          movlw B'00110011'       ;001    1       (bit6_VarZ, bit6_VarY, bit6_VarX)
15          movwf VarY              ;010    1                .
16          movlw B'00001111'       ;011    1                .
17          movwf VarZ              ;100    1                .
18                                  ;101    0                .
19                                  ;110    0                .
20                                  ;111    0       (bit0_VarZ, bit0_VarY, bit0_VarX)
21          movf    VarX, w
22          iorwf   VarY, w
23          xorwf   VarZ, w
24          movwf Result
25          nop
26          end
```

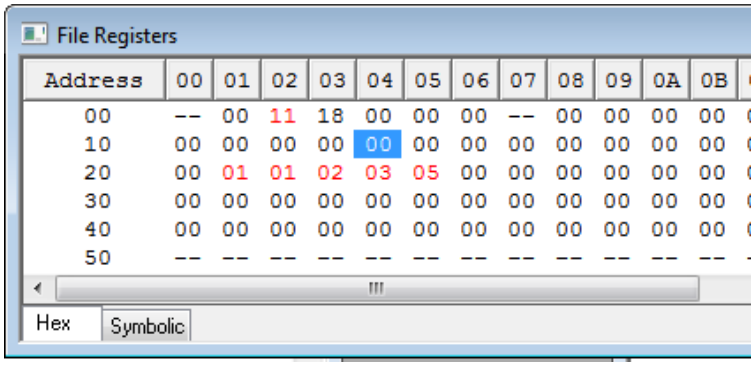
1. Start a new MPLAB session, add the file ***example2.asm*** to your project
2. Build the project
3. Select **Debugger ↳ Select Tool ↳MPLAB SIM**
4. Add the necessary variables and the working register to the watch window (remember that user defined variables are found under the "**Add Symbol**" list)
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 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 lfff 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 lfff 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. To do so: Select View Menu File Registers



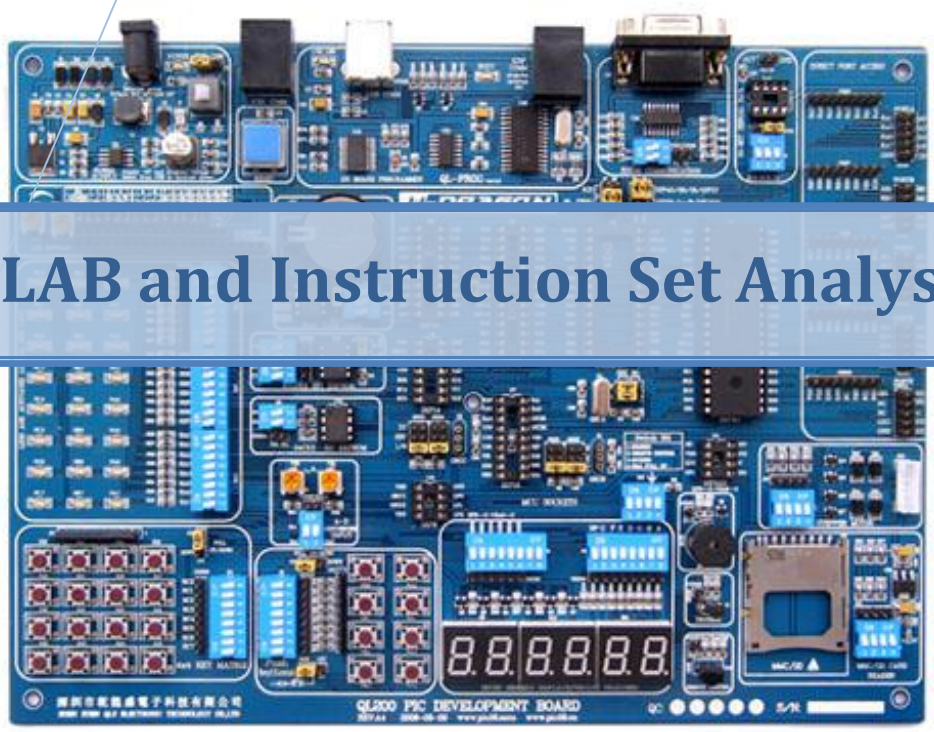


After building the Example1.asm codes, start looking at address 20 (which in our code corresponds to Fib0), to the right you will see the adjacent file registers from 21 to 2F.

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

Labsheet 1

University of Jordan
School of Engineering
Department of Computer Engineering
Embedded Systems Laboratory 0907334



MPLAB and Instruction Set Analysis 1

Name:

Student ID:

Section (Day/Time):

UNIVERSITY OF JORDAN
FACULTY OF ENGINEERING AND TECHNOLOGY
DEPARTMENT OF COMPUTER ENGINEERING

COMPUTER NAME:

Labsheet 1: MPLAB and Instruction Set Analysis 1

Name: **Student ID:**

Section:

(Pre-lab) Part 1: Starting up with instructions

Answer the following short questions:

A) Write the instruction(s) needed to load the working register with the value 9F.

B) Write the instruction(s) needed to load the register REGX with the value 6B.

C) Write the instruction(s) needed to switch to Bank 2.

D) Write the instruction(s) needed to decrement the value found in PORTC with 9.

E) Write the instruction(s) needed to complement the value found in REGA.

F) Write the instruction(s) needed to Multiply the value found in TMR0 with 8 (Hint: rotating a number to the left multiplies it by two, use **RLF** instruction. Remember that the rotation operation in PIC16 series is through the carry flag. (*refer to the pic16f84A datasheet Page38)*

G) Write the instruction(s) needed to divide the value found in PORTB by 2 (Hint: rotating a number to the right divides it by two, use **RRF** instruction. Remember that the rotation operation in PIC16 series is through the carry flag.

(Pre-lab) Part2: Implementing logical function

1. Start a new MPLAB project, add the file ***example2.asm*** to your project.
2. Build the project.
3. Select **Debugger** ↳ **Select Tool** ↳**MPLAB SIM.** Add the variable **Result** and the **Working register** to the watch window.
4. Simulate the program step by step. Stop at the NOP instruction.
5. What is the content of Result register after executing your code?

Result =

Part 3: Simulate a program in MPLAB and check the results

1. Create a project with the code below in your ASM file.

```
#include "p16F84A.inc"
Val1    equ     22
Val2    equ     33
Val3    equ     44
Val4    equ     45

        movlw   03
        movwf   VAL1
        movlw   09
        movwf   Val2
        addwf   Val1, w
        movwf   Val3
        rrf     Val3,1
        movf    Val3, w
        iorlw   80
        movwf   Val4
        nop
```

2. Build the project and the output window will show you numerous messages for errors and warnings. Double click on the error. The pointer will move you to the line that caused it.
3. In the space below, list the two errors in the program

Line Number	Error	Correction

4. After correcting the errors in the ASM file:
   - **a)** Select **Debugger** ↳ **Select Tool** ↳**MPLAB SIM.**
   - **b)** Select **View** ↳ and select **Watch.**
   - **c)** Under the "**Add Symbol**" list, add the variables Val1, Val2, Val3 and Val4.
   - **d)** Under the "**Add SFR**" list, add the working register to the watch window.
   - **e)** Simulate the program step by step, analyze and study the function of each instruction.
   - **f)** **Stop** at the NOP instruction**.**
   - **g)** What is the content of Val3 and Val4 registers after executing your code?

Val3 =
Val4 =

Part 4: Writing and Simulating Programs (1)

1. Create a project with an ASM file, using the steps in the Experiment 0 file.
2. In the ASM file, define the following variables in memory at the specified address.

Variable	Address
NUM1	0x20
NUM2	0x21
Result	0x22

3. In the ASM file, write a program that performs the following operation. Copy and paste your code in the space below.

$$\boldsymbol{Result = NUM1 + NUM2 - D'13'}$$

4. Test your program when NUM1 = D'16' and NUM2 = D'89'. Use the Watch window to examine the Result variable.

Part 5: Writing and Simulating Programs (2)

1. Create a project with an ASM file, using the steps in the Experiment 0 file.
2. In the ASM file, define the following variables in memory at the specified address.

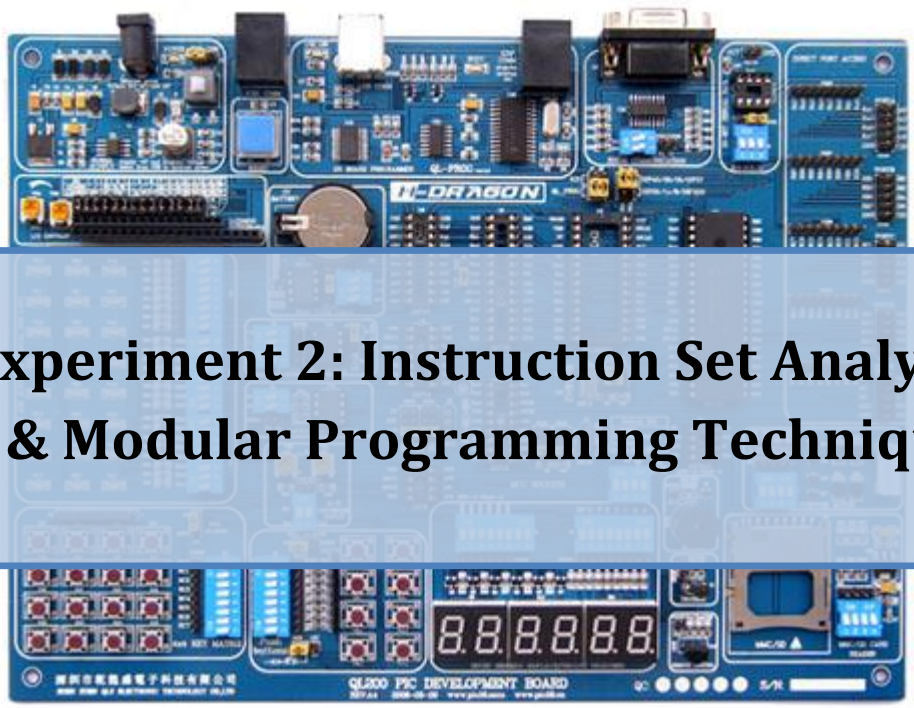
Variable	Address
NUM1	0x30
NUM2	0x31
Result	0x32

3. In the ASM file, write a program that performs the following operation. Copy and paste your code in the space below.

$$\boldsymbol{Result} = \frac{(\boldsymbol{NUM1} \times \mathbf{2} + \boldsymbol{NUM2} + \mathbf{15})}{\mathbf{2}}$$

4. Test your program when NUM1 = D'20' and NUM2 = D'12'. Use the Watch window to examine the Result variable.

University of Jordan
School of Engineering
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.

Written by Eng. Enas Jaara and Dr. Ashraf Suyyagh – Revised by Prof. Iyad Jafar

Introducing Conditional Instructions in PIC

The PIC 16series instruction set has four instructions which implement a sort of conditional statements. These are: ***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 *btfsc PORTA, 0* instruction tests bit 0 of PORTA and checks whether it is clear (0) or not such that:

- 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

Accordingly, **if the condition fails**, the code will continue normally and both instructions will be executed.

Now, let's consider the following example.

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

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

- If bit 0 in PORTA is clear (0), then 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

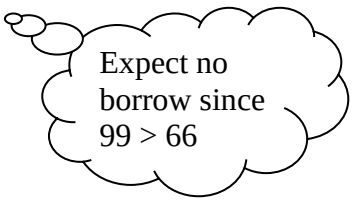
Conditional using Subtraction and how the Carry/Borrow flag is affected?

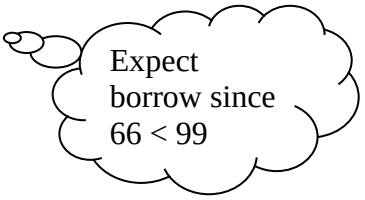
The Carry concept is easy when dealing with addition operations but it differs in borrow operations according to Microchip implementation.

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/Borrow $\overline{\text{flag}}$ and how it is affected after addition and subtraction operations.

Ex1) 99-66	Ex 2) 66 – 99
10011001 – 01100110 10011001+ 10011010 (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	01100110- 10011001 01100110+ 01100111 ^{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, then Result will have the ASCII value G. Otherwise, Result will have the ASCII value S.

```
1   include "p16F84A.inc"
2   cblock  0x25
3           testNum
4           Result
5   endc
6           org     0x00
7   Main
8           movf    testNum, W
9           sublw   .10             ;10d - testNum
10          btfss   STATUS, C
11          goto    Greater         ;C = 0, that's B = 1, then testNum > 10
12          goto    Smaller         ;C = 1, that's B = 0, then testNum < 10
13  Greater
14          movlw A'G'
15          movwf Result
16           goto   Finish
17  Smaller
18          movlw A'S'
19          movwf Result
20  Finish
21          nop
22          end
```

Let's simulate this program in MPLAB to verify its operation.

1. Start a new MPLAB session, add the file ***program1.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 here 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, i.e. 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)

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

```
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
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 program 2, 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: Repetition Structures

You have observed in the code of Program 2 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. This reduces the total number of instruction in the program.

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

```
1             include "p16f84a.inc"
2             cblock 0x20
3                     testNum
4                     tempNum
5             endc
6
7             cblock 0x30
8                     numOfOnes
9                     counter         ; since repetition structures require a counter, one is declared
10            endc
11
12            org 0x00
13            clrf    numOfOnes
14            movlw   0x04            ; counter is initialized by 4, the number of the bits to be tested
15            movwf   counter
16            movf    testNum, W
17            andlw   0x0F
18            movwf   tempNum
19    Again
20            rrf     tempNum, F
21            btfsc   STATUS, C
22            incf    numOfOnes, F
23            decfsz  counter, F          ; The contents of register counter are decremented then test :
```

```
24          goto    Again           ; if the counter reaches 0, it will skip to "nop" and program
25  ends
26          nop                     ; if the counter is > 0, it will repeat "goto Again"
            end
```

Introducing 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 that we learnt in high-level languages. Subroutines have the following requirements and features:

- A subroutine starts with a **Label** giving them a name and end with the instruction **return.**
- 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.

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

Remember that subroutines alter the flow of the program; thus they affect the program counter and stack. So what is the stack and how is it used? Consider the following code which contains the main program and the doMath subroutine.

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

```
        Nop

doMath
        Instruction35
        Instruction36
        Instruction37
        return
```

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, and **return**. What will happen when the return instruction is executed? What is the next instruction that is executed next?

When the Call doMath instruction is executed, the address of the next instruction (which as you should already know is 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 using the macro and endm directives as shown below.

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

Macros have the following requirements and features:

- ❖ Macros should be declared at the beginning of your code, i.e. before the main program. It is not recommended to declare macros in the middle of your program.
- ❖ Macros can be used in your code by only writing their name: macroName
- ❖ Each time you use a macro, the assembler will replace the macro name the you write with its body as show in it will be replaced by its body. Therefore, the program will execute sequentially, i.e. the flow of the program will not change and 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		**Example5 using Subroutine**
1	include "p16f84a.inc"	1	include "p16f84a.inc"
2		2	
3	cblock 0x30	3	cblock 0x30
4	Num1	4	Num1
5	Num2	5	Num2
6	endc	6	endc

```
7
8   Summation     macro
9                 movf    Num1, W  ;Macro
10  Body
11                addwf  Num2, F
12                endm
13
14          org 0x00
15  Main
16          Movlw 4
17          Movwf Num1
18          Movlw 8
19          Movwf Num2
20          Summation
21          Movlw 1
22          Movwf Num1
23          Movlw 9
24          Movwf Num2
25          Summation
26
27  finish
28          nop
            end
```

```
7
8
9
10
11
12
13          org 0x00
14  Main
15          Movlw 4
16          Movwf Num1
17          Movlw 8
18          Movwf Num2
19          Call Summation
20          Movlw 1
21          Movwf Num1
22          Movlw 9
23          Movwf Num2
24          Call Summation
25          goto finish
26
27  Summation
28          movf    Num1, W
29          addwf  Num2, F
30          return
31  finish
32          nop
33          end
```

Analyzing the two programs and highlighting the differences

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

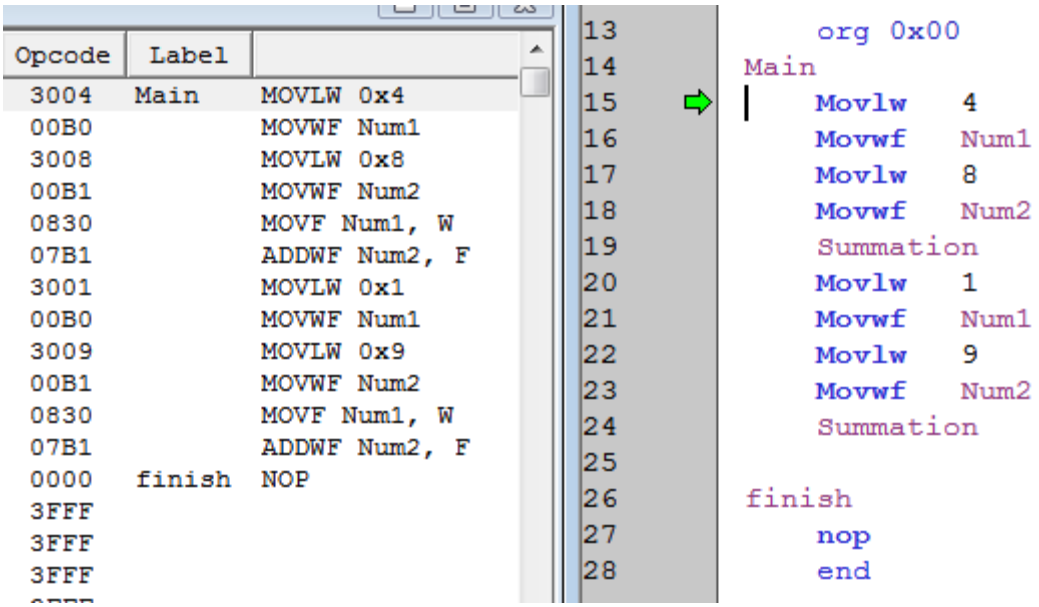
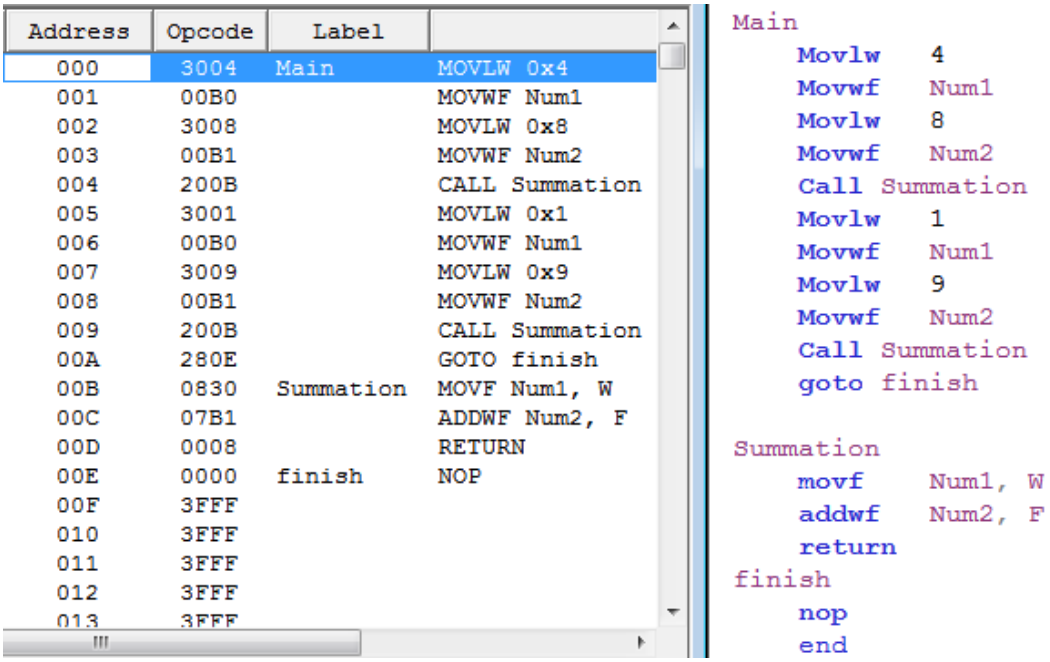


Figure 1. The example using macros

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		

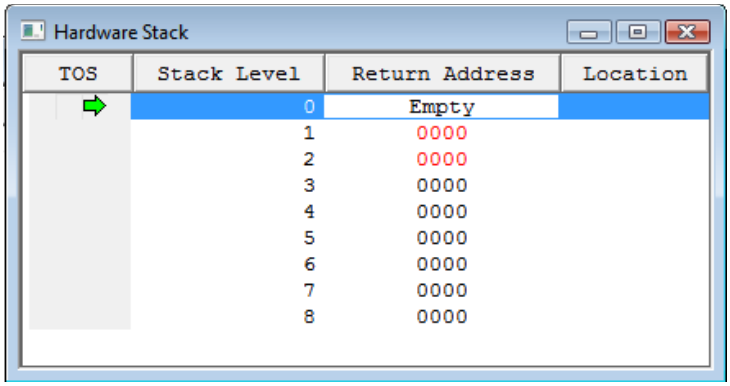
```
Main
    Movlw   4
    Movwf   Num1
    Movlw   8
    Movwf   Num2
    Call Summation
    Movlw   1
    Movwf   Num1
    Movlw   9
    Movwf   Num2
    Call Summation
    goto finish

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

Figure 2. The example using subroutines

On the other hand, Figure 2 shows 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. To investigate the effect of subroutines on the stack, do the following for ***Program Two***:

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. Also, 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.**

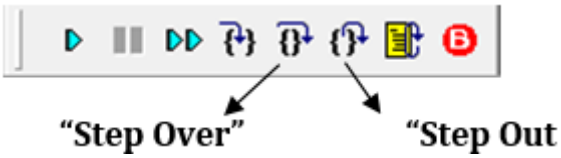
Important Notes:

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

While stepping through program execution, you might need to execute the subroutine without seeing the execution of instruction inside it. This is usually used when you know that that the subroutine executes correctly and you are only interested to see execution of its instructions. For this purpose, you can use the Step Over option in the simulation toolbar, as shown below, when you the execution reaches the call instruction. For example, to use this option in Example 5, you need to do the following:

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



The Step Out option shown in the toolbar resembles Step Over operation; however, it is used 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. For example, to use this option in Example 5, you need to do the following:

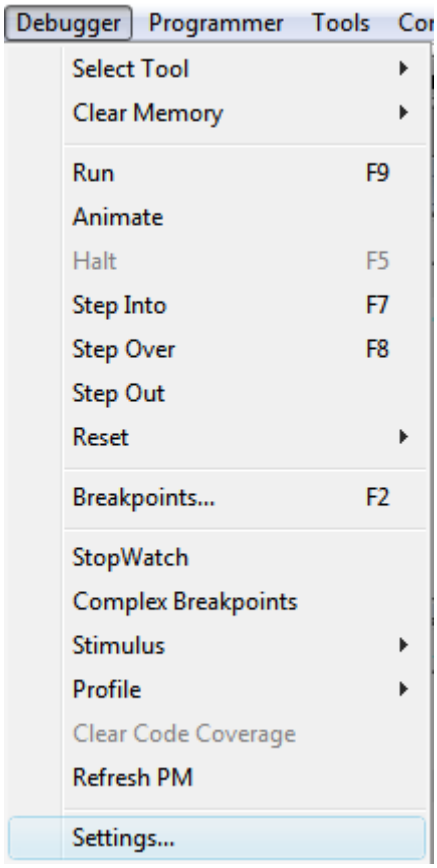
1. Simulate the program up to the point when the green arrow points to the first instruction inside the Summation subroutine: movf Num1, W
3. Press Step Out. Observe how the program execution continues from the instruction after the Call Summation instruction.

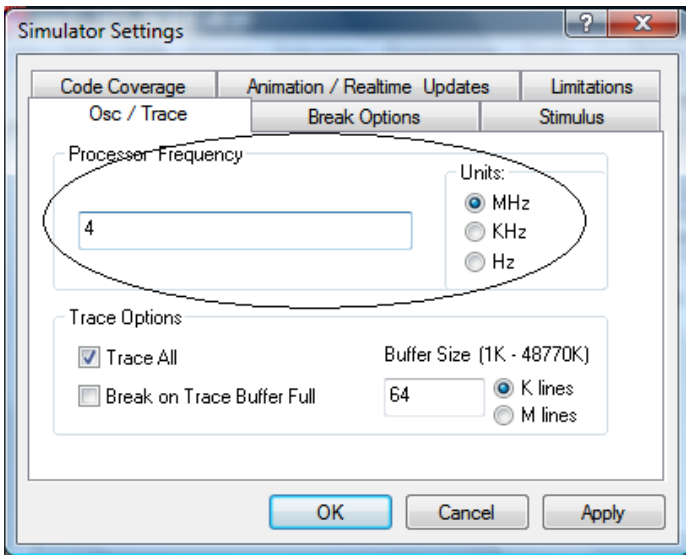
In both cases, the instructions inside the subroutine are executed but you only see the end result of the subroutine.

Measuring the Execution Time

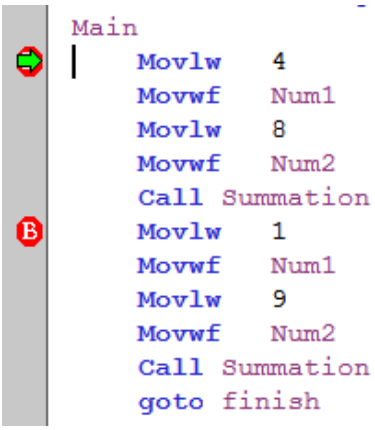
To measure the total time spent in executing the whole program or a certain subroutine, do the following:

1. Set the oscillator (external clock speed, Fosc) by following the figure below.
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.**

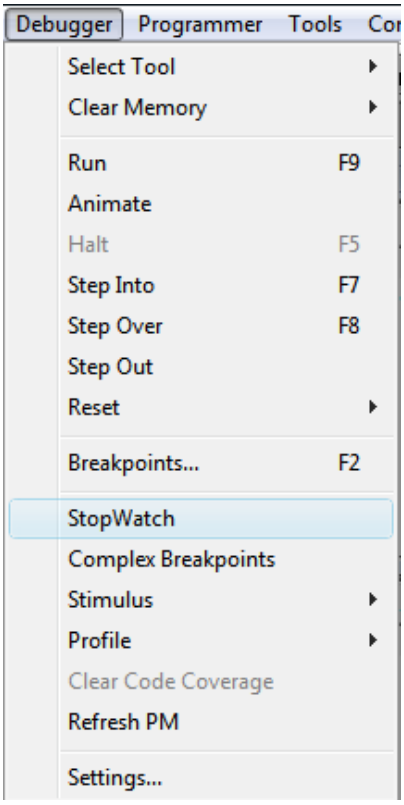


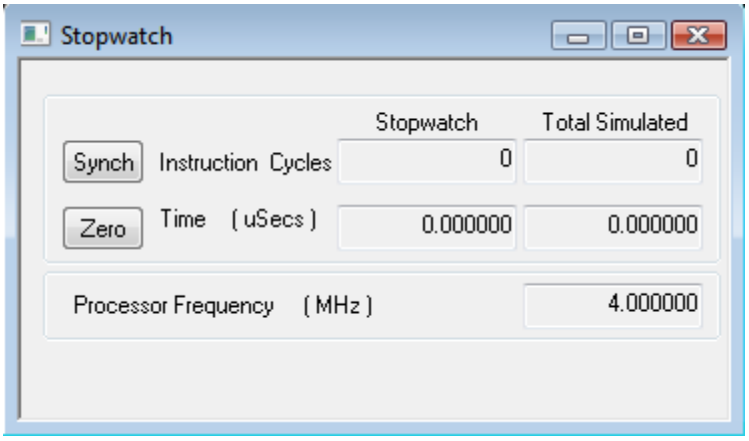


3. Now set breakpoints at the beginning and end of the code you want to calculate time for



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 Approaches: Read the following problem statement

A PIC microcontroller will take as an input two sensor readings and store them in Num1 and Num2. Then, it will multiply both values 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. Remember:

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 that repeats the addition operation certain number of times. For example, suppose we want to multiply 9x4 and number 4 is in location temp. 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
```

The following table compares the code required to perform multiplication by 5 using non-modular and modular approaches.

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 25 function	5
include "p16f84a.inc" cblock 0x30		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
        movwf Num1_25

        movf    Num2, w    ;Num2 x 25
        movwf temp
        movlw .24
```

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

```
        movwf counter
        movf   temp, w
add4
        addwf  temp, w
        decfsz counter, f
        goto   add4
        movwf Num2_25
        goto   finish
finish
        nop
        end
```

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:

Approach 1	Approach 2
Place the input in the working register Take the output from the working register **Example:**	Store the input(s) in external variables Load the output(s) in external variables **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
	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	This approach is useful when the subroutine/macro takes many inputs and

output	produces multiple outputs

Special Subroutines: Lookup Tables

Lookup 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; therefore, alter the flow of instruction execution. The **retlw** instruction is simply a **return** instruction with additional feature that is returning 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 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
```

Exercise.

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. In MPLAB IDE, 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   X
```

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.

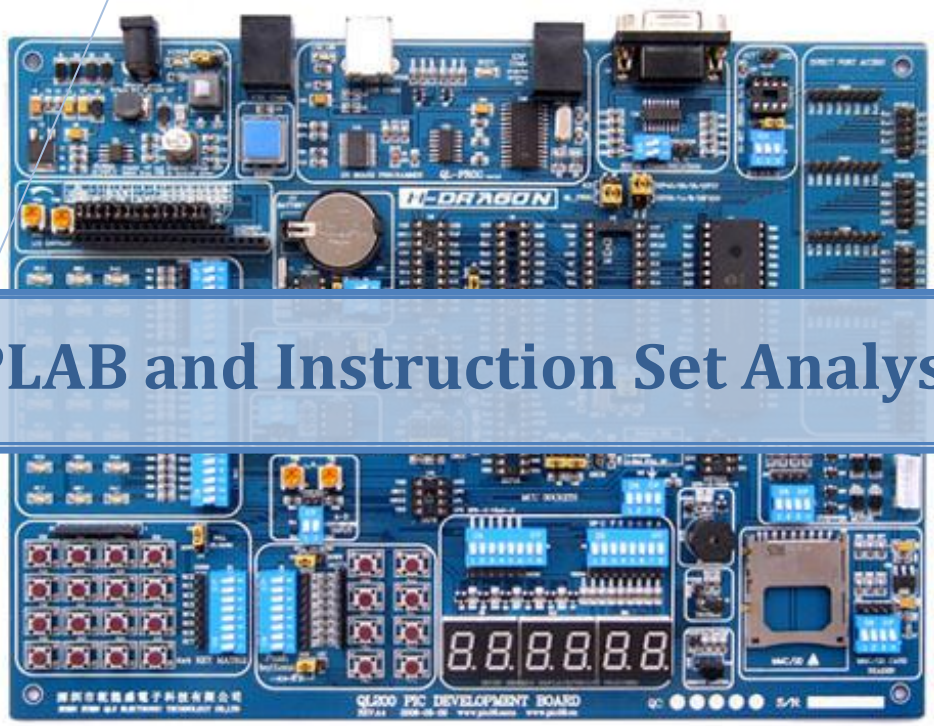
```
;*****************************************************************************************
;* Subroutine Name: ExampleSub
;* Function: This subroutine multiplies the value found in the working register by 16
;* Input: Working register
;* Output: Working register * 16
;***************************************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

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 lfff 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 lfff 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	

Labsheet 2

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



MPLAB and Instruction Set Analysis 2

Name:

Student ID:

Section (Day/Time):

COMPUTER NAME:

UNIVERSITY OF JORDAN
SCHOOL OF ENGINEERING
DEPARTMENT OF COMPUTER ENGINEERING
EMBEDDED SYSTEMS LABORATORY CPE0907334

EMBEDDED SYSTEMS LABORATORY CPE0907334

Labsheet 2: Instruction Set Analysis 2 & Modular Programming Techniques

Name: **Student ID:**
Section:

Pre-lab) Part 1: Starting up with instructions

Answer the following short questions:

A) Write the instructions needed to *check if the value found in REGA equal to 10*.

B) Write the instructions needed to *check if the value found in REGA equal to zero.*

(Pre-lab) Part2: Code Analysis Skills

Answer the following questions regarding Program 3 in the experiment on page 5. Each question is independent from the others.

1. What will the results of the **Program 3** be when we substitute the instruction @ line 23 "decfsz counter, F" with decfsz counter, W?

2. Assuming that the PIC runs at an external oscillator speed of 4 MHz? What is the time spent in executing the **Program 3.asm** code to reach the NOP instruction?

(Pre-lab) Part3: Code Writing Skills

Modify Program1.asm code in the experiment to test if testNum has the decimal value 10, then Result will have the ASCII value 'R'.

Your code

Part4: Code Analysis Skills

Read and simulate the given code *Labsheet2.asm* and answer the questions which follow. To prepare for simulation, perform the following steps:

- Go to View Menu -> Watch.
- From the drop out menu choose the registers you want to watch during simulation and click ADD Symbols for each one (Num, Num_7, Num_49).
- Select Debugger ->Select Tool ->MPLAB SIM.
- Simulate the program.

1. What will be stored in the following registers when Num has the values listed in the table below?

Num	Value of Num_7 after Mul7 subroutine call in Main	Value of Num_49 after Mul49 subroutine call in Main
0x02		
0x05		

2. What is the total number of instructions inside the Mul49 subroutine?

3. Where does Mul7 subroutine expect to find its input? Where does it store its output?

Input: Output:

4. What is the value at the top of stack when the Mul49 subroutine call instruction executed is?

Part 5: Timing Program Execution

Consider the following code. Determine the elapsed time when the program execution reaches the **NOP** instruction inside the L1 subroutine. Assume a PIC16F84A microcontroller running at 800KHz. Use the Stop Watch tool in MPLAB. You need to load locations CNT1 and CNT2 and VAL with values 0x15, 0xC2 and 0x05, respectively, before simulating the code. Show your simulation to the supervisor.

```
        #include "P16F84A.inc"
        cblock  0x25
                CNT1
                CNT2
                VAL
        endc

        org     0x0000
        movf    CNT1, W
        subwf   CNT2, F
        decf    CNT2, F
        btfss   STATUS, C
        call    L1
        incf    VAL, F

L1      incf    VAL, F
        movf    CNT2, W
        andwf   VAL, F
        bcf     STATUS, C
        rrf     VAL, F
        decfsz  CNT1, F
        goto    L1
        nop
        return
        end
```

Time =	

Part 6: Code Writing Skills

Write a program which converts a number from unpacked BCD format saved in three registers to one decimal number, assuming that registers names are BCDH, BCDM and BCDL. BCDH (High Digit of the decimal number) is at location 0x21, BCDM (Mid Digit of the decimal number) is at location 0x22, BCDL (Low Digit of the decimal number) is at location 0x23, and Result is at location 0x40. Also, assume that all BCD numbers are in the valid range of 0 – 9.

For example, if BCDH = 2, BCDM = 4 and BCDL = 3, then your program should do the math to store the value D'243' in location Result. This can be done by writing the code to perform the following operation

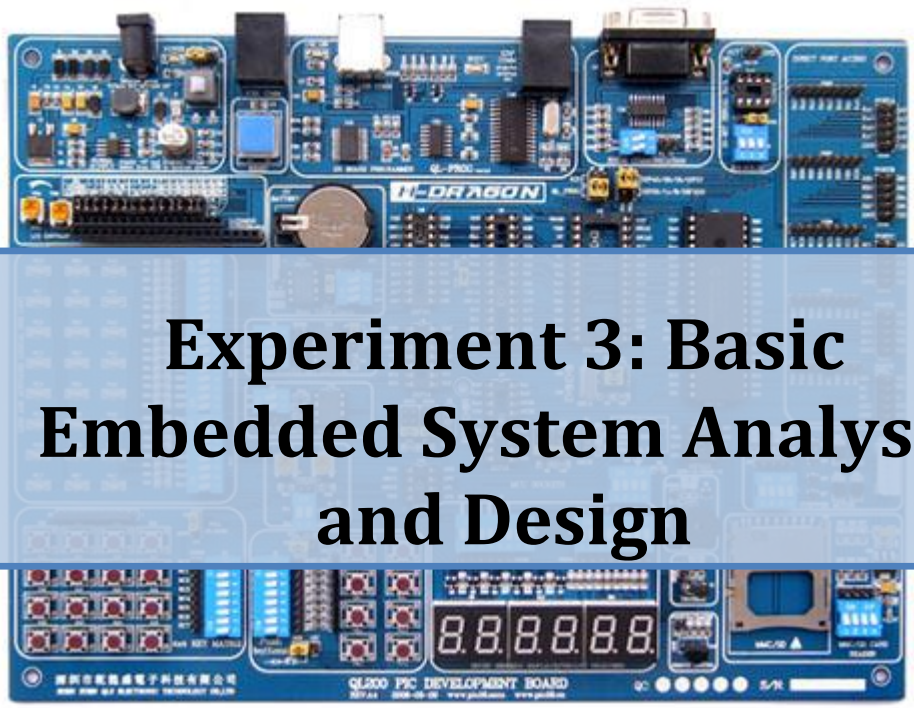
Result = BCDL + BCDM x 10 + BCDH x 100

You are required to use modular programming in your code.

Perform simulation after entering some values for BCDL, BCDM and BCDH and show the results to the supervising engineer. You will need to watch the variables BCFL, BCDM, BCDH and Result.

Copy and paste your code here.

University of Jordan
School of Engineering
Department of Computer Engineering
Embedded Systems Laboratory 0907334



3 Experiment 3: Basic Embedded System Analysis and Design

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.

Written by Eng. Enas Jaara and Dr. Ashraf Suyyagh – Revised by Prof. Iyad Jafar

1. Starting-up System Design

When we attempt to design a system that is required to perform complex tasks, it is essential that we think 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 *flowchart*. A flowchart is a schematic representation of an algorithm, showing the steps and operations in the algorithm using different shapes that are connected using unidirectional arrows to show their sequence. Flowcharts are used in designing and/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 diagrams which are not covered in this course.

A good practice in designing complex systems is to break them into smaller pieces where each carries out few simple related tasks of the overall system. Thus, the system is built from these simple subsystems. In this approach, you need only to care about how these subsystems interface with each other. As you learned in Experiment 2, 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.

Generally, 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-system.

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 to demonstrate the design process.

2. Design 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 and a filling pump starts working for some time to fill the bottle. When the bottle is filled, the total number of filled bottles is increased by one and is shown on a common cathode 7-Segments display. Afterwards, 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. At this phase, 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. Figure 1 shows an example of the filling machine.

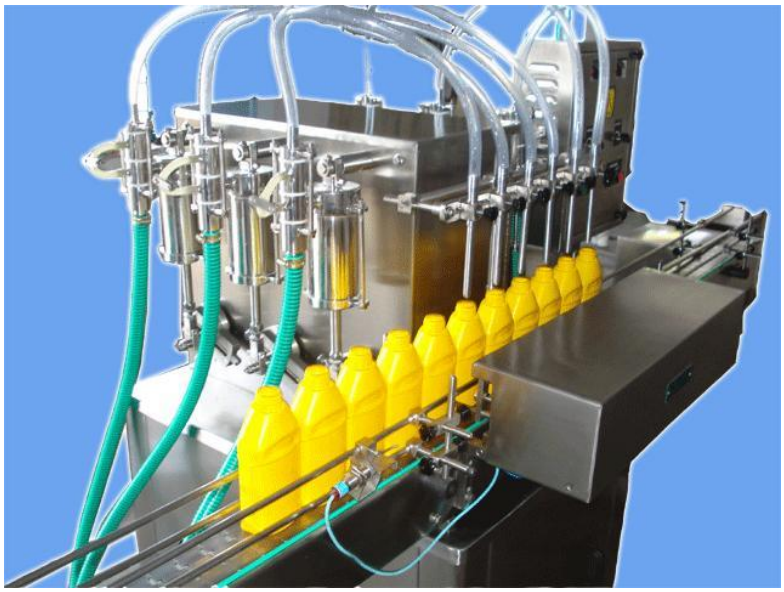


Figure 1. An industrial filling machine

In order to design this system and determine the required hardware and their role as input or output, we will follow the steps listed previously.

Step1: Requirements Definition and Analysis

To analyze the system, let's remember the following:

- An Output in embedded system means a signal need be sent from the PIC to external hardware for control purposes.
- An input in embedded system means a signal is received from external hardware into the PIC for processing.
- Processing in embedded system means a certain code which does the required job.

Accordingly, and based on the problem statement, we can identify the following operations:

1. *The empty bottles move on a conveyer belt and when a bottle is detected, the conveyor belt stops. This implies that:*
   - There is a motor which controls the conveyor "conveyor motor". The PIC should control the motor. So, there we need an Output to control the motor.
   - There is a sensor which detects the presence of a bottle "bottle sensor". The PIC should read the sensor reading. So, we need an Input.
2. *A filling machine starts filling the bottle for a specified period of time after which the filling machine stops. This implies that:*
   - There is a pump/motor which is turned on/off to fill the bottle "filling motor". The PIC should control the pump. So, we need an Output.
   - We need a mechanism to calculate this time period. This is Processing done by the PIC. Can be done using hardware timers or software delay loops.
3. *The total number of filled bottles is increased by one and shown on a common cathode 7-Segments display. This implies that:*
   - We need some sort of a counter. Reserve a memory location (GPR) in the PIC
   - We need to output the value of this counter to a 7-segment display. Output
4. *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. This implies that:*
   - We need to check the counter value continuously. This is Processing done by the PIC.
5. *When the number of bottles is 9, one LED lights on an 8-LED row and moves back and forth. Also, the conveyor belt does not start again until the resume button is pressed. When the button is*

pressed, the LED array turns off and the system restarts the operation. This implies that:

- We need a code to control the LED lights. The PIC should control the LEDs. So, we need an Output for each of these 8 LEDs.
- We need a mechanism to check for the resume button key press. The PIC should read the button. So, we need an Input.

As you have seen above, we need to interact with external components; like the motors, 7-Segments and the LEDs (output devices), as well as s ensors and switches (input devices). 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 reveals multiple I/O ports: PORTA and PORTB for the 16F84A, and the additional PORTC, PORTD and PORTE for the 16F877A. These ports are general-purpose bi-directional digital ports. Each port is associated with a direction register called *TRISx* that controls the direction of *PORTx* pins. A logic one in bit position ***k*** of *TRISx* register configures pin ***k*** of *PORTx* as input. On the other hand, if this bit is 0, the pin is configured as output. A pin can be configured as input or output an any instant of time, but not simultaneously. Figure 2 shows an example of configuring the pins in PORTA. Also, Table 1 shows some examples on how to configure the ports in software.

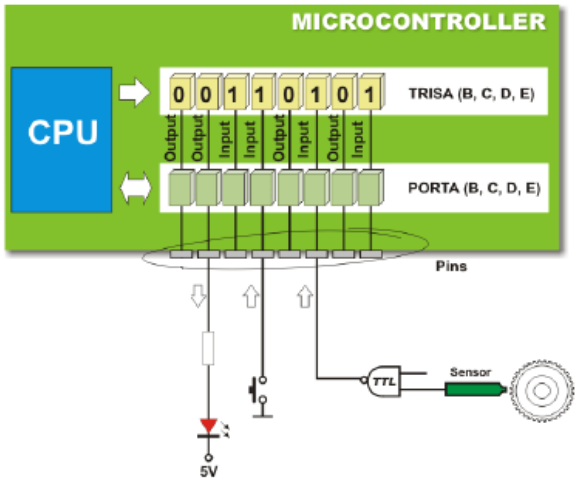


Figure 2. Configuring pins in PORTA.

Table 1. Examples on configuring ports in software

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

Note on PORTA and PORTE:

As you remember from the Embedded Systems course, PORTA and PORTE pins can be used as analog inputs and in this case they are connected to the A/D converter. In order to specify whether these pins are digital or analog, we need to configure the bits in the ADCON1 register (Refer to the datasheet for more details on choosing the values for ADCON1). For example, to configure all the pins in PORTA as digital, we can write:

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

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. For example:

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

In the system under consideration, we will use the following configuration:

Inputs:

- RA2: Bottle sensor
- RA3: Resume button

Outputs:

- RB0 to RB7: LEDs array
- 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:

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

A good practice in writing programs for embedded systems is to have "Initial" and "Main" subroutines in the program. 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 embedded applications require that these functions to be performed repeatedly; thus the program usually loops through the Main subroutine code infinitely.

Note: When designing a system, you should not expect to get the same design that others get. 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

Based on the analysis of the system operation we obtained in Step 1 and Step 2, we can visualize the general operation of the system in the flowchart shown in Figure 2. As we mentioned earlier, it is good practice to divide the tasks in the system into subroutines. In our design, we decided to distribute these tasks to five main subroutines as shown in Figure 2. These subroutines are:

- **Initial Subroutine:** it configures the ports and starts the conveyer belt.
- **Update_Seven_Seg subroutine:** reads the total number of filled bottles and converts it to seven segment code.
- **Test_and_Process subroutine**: displays the number of bottles on the 7segment, waits for bottle, stops the conveyor, fills the bottle, and restarts the conveyor.
- **Test_Resume subroutine:** checks if total number of filled bottles is nine. If so, it stops the machine and invokes the LEDs subroutine.
- **LEDs:** moves the LED in the LED array back and forth and tests for pressing the resume button press.

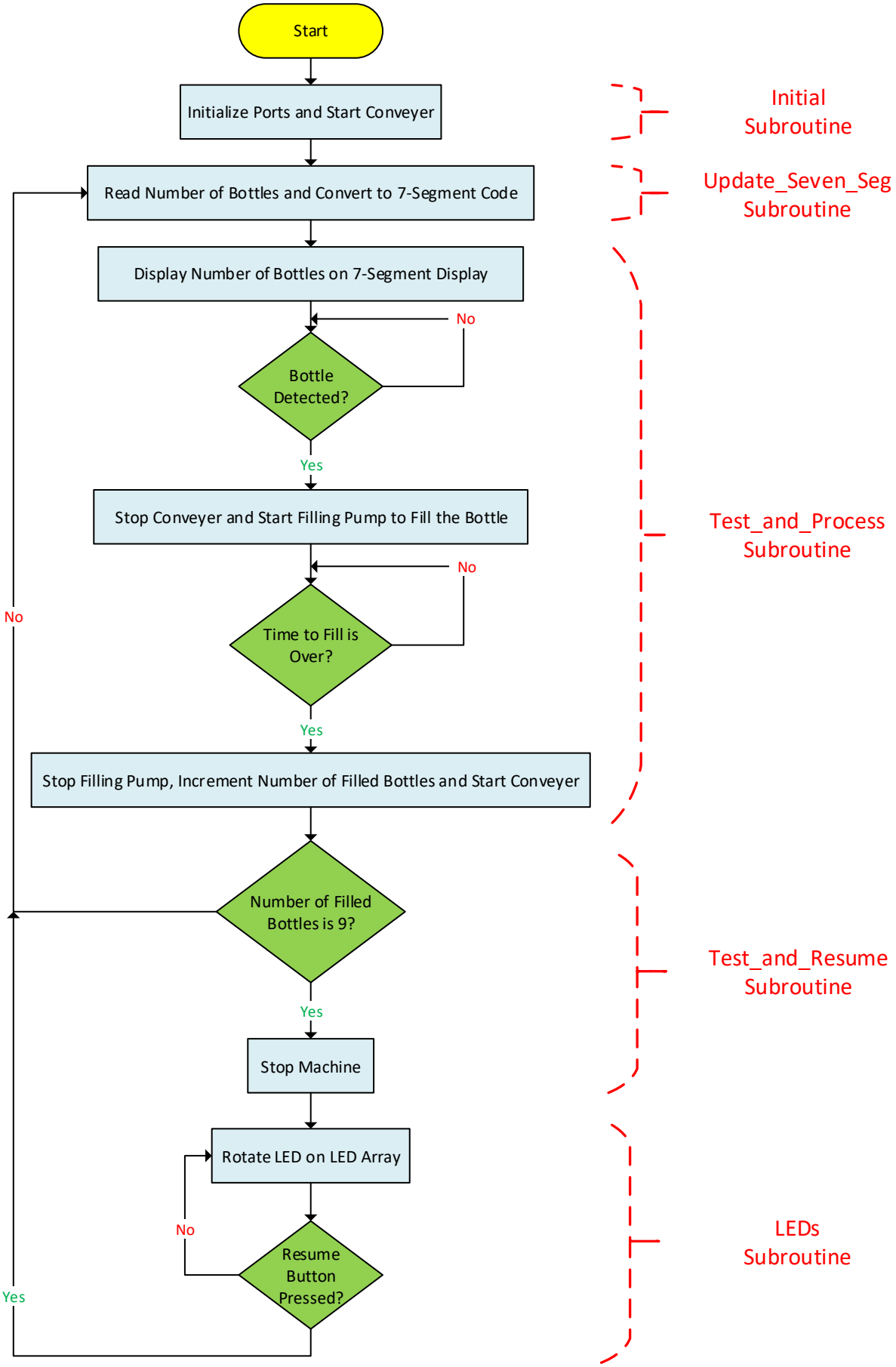


Figure 2. Overall flowchart of the system.

In addition to these subroutines, we will need to generate time delay to wait for the filling pump to fill the bottle and to move the LED on the LEDs array. For this purpose, we will define the **Simplest_Delay** subroutine

Based on our analysis and design, the code of the **Main** program is given below. Note the sequence of calling the subroutines and how the operation is repeated indefinitely. In the following, we will discuss each of these subroutines in details.

```
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
```

The Initial Subroutine

This subroutine is primarily used for configuring the ports as required and initializing the variables. The code is given below.

```
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
```

The 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. Clearly, the signals sent to the 7-Segments display are not decimal values, but according to the 7-Segment layout (Refer to the Hardware Guide for more information.). Accordingly, we have to convert 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 7-segment representations of the decimal number 0-9 as constants and use a Look-up table to get the correct representation for each bottle number.

```
Zero    equ   B'11000000'
One     equ   B'11111001'
Two     equ   B'10100100'
Three   equ   B'10110000'
Four    equ   B'10011001'
Five    equ   B'10010010'
Six     equ   B'10000010'
Seven   equ   B'11111000'
Eight   equ   B'10000000'
Nine    equ   B'10010000'
```

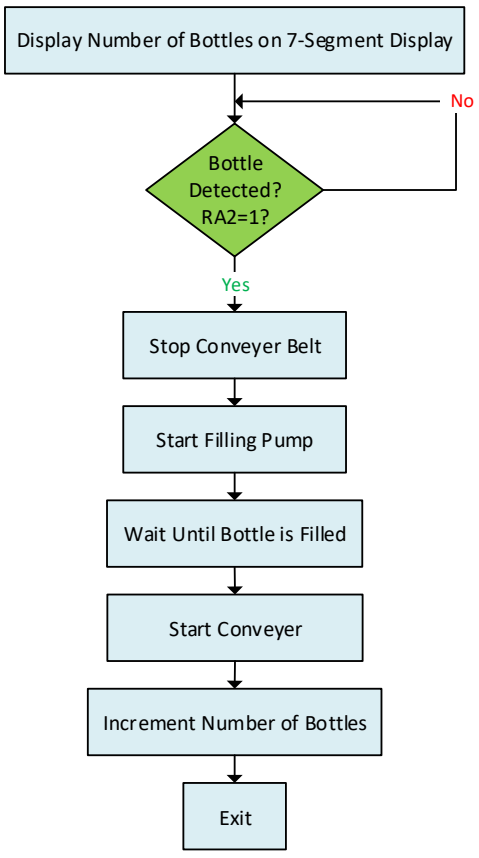
```
Update_Seven_Seg
        Movf   BottleNumber, W
        Addwf  PCL, F
        Retlw  Zero
        Retlw  One
        Retlw  Two
        Retlw  Three
        Retlw  Four
        Retlw  Five
        Retlw  Six
        Retlw  Seven
        Retlw  Eight
        Retlw  Nine
```

Figure 3. The Update_Seven_Seg Subroutine and the definition of the seven segment codes.

In our design, the 7-segment display is common-anode and is connected to PORTD such that RD0 is connected to segment a, RD1 is connected to segment b, and so on. So, in the beginning of our program we define a set of 10 constants for the 7segment codes and assign them the values as shown in Figure 3.

The Test_and_Process Subroutine

This subroutine displays the current bottle count on the 7-segment display and tests if a bottle is present or not. If a bottle is detected, the conveyor motor is stopped, the filling pump starts working for a specified period of time to fill the bottle and then stops. Afterwards, the conveyor belt starts moving again. Finally, the number of bottles is incremented by one. Figure 4 shows the flowchart of this subroutine and the corresponding code. Note how the subroutine starts with the movwf PORTD to output the 7-segment code to PORTD. The code is already in the Working register after calling the **Update_Seven_Seg** subroutine.

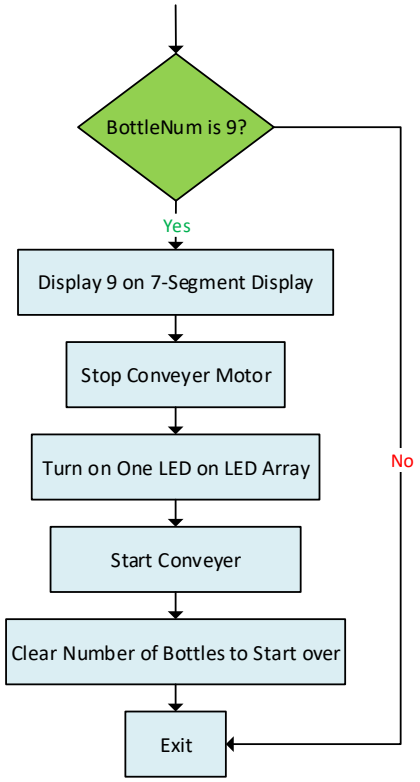



```
Test_and_Process
        movwf    PORTD          ; display on the 7-Seg
poll    btfss    PORTA,2        ; Test the bottle sensor
        goto     poll
        bcf      PORTC,0        ; stop conveyer motor
        bsf      PORTC,1        ; start filling motor
        call     Simplest_Delay ;Insert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

Figure 4. Test_and_Process Subroutine.

The Test_and_Resume Subroutine

This subroutine checks if the total number of bottles has reached nine. If not, it will exit and return to the main program to continue the operation. Otherwise, it stops the conveyer motor to package the filled bottles manually. At this moment, 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. These last two operations are performed using the **LEDs** subroutine that is called inside this subroutine. The flowchart and the code for the **Test_and_Resume** subroutine is given Figure 5.




```
Test_and_Resume
        movf    BottleNumber, w
        sublw   .9
        btfss   STATUS, Z
        goto    fin1            ; return
        call    Update_Seven_Seg
        movwf   PORTD           ; display on the 7-seg
        bcf     PORTC, 0        ; stop conveyer motor
        bcf     STATUS, C
        clrf    BottleNumber    ; Reset System
        call    LEDs            ; rotate LEDs and check resume
                                ; button
fin1
    return
```

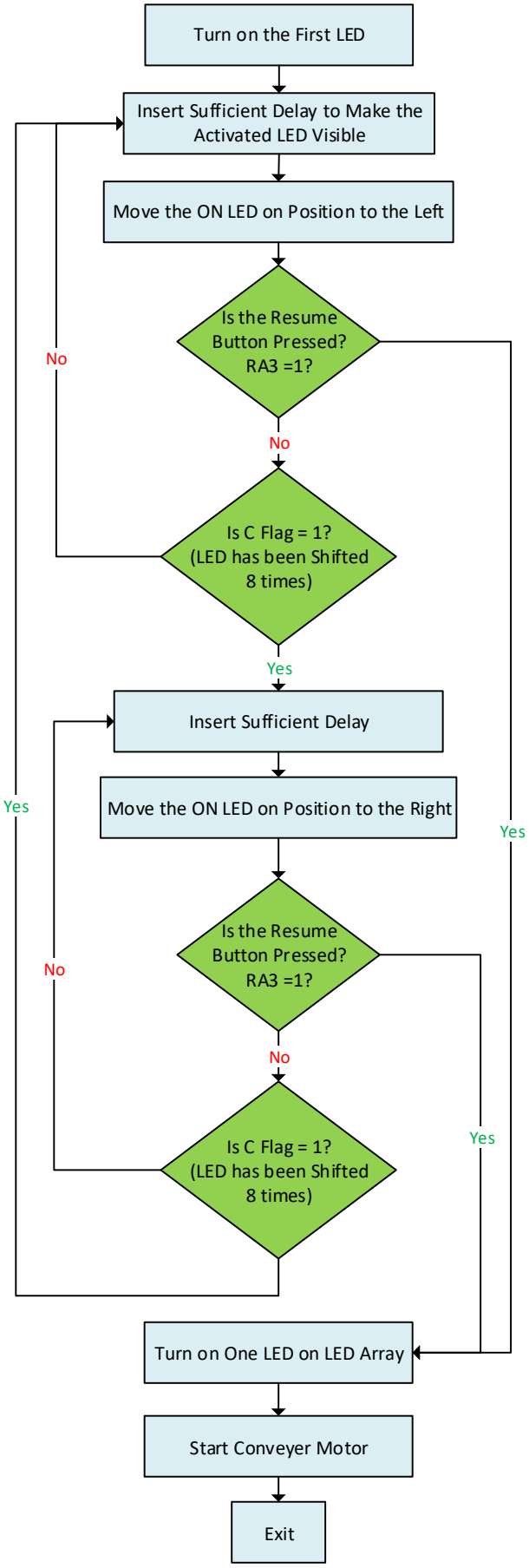
Figure 5. Test_and_Resume Subroutine.

The LEDs Subroutine

This subroutine lights one LED on an 8-LED-row and continuously moves it back and forth in this fashion. In between, it checks the resume button. If pressed, the conveyor motor starts again and the LED array turns off; else the LEDs keep rotating and the resume button checked. Figure 6 shows the flowachart and code for this subroutine.

The Simplest_Delay Subroutine

This subroutine is used to generate a time delay for filling the bottle and controlling the motion of the LED on the LED array. It is composed of two nested loops that decrement two counters; as we learned in class. The code for this subroutine is shown in Figure 7.




```
LEDs
        bsf     PORTB, 0 ; turn on the 1st LED
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  ; turn off LED array
        bsf     PORTC, 0 ; start conveyer
        motor
        return
```

Figure 6. LEDs Subroutine.

```
Simplest_Delay
        movlw     0xFF
        movwf     MSD
        clrf      LSD
loop2
        decfsz    LSD, F
        goto      loop2
        decfsz    MSD, F
        goto      loop2
        return
```

Figure 7. The Simplest_Delay Subroutine.

4. How to Simulate This Code in MPLAB?

Step4: System Testing and Debuggin

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**. The Stimulus option is available in the Debugger menu as shown in Figure 8. When you select New Workbook, a new window will appear where you can add the pins to stimulate and specify type action to happen on this pion when stimulated.

In our system, we are observing two external inputs: The first one is the bottle sensor which is connected to RA2. We will assume that the sensor generates a positive pulse when it detects a bottle. The second input is the Resume button which is connected to RA3. Also, it is assumed that it generates a positive pulse when it is pressed. In order to activate these inputs while simulating the program, follow the following steps:

1. Select Debugger → Stimulus → New Workbook as shown in Figure 8. A stimulus window opens

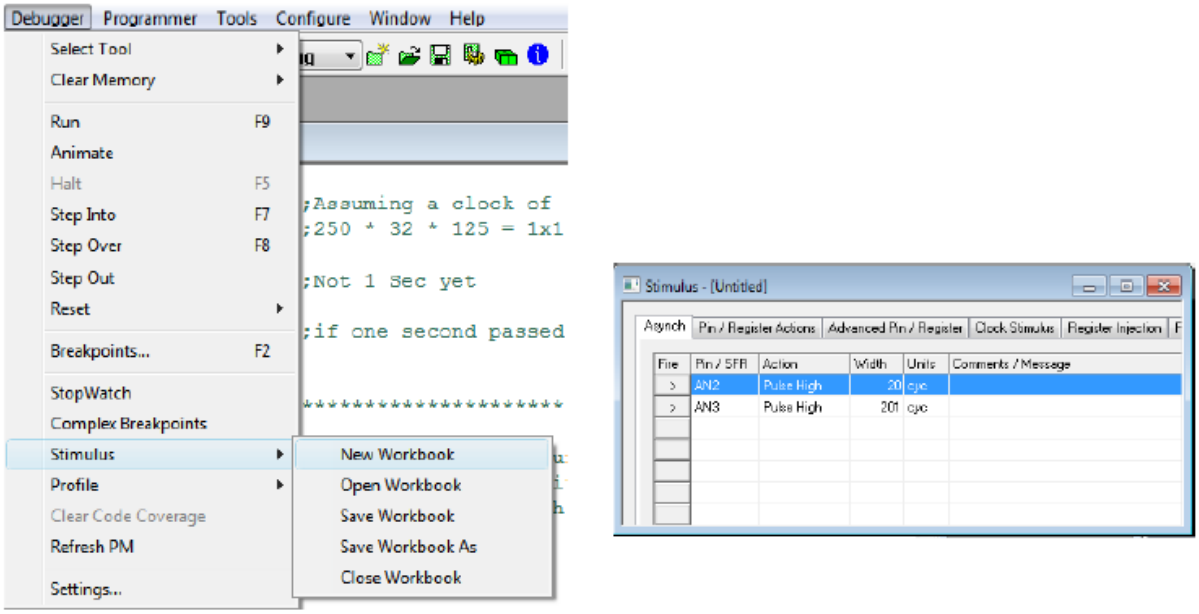


Figure 8. Stimulus window.

2. Click on a cell under Pin/SFR and select the pin that you want to stimulate. Add AN2 and AN3 which correspond to RA2 and RA3, respectively. For both pins, specify the action to be Pulse High under the Action column, under the Width column specify the width of the pulse to be 20, and under the Unit write 20 (Check Figure 8). This implies that this pin will receive a positive pulse with duration of 20 processor cycles when activated.
3. In the code of the program, place a breakpoint at the instruction BTFSS PORTA, RA2 inside the Test_and_Process subroutine. This will allow us to change the reading of the bottle sensors during program simulation.
4. In the code of the program, place a breakpoint at the instruction BTFSS PORTA, RA3 inside the LEDs subroutine. This will allow us to change the reading of the bottle sensors during program simulation.
5. Run your code, you will go to the First break point then press "Step Into" you will observe that you have stuck in loop.
6. Now Press "Fire", the arrow next to the RA2 in the Stimulus pin. What do you observe?
7. Now press "Step Into" again, observe how the value of BottleNumber change.
8. Press "Run" then "Fire" again, observe how the value in BottleNumber changes whenever you reach the first breakpoint. Keep doing this until the program stops at the second breakpoint inside the LEDs subroutine.
9. Press "Step Into" you will observe that you have stuck in loop.
10. Now Press "Fire", the arrow next to the RA3 in the Stimulus pin.
11. Now press "Step Into" again, observe how the value of BottleNumber changes to ZERO.

5. Simulation Using Proteus

Proteus PIC Bundle is the complete solution for developing, testing and virtually prototyping your embedded system designs based around the Microchip Technologies TM series of microcontroller. This software allows you to perform *schematic capture* and to *simulate* the circuits you design. Please to refer to the "Introduction to Proteus.pdf" file to learn how to use Proteus to simulate the filling machine example.

The complete code that control the system can be found in the "Experiment 3 Filling Machine Code.asm" file and the Proteus circuit for the system is available in the "Experiment 3 Filling Machine Proteus Circuit" file.

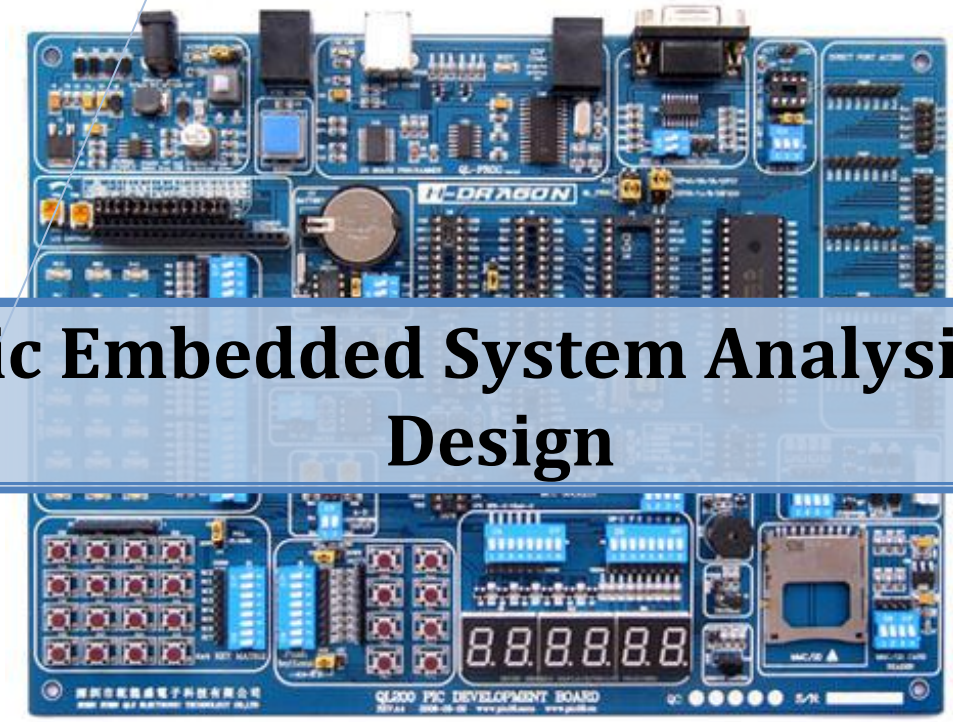
6. Building the Hardware of the System

Once you have designed your system and tested it, you will need to build it using actual hardware components. This usually requires knowledge about the hardware components and how they can be used, in addition to good fundamentals in circuits and electronics. The file "Guide to Hardware.pdf" outlines the basic hardware components and some technical issues that you usually need to know in order to build simple build embedded systems.

In order to give you some experience in this, the second part of this experiment is a simple exercise on how to wire up a simple embedded system. The details of this part are in Labsheet 3B.pdf file.

Labsheet 3

University of Jordan
School of Engineering
Department of Computer Engineering
Embedded Systems Laboratory 0907334



Basic Embedded System Analysis and Design

Name:

Student ID:

Section (Day/Time):

COMPUTER NAME:

UNIVERSITY OF JORDAN
SCHOOL OF ENGINEERING
DEPARTMENT OF COMPUTER ENGINEERING
EMBEDDED SYSTEMS LABORATORY CPE0907334

Labsheet3: Basic Embedded System Analysis and Design

Name: **Student ID:**
Section:

Note: Before you start, you should read the tutorial on how to use The Proteus Program and The Guide to Hardware Tutorial that are available on the website.

In this experiment, you are required to use a PIC16F877A microcontroller to design a simple embedded system that has eight light emitting diodes (LED) two pushbuttons (START and INCREMENT). The anodes of the LEDs are connected to PORTC while the pushbuttons are connected to RB0 and RB1 using pull-down resistors (Refer to the Guide to Hardware Tutorial).

The system supposed to control the LEDs as follows:

1. When the system starts, the LEDs are off and the system waits the user to press the START pushbutton. Nothing will happen until the user presses the START button.
2. When the START pushbutton is pressed, the system will turn ON all the LEDs for some time and then turn them off. Afterwards, it starts monitoring the second pushbutton which we will call the INCREMENT button.
3. Whenever this button is pressed, the system increments internal variable VALUE by 50 and displays it on the LEDs.
4. When the VALUE is greater than 155, the system flashes all LEDs 3 times, resets VALUE and restarts its operation; i.e. it will wait for the START button to be pressed.

(Pre-lab)

1) Determine the required hardware and assign I/O pins.

Inputs:	
Outputs:	

2) The Initial and Main Codes:

Your code should have at least 2 subroutines: 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 on the system start up. The Main subroutine contains all the subroutines which perform the functions of the system. In the space below, write the code of the Initial Subroutine.

Initial

3) **In addition to the Initial and Main subroutines, you will need a delay subroutine to control the flashing of the LEDs when VALUE is greater than 50. You can use the Simplest_Delay subroutine given in Experiment 3. Write the code of this subroutine in the space below.**

Simplest_Delay

4) **After analyzing the system operation, draw the flowchart(s) of the main program and the new subroutines that you will use.**

In Lab

1)Write the code all subroutine and the main program. Note: The nature of the system requires it runs continuously, the program code will loop through specific subroutines which implement the system function.

In the space below, copy and paste the whole code of your program.

2) In order to test your program, you will use the Proteus circuit simulator. In this simulator, you can draw the circuit of your system and load your code to test it. Please refer to the Introduction _to _ Proteus tutorial that is available on the website.

For this experiment, we have prepared the circuit of the system in the Labsheet 3 Proteus file. You can use this file to test your program; however, make sure to install the program on your computer and practice building circuits in Proteus.

***Note:* You might need to insert a small delay in your code after you read the pushbuttons to avoid reading them more than one time during simulation. You may use the Simplest Delay subroutine or write a new one.**

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.

Written by Eng. Ashraf Suyyagh and Eng. Enas Jaara – Revised by Prof. Iyad Jafar

Introduction

A ***L***iquid Crystal Displays (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. Figure 1 shows a typical LCD module. This experiment teaches you about LCDs which are based upon the Hitachi HD44780 controller chipset.

LCDs come in different shapes and sizes in terms of number of characters and lines. Typical LCDs may have 8, 16, 20, 24, 32, and 40 display characters that are arranged in 1, 2 or 4 lines. **However, all regardless of the external shape of the LCD, they are internally built as a 40x2 format as shown in Figure 2.** Each of the small rectangles can be used to display some character as we will see later. The numbers shown in the small rectangles are the corresponding RAM hexadecimal addresses that you need to write to in order to write to that character. Figure 3 shows the relation between the character position and its RAM address in the HD44780 controller based LCDs.

Figure 1: A typical LCD module.

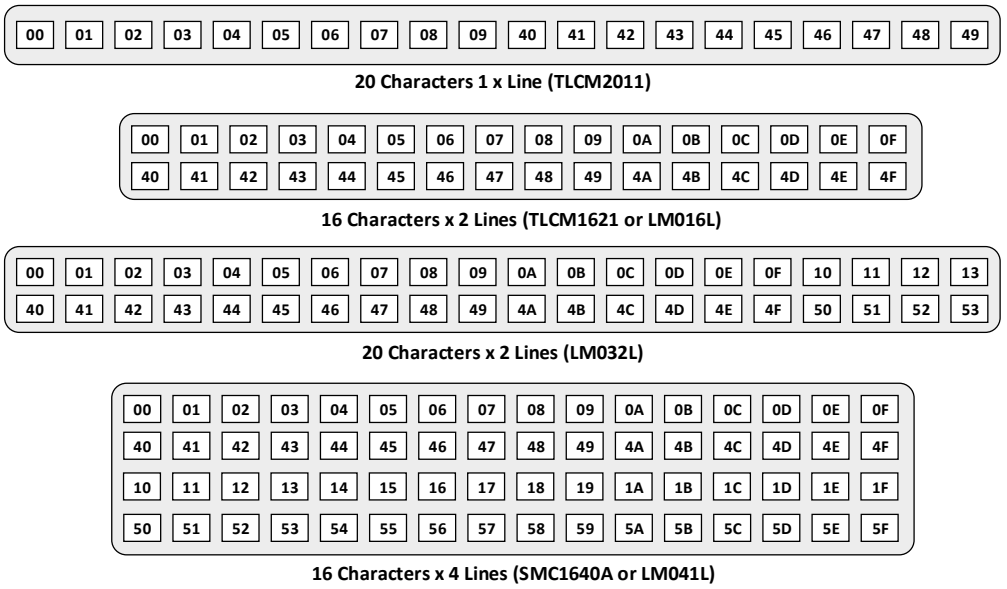


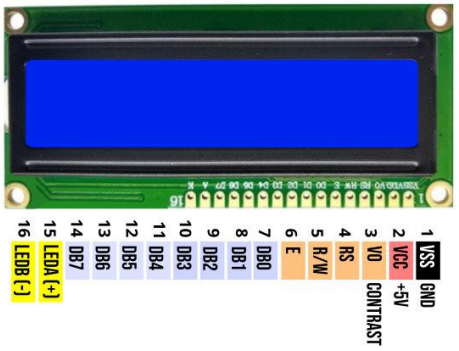
Figure 2: Different LCD modules shapes and sizes.

Display Position (Decimal)	1	2	3	4	5	6	…	39	40
RAM Address (HEX)	00	01	02	03	05	06	…	26	27
	40	41	42	43	45	46	…	66	67

Figure 3: Display address assignments for HD44780 controller based LCDs.

LCD Pin Out

Most LCD modules conform to a standard interface specification. A 16-pin access is provided with *eight data lines*, *three control lines*, three power lines and two additional pins (L+ and L-) that are typically used for backlight control purposes. Figure 4 shows the pinout of a 16-pin LCD and their description. Note that, some LCDs are 14-pin and don't have the L+ and L- pins.



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 4: 16-Pin LCD pinout and description.

Note: The LCD in Figure 4 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 the LCD

Powering up the LCD requires connecting three lines: one for the positive power ***VDD*** or ***VCC*** (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. In this experiment, we will connect this pin to ground. As you can see from Figure 4, the LCD connects to the microcontroller through three control lines: ***RS***, ***RW*** and ***E***, and through eight data lines ***D0*** through ***D7***.

When powered up, the LCD display shows 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)

Interfacing LCD to PIC

Figure 5 shows an example of interfacing a 14-pin LCD to the PIC16F877A microcontroller. The data pins are connected to PORTD, RA1 is connected to ***RS***, RA2 is connected to ***R/W*** and RA3 is connected to ***E***. Of course, you can use different pin assignment to interface the LCD. **Note:** in this experiment, we will be only writing to the LCD, so the ***R/W*** input is fixed to 0. Reading from the LCD is left to the students as exercise.

Communicating with the LCD

Using an LCD is a simple procedure once you learn it. Simply, 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?**

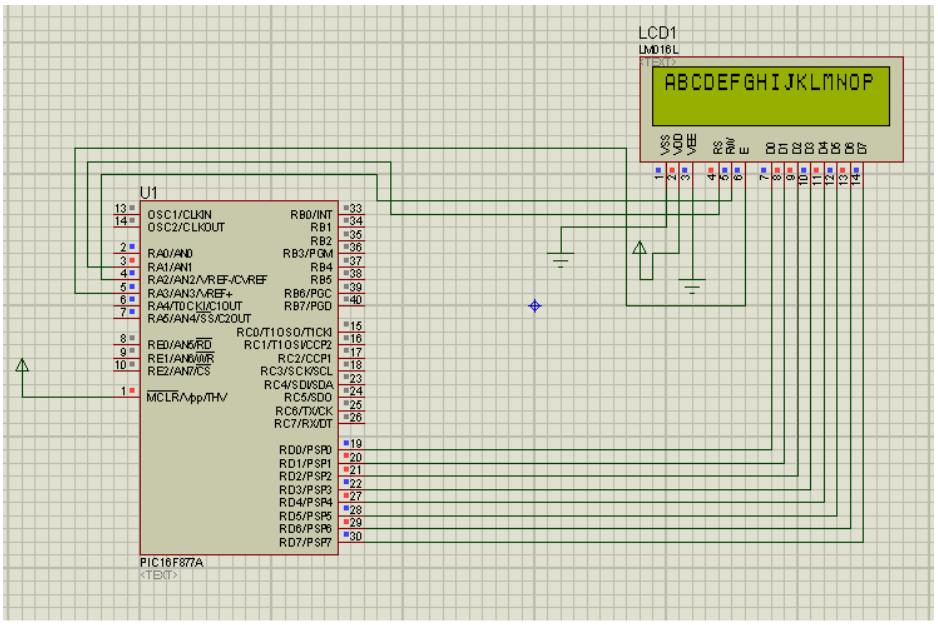


Figure 5: A typical interfacing between a PIC16F877A and an LCD module

For this purpose, the ***Register Select (RS)*** input is used. When this input is set to 1 and the ***Enable (E)*** input transitions from high to low, the LCD interprets the value on ***D0-D7*** as data. On the other hand, when ***RS*** is 0 and ***E*** transitions from high to low, the LCD considers the value on the data pins as a command. Figure 6 shows this simple operation. To implement the operation of the operations shown in Figure 6 in software, we will use two subroutines: **send_char** and **send_cmd**.

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 6: Necessary control signals for Data/Commands

The **send_char** subroutine is supposed to perform the following operations to send the data (character) to the LCD:

1. Output the data to D0-D7 lines. It is assumed that the data is in W and the data pins are connected to PORTD.
2. Set Register Select (RS) to **1** to tell the LCD that we are outputting data. RS is connected to RA1, so RA1 should be set to **1**.
3. Generate a falling edge on the Enable (E) input to trigger the LCD to read the data pins. The E input is connected to RA3. So, RA3 is set 1 for some time then set to 0 after some delay.
4. Generate a delay to give LCD the time needed to display the character.

Accordingly, the code for this subroutine is as follows:

```
send_char
        movwf  PORTD
        bsf    PORTA, 1
        bsf    PORTA, 3
        nop
        bcf    PORTA, 3
        bcf    PORTA, 2
        call   delay
        return
```

Similarly, the **send_cmd** subroutine is supposed to perform the following operations to send a command to the LCD:

1. Output the command to D0-D7 lines. It is assumed that the data is in W and the data pins are connected to PORTD.
2. Set Register Select (RS) to **0** to tell the LCD that we are outputting data. RS is connected to RA1, so RA1 should be set to **0**.
3. Generate a falling edge on the Enable (E) input to trigger the LCD to read the data pins. The E input is connected to RA3. So, RA3 is set 1 for some time then set to 0 after some delay.
4. Generate a delay to give LCD the time needed to execute the command.

Accordingly, the code for this subroutine is as follows:

```
send_cmd
        movwf   PORTD
        bsf     PORTA, 0    ; the only difference from the send_char subroutine
        bsf     PORTA, 3
        nop
        bcf     PORTA, 3
        bcf     PORTA, 2
        call    delay
        return
```

Note that the only difference between the two subroutines is in the highlighted instruction that controls the value of the ***RS*** input.

Displaying Characters on the LCD

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.

Figure 7 shows the character map for the LCD that we are using in this experiment. 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). On the other hand, 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.

Sending Commands to the LCD

There are many commands that you need to be aware of in order to control the LCD. The list, format, and options of these commands is shown in Figure 8. All the commands are 8-bit. To issue any of these commands, we determine the values of its parameter(s) and then issue the **send_cmd** command. In the following, we will discuss these commands in details.

Clear Display Command

Moving the value 01 to the working register followed by call send_cmd will clear the LCD display. However, the cursor will remain at its 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.

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

Figure 8: LCD command control codes.

Display and Cursor Home Command

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 Command

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, i.e. the D1 bit in the command should be 0.
- **S:** Accompanies the D/C parameter that is explained later.

Display On/OFF and Cursor Command

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). In other words, this command activates the LCD in order to be ready to use.
- **U:** This command displays the cursor in the form of a horizontal line at the bottom of the character when its value 1 and turns the cursor off when it is 0.
- **B:** If the underline cursor option is enabled, this will blink the cursor if high.

Display/Cursor Shift Command

All LCDs based on the HD44780 format - whatever their actual physical size is - are internally built to be 40 characters x 2 lines with the upper row having the display addresses $0\text{-}27_H$ (40 Characters) and the lower row from 40_H -67_H (40 characters).

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 14_H, 15_H, 16_H, … , 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 has 20 **visible** Characters wide from the outside and 40 from the inside. All you have to do is shift the display. To do that, use the Display/Cursor Shift command as follows:

1. Determine the direction of the shift using the (R/L) bit. This bit controls the direction for shift (Right or Left).
2. Specify the value of the D/C bit in the command. If this bit is 0, the display is not shifted and the cursor moves one position to right or left according to R/L bit. If this bit is 0, the display is shifted to right or left to show the hidden characters. Note that you need issue this command multiple times in order to shift the display by multiple locations!

Function Set

This command controls different features of the LCD and it has three parameters:

- **8/4:** this parameter specifies whether the LCD is receiving data as eight bits or four bits. In 8-bit mode, the data is sent as 8 bit on D0-D7 lines. When 4-bit is used, the data is sent on D4-D7 lines in two stages. The 4-bit is useful when interfacing the LCD to save output pins of the microcontroller.
- **2/1:** this parameter specifies the line mode. When it is set to 1, then you can use both lines on the LCD. Otherwise, you can use the upper line only.
- **10/7**: this parameter control height of the displayed character, i.e. the Dot format. The LCD supports 5x7 or 5x10 format such that:
    - 5x7 format (Default) is used whenever you use the characters found in columns 1 to D in the character map shown in Figure 7.
    - 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 the character map in Figure 7.

Set Display Address Command

This command allows you to move the cursor to whichever location you want. The syntax of the command is 1AAAAAAA. The A's in the command are used to specify the address of the character position on the display. For example, suppose you want to start writing in the middle of 20x2 display (***visible*** width of the LCD screen is 20), then, from Figure 2 you will observe that location $0A_H$ is approximately in the middle, so

you replace the A's with $0A_H$ and the command becomes $(1\mathit{0001010})_2$ or 8 A_H. Another example is when you want to start writing starting at the second line, which is location 40_H, then, then the you should issue the command is $(1\mathit{1000000})_2$ or $C0_H$. With the command in hand, you can put it in W and send it to the LCD and calling the send_cmd subroutine.

Example

The following code is an example of initializing the LCD and using it to display the 26 English characters. Note the initialization of the LCD in the Initial subroutine. After initialization, the program starts by loading letter A in the working register and sending it to the LCD. Afterwards, it loops through the other characters by adding 1 to the tempChar variable to move to the next English character. This operation is performed in main program in the **CharacterDisplay** loop. Once all characters are sent to the LCD, you will see the first 20 characters only since the display is 20x2. To see the remaining characters, you need to shift the screen continuously. This is done by issuing the shift display command infinitely in the **MainLoop** loop in the main program.

```
1  ;*******************************************************************************
2  ;                                   EXAMPLE CODE 1
3  ;*******************************************************************************
4  ; This code displays on the upper row of the LCD the 26 English letters in alphabetical order
5  ; The code starts with LCD initialization commands such as clearing the LCD, setting modes and
6  ; display shifting.
7  ;
8  ; Outputs:
9  ;       LCD Control:
10 ;                           RA1: RS (Register Select)
11 ;                           RA3: E  (LCD Enable)
12 ;       LCD Data:
13 ;                           PORTD 0-7 to LCD DATA 0-7 for sending commands/characters
14 ; Notes:
15 ;       The RW pin (Read/Write) - of the LCD - is connected to RA2
16 ;       The BL pin (Back Light) – of the LCD – is connected to potentiometer
17 ;*******************************************************************************
18        include         "p16f877A.inc"
19 ;*******************************************************************************
20        cblock          0x20
21              tempChar        ;holds the character to be displayed
22              charCount       ;holds the number of the English alphabet
23              lsd             ;lsd and msd are used in delay loop calculation
24              msd
25        endc
26 ;*******************************************************************************
27 ; Start of executable code
28              org     0x000
29              goto    Initial
30 ;*******************************************************************************
31 ; Interrupt vector
32 INT_SVC      org     0x0004
33              goto    INT_SVC
34 ;*******************************************************************************
35 ; Initial Subroutine
36 ; INPUT:      NONE
37 ; 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         ; configure 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
```

```
92  ;*******************************************************************************
93  send_char
94                  movwf   PORTD           ; Refer to table 1 on Page 5 for review of this subroutine
95                  bsf     PORTA, 1
96                  bsf     PORTA, 3
97                  nop
98                  bcf     PORTA, 3
99                  bcf     PORTA, 2
100                 call    delay
101                 return
102 ;*******************************************************************************
103 delay
104                 movlw   0x80
105                 movwf   msd
106                 clrf    lsd
107 loop2
107                 decfsz  lsd,f
109                 goto    loop2
110                 decfsz  msd,f
111 endLcd
112                 goto    loop2
113                 return
114 ;*******************************************************************************
115                 End
116
```

Generating and Storing Custom Characters Using the CGRAM

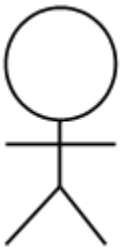
The character map of the LCD that is shown in Figure 7 contains the English and Japanese characters, numbers, symbols and special characters. Now, suppose you want to display a custom character that is not shown in the table such as an Arabic letter, i**s that possible?**

If you check the character map in Figure 7, you will see that the locations under column 0 are labeled with CG(1) to CG(8). These locations are reserved to store custom characters that a user may wish to use, such as the Arabic characters. Notice that despite that there are 16 locations under column 0, you can only use the first 8 locations as the remaining are mirrors of the first.

Given these locations, how can we generate the characters and store them?

The **first step** is to draw the character. This is actually a fun thing to do. Simply, draw a 5x8 grid as shown in Figure 9(a) and then start drawing your character inside it by shading the cells. Figure 9(b) shows and example of drawing a stickman character inside the grid. Next, add three columns to the gird and replace put 1 in the shaded squares and 0 in empty squares as shown in Figure 9(c). Each of the rows in the 8x8 grid is a byte that is to be stored in the LCD memory. The hexadecimal value of each row is shown in Figure 9(d).

Once we have the values to be stored, the **second step** is to store it in one of the CG RAM locations. To do so, we need to specify which of the CG locations we want to use. This is done using the **Set CG-RAM Address command**. The syntax of this command is 01*AAAAAA*, where AAAAAA is the starting address of the CG location.



(a)

B4	B3	B2	B1	B0
	■	■	■	
■				■
	■	■	■	
		■		
■	■	■	■	■
		■		
	■		■	
■				■

(b)

B7	B6	B5	B4	B3	B2	B1	B0
0	0	0	0	1	1	1	0
0	0	0	1	0	0	0	1
0	0	0	0	1	1	1	0
0	0	0	0	0	1	0	0
0	0	0	1	1	1	1	1
0	0	0	0	0	1	0	0
0	0	0	0	1	0	1	0
0	0	0	1	0	0	0	1

(c)

Value (H)
0x0E
0x11
0x0E
0x04
0x1F
0x04
0x0A
0x11

(d)

Figure 9: Steps to draw a stickman.

For example, if we want to store the character we generated previously in CG(0), we issue the command 01*000000* or 0x40 to the LCD using the **send_cmd** subroutine. However, if we want to store it in CG(1), we need to skip eight locations and issue the command 01*001000* or 0x48. This is necessary as each character is actually eight bytes, so the next character in the CG RAM starts after eight bytes. Figure 10 illustrates this idea. Note how CG(2) starts at 0x48 and CG(7) starts at 0x78.

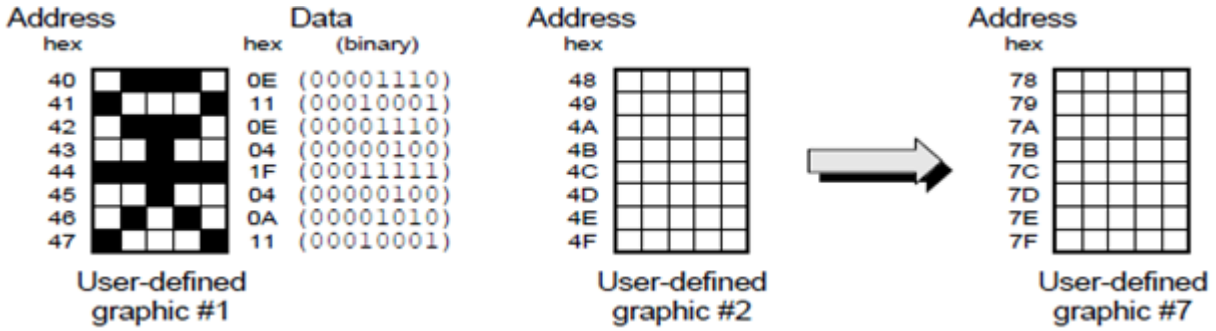


Figure 10: Storing the character in CG RAM.

The **third step** after specifying the CG RAM address is to send the 8 bytes that we obtained using the send_char subroutine as shown in the code below.

```
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
```

Once the character is stored in the CG RAM, we can display it on the LCD by simply calling the send_char subroutine after placing the address of the CG character in the working register.

The following example shows how to store two stickman characters and display them on the LCD in animated fashion as if the stickman is moving. There are two subroutines to store the two characters in the CG RAM; DrawStick1 and DrawStick2. Once the characters are stored, the program continuously shows them at the same position on the LCD in the Main loop. Note how the LCD has to be cleared after storing the characters in the CG RAM. Also, note how we send the command 0x02 after displaying each character in order to move the cursor to the first position.

```
;*****************************************************************************
**
;                                   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
```

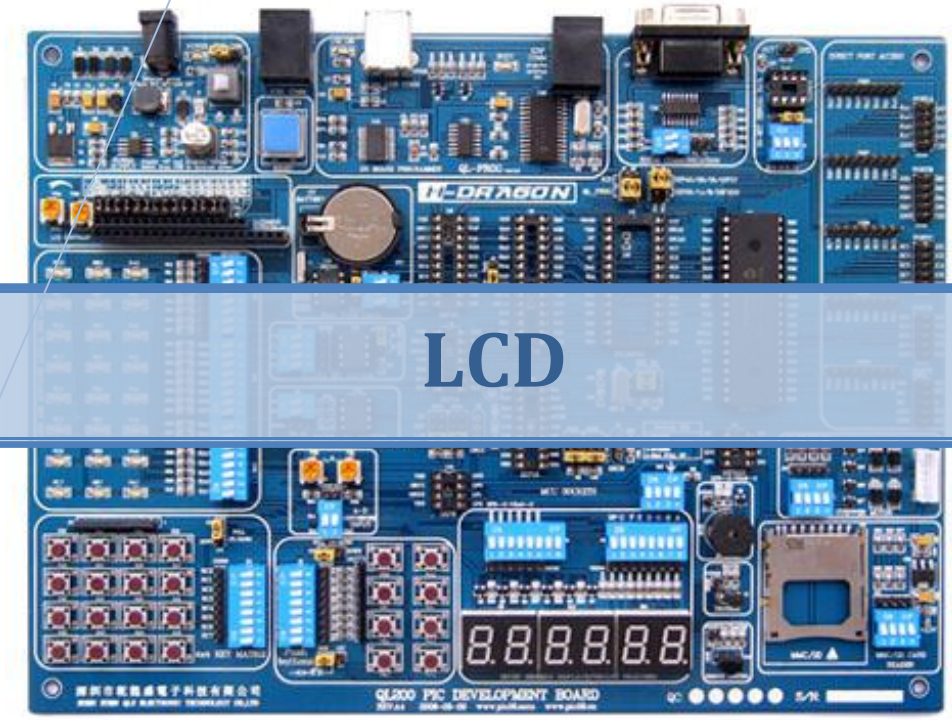
```
; RESULT:       Configure I/O ports (PORTD and PORTA as output, PORTA as digital)
;               Configure LCD to work in 8-bit mode, with two lines of display and 5x7 dot format.
;               Set the cursor to the home location (location 00), set the cursor to the visible state
;               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
                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
```

```
99                  bcf     PORTA, 3
100                 bcf     PORTA, 2
101                 call    delay
102                 return
103 ;*****************************************************************************
104 **
105 send_char
106                 movwf   PORTD           ; Refer to table 1 on Page 5 for review of this subroutine
107                 bsf     PORTA, 1
108                 bsf     PORTA, 3
109                 nop
110                 bcf     PORTA, 3
111                 bcf     PORTA, 2
112                 call    delay
113                 return
114 ;*****************************************************************************
115 **
116 delay
117                 movlw   0x80
118                 movwf   msd
119                 clrf    lsd
120 loop2
121                 decfsz  lsd,f
122                 goto    loop2
123                 decfsz  msd,f
124 endLcd
125                 goto    loop2
126                 return
127 ;*****************************************************************************
128 **
129 DrawStick1                          Setting the CGRAM address at which we draw the stick
130 man
131                 Movlw   0x40            ; Here it is address 0x00 in Figure 8 which transforms
132                 Call    send_cmd        ; into command 0x40
133                 Movlw   0X0E            ;Sending data that implements the Stick man
134                 Call    send_char
135                 Movlw   0X11
136                 Call    send_char
137                 Movlw   0X0E
138                 Call    send_char
139                 Movlw   0X04
140                 Call    send_char
141                 Movlw   0X1F
142                 Call    send_char
143                 Movlw   0X04
144                 Call    send_char
145                 Movlw   0X0A
145                 Call    send_char
146                 Movlw   0X11
147                 Call    send_char
148                 Return
149 ;*****************************************************************************
150 **
151 DrawStick2                          ;Setting the CGRAM address at which we draw the stick
```

```
152 man
153                 Movlw  0x48            ;Here it is address 0x01 in Figure 8 which transforms
154                 Call     send_cmd      ; into command 0x48
155                 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
```

Labsheet 4

University of Jordan
School of Engineering
Department of Computer Engineering
Embedded Systems Laboratory 0907334



LCD

Name:

Student ID:

Section (Day/Time):

COMPUTER NAME:

UNIVERSITY OF JORDAN
SCHOOL OF ENGINEERING
DEPARTMENT OF COMPUTER ENGINEERING
EMBEDDED SYSTEMS LABORATORY CPE0907334

Labsheet 4: LCD

Name: **Student ID:**
Section:

(Pre-lab) Part 1: Code Analysis and Understanding:

Answer the following questions regarding Code1:LDC1.asm of the experiment.

1. What changes to the instruction at line 57 are necessary in order for the cursor to be blinking?

2. What change(s) would you make to the code in order for it to start displaying the characters at approximately the middle of the visible screen (i.e. at address 6)

(Pre-lab) Part 2: Suppose that you will store your Arabic characters at CG RAM Locations 0, Location 1, Location 2 and location 3. Complete the modified main subroutine down with the appropriate values in the Movlw instructions.

```
Main
        Movlw   _________       ;Display character stored in location 00 which in
        Call    send_char       ;this case is your first charcter in CG-RAM
        Movlw   _________
        Call    send_char
        Movlw   _________
        Call    send_char
        Movlw   _________
        Call    send_char
        Goto    Main            ; This loop makes the character rotate continuously
```

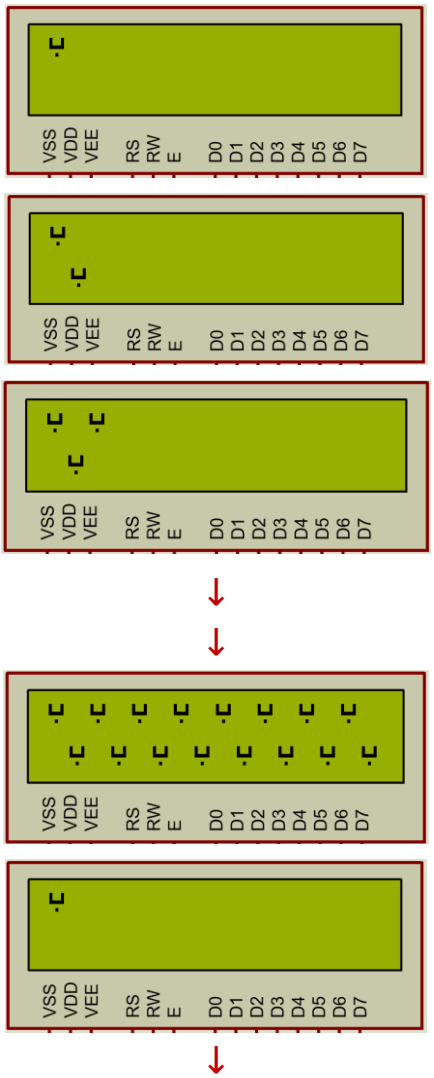
(Pre-lab) Part 3: On the grid given below, shade the squares required to generate the first letter of your name in Arabic. If the first letter of your name is (أ), generate the character for the second letter in your name. Then, write the instructions needed to store this character in CG(1).

Draw your character below						Replace each shaded cell with one and not shaded ones with zero.									Data in Hex
B4	B3	B2	B1	B0		B7	B6	B5	B4	B3	B2	B1	B0		
						0	0	0							0x
						0	0	0							0x
						0	0	0							0x
					→	0	0	0							0x
						0	0	0							0x
						0	0	0							0x
						0	0	0							0x
						0	0	0							0x

```
; Code to store the first letter of your name in Arabic
```

Part 3: Code Modification

In this part, you are required to modify the code in LCD2 - Animating Stickman.asm to display the first Arabic letter of your name in a zigzag fashion as shown in the figure below. When the last position is reached on the second row, the screen is cleared and the operation is repeated. You can use the Embedded_Ex4 Proteus file to test your code. Assume that you are using 16x2 display.



Hints:

1. You will need to use the **Set Display Address** command to select the display location. Check Figure 2 in the tutorial to know the addresses of the locations to be used.
2. For proper display of the pattern, it is advised to turn off the cursor when you initialize the LCD in the Init subroutine.
3. To reduce the program size, you may write a lookup table that stores the display addresses and call this table in a loop.

Ask your engineer to check the run.

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

5

Experiment 5: 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

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 for large and very complex programs due to their simplicity which allows for faster program development, 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 have the disadvantage of higher program memory requirement. This is crucial as most microcontrollers have limited memory space. Additionally, longer codes imply longer time to execute. However, 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 hardware limitations. Hence, it is common for programmers combine C and Assembly language in the same developed source code.

There are many C compilers available commercially, such as mikroC, CCS and HI-TECH. 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.

The HI-TECH C Compiler

In order to use the HI-TECH C compiler to write your PIC programs in C language, it is assumed that you have already installed it during the installation of MPLAB. Also, when write your program in high-level C, you need to save the source code in a file with .C extension.

Given that, you can create a project in MPLAB following the same steps that you learned in Experiment 0. However, in **Step Two**, you need to select the HIGH-TECH Universal ToolSuite as the Active ToolSuite in the dialog box as shown in Figure 1. Click Next.

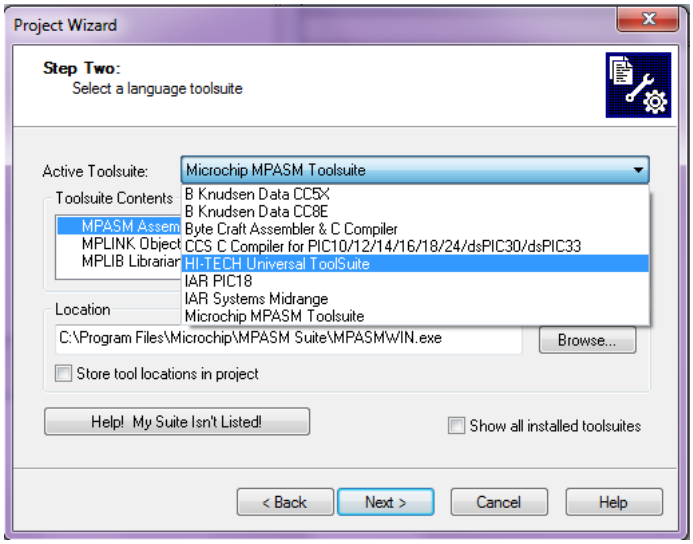


Figure 1: Selecting the HIGH-TECH ToolSuite.

Next, give a name to your project and specify its location in **Step Three** as shown in Figure 2. Click next to move to **Step Four** where you can add the C file to your project as shown in Figure 3. Click next. If you followed the steps correctly, you should see the project window with selected C file under the Source File menu as shown in Figure 4. If you did not save your source file with .C extension, it will appear under the Other Files menu as shown in Figure 5. To correct that, save the file in .C extension and the right click on the Source File menu and select Add Files to select the add the new file.

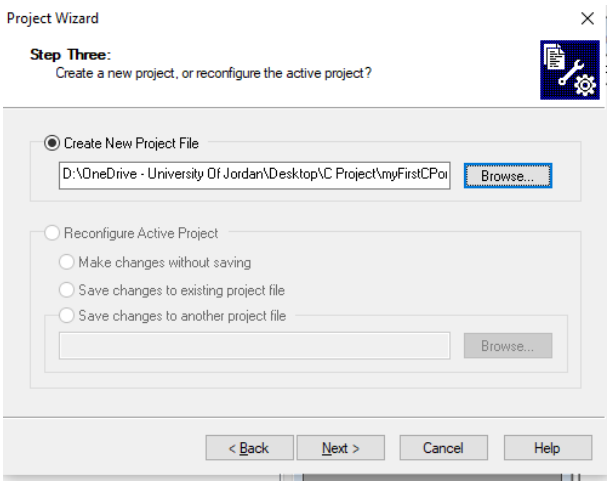


Figure 2: Step Three in creating the project.

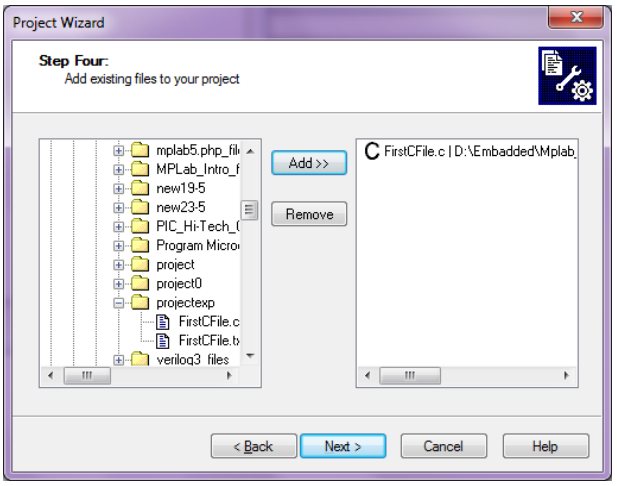


Figure 3: Step Four in creating the project.

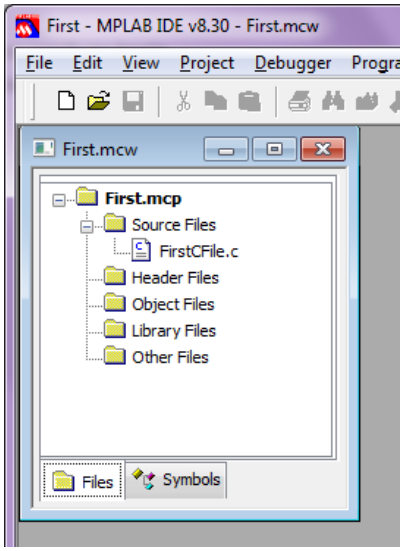


Figure 4: Project window. (CORRECT)

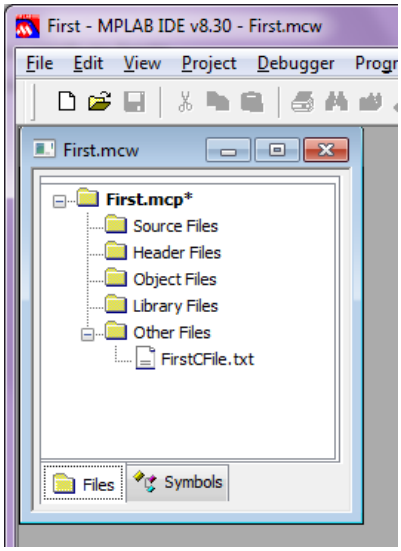


Figure 5: Project window with incorrect file extension. (WRONG)

Building/Compiling C Programs in MPLAB

In this experiment, we will assume that you have prior knowledge in writing general C/C++ programs. In this section, we will show how to write and simulate a general C program in MPLAB IDE.

Now, double-click on the C file that you have added to your project and write the following code.

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

}
```

This is a simple C program that has the main function. As you know, every C program should have the main function. Also, notice how we have included the **htc.h** file, which is essential for the HI-TECH compiler to work.

After writing this simple 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. These shortcuts are circled in Figure 6.

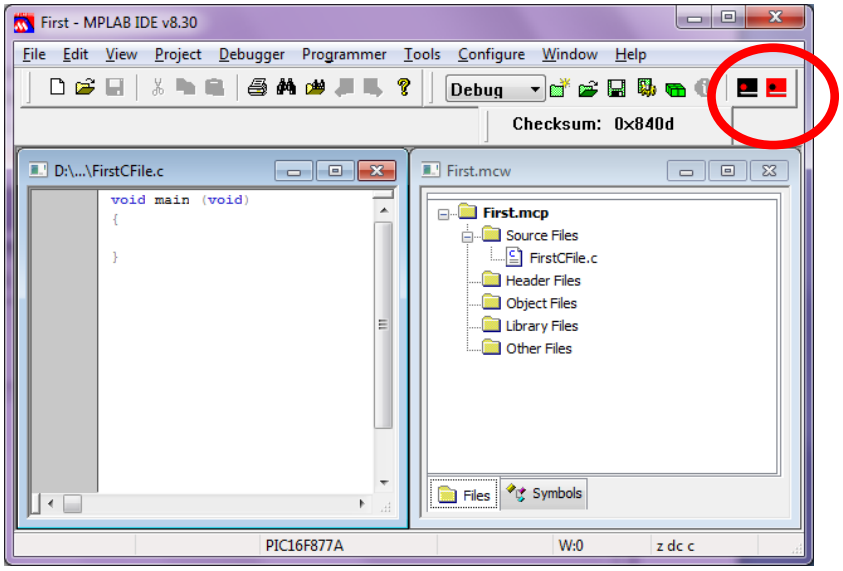


Figure 6: Shortcuts for building C-based project using HI-TECH compiler.

Once you build the project, you should see the output shown in Figure 7. The compiler has produced memory summary and there is no message indicating that the build failed, so we have successfully compiled the project. As we had with ASM files, 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.

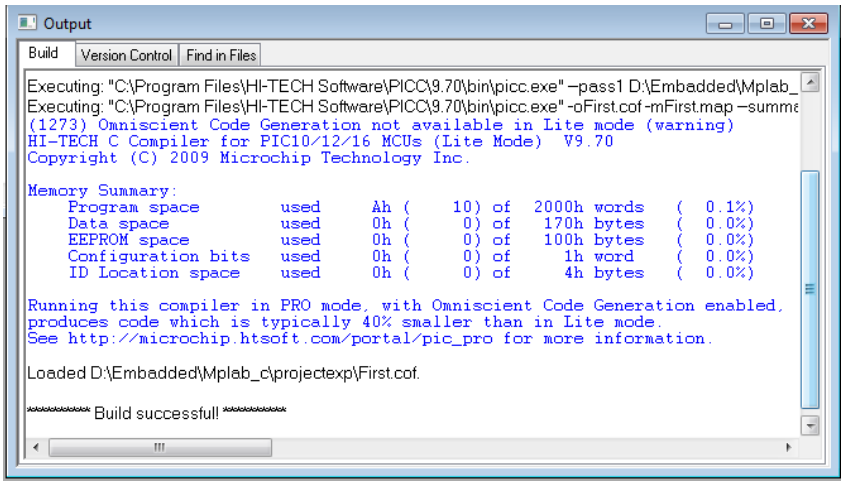


Figure 7: Result of compiling the project.

Remember that 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.

Structure of C Program and Functions

When writing C programs, you need to remember the following **general rules**:

1. **One-line comments start with 2 slashes: //**
   ```
   // This is a one-line comment
   ```
2. **Multi-line comments line start with /* and 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
   }
   ```

In general, the **ordered basic structure of the C program** is as follows:

1. **Libraries**
   Libraries such as htc.h, math.h and stdlib.h, are files that contain functions and constants which you can use. In order to use these functions, you need to include the library that has this function in your program. To do so, you write the statement #include <filename.h>. This statement should be placed at the beginning of your code.

2. **Global Variables**
   Global variables are those that you can use in the main function and other functions in your program. Declaring the global variables should be done at the beginning of the code.

3. **Function Prototypes**
   A C program has a main function and possibly other functions as well which a user may write after the main function. In this case, and to avoid compilation errors, the user has to define these functions before the main function. This is done by simply writing the function prototype before the main program. The function prototype is the header of the function followed by a semicolon. You can avoid using prototypes, although not preferred, by placing the entire function before the main function.

4. **Main Function**
   This is the function that is first called when the program execution is started. ***Every C program must have a main function.***

5. **Functions Definition**
   Functions are the high-level representation of subroutines that you learned in assembly. The body of the function contains a set of statements that can be executed from any place in your code and as many times as needed. To define a function, you use the following 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
}
```

The Type identifier of the function determines the type of the return value. It could be int, long, short, char, void ….. etc. The same thing applies to the input variables (identifiers) that you place between the parenthesis. Remember that a function may return one value, but can take as many inputs as you want. Table 1 shows some examples on defining functions.

To call a function, you simply use its name in your program as shown in Table 1. Notice how tesFunction2 requires using an output variable to store the result and how the parenthesis are still needed when calling testFunction3.

Table 1: Examples on Defining Functions

Function Definition	Comments	How to Call the Function?

void testFunction1(int x, int y)
{
int k;
k = x;
y = 2 + x;
}
	The name of the function is testFunction1. It accepts two integer inputs; x and y, and returns no values. We express this by setting the type of the function as **void**.	testFunction1(4,129);

int testFunction2 (int x)
{
return x*x;
}
	The function name is testFunction2. It has one integer input x and returns an integer value that is x*x.	A = testFunction2(44);

void testFunction3 (void)
{
//some code
}
	The function name is testFunction3. This function accepts no input and returns nothing. Such function might be used to do initialization, print something on the screen, or it may modify global variables.	testFunction3();

Example 1 below is an example on writing a C program and function calc().

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

Variables in C Language

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 included in the program. They are implemented by associating memory locations with variable names. They do not get recreated if the function is recalled. *In Example 1, a, b, c, and temp are **GLOBAL VARIABLES.***
- **Local Variables**
  These variables only exist inside the specific function that creates them. They are not visible or accessible by other functions and to the main program. Local variables don't exist once the function that created them is completed. They are recreated each time a function is executed or called. *In Example 1, the variable p is a **LOCAL VARIABLE.***

In C Language, variables may take different types. Table 2 lists the available data types supported in C. Example 2 below shows an example of defining different types of variables.

Table 2: Data Types in C Language

Type	Size in Bits	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)

```
// 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

C Language supports many types of arithmetic, logic and relational operators. These operations can be applied to variables and constants. Table 3 lists the operators supported in C.

Table 3: C Operators

Type of Operation	Operation	Symbol
Arithmetic	Addition	+
	Subtraction	-
	Multiplication	*
	Division	/
	Modulus (remainder after division)	%
	Increment by 1	x++
	Decrement by 1	x--
Bit	Bitwise NOT	~
	Bitwise AND	&
	Bitwise OR	|
	Bitwise XOR	^
	Shift to left	<<
	Shift to right	>>
Relational	Greater than	>
	Greater than or similar to	>=
	Less than	<
	Less than or similar to	<=
	Equal to	==
	Not equal to	!=

In your programs, you usually write expressions that mix between different types of operators. For example, you may write A = 4*B – 15. In this case, which operation is performed first? Hence, it is necessary to know how to evaluate the expression. This requires defining an order for evaluating operations in the expression. We call this precedence. Table 4 shows the precedence of different operators in the HI-TECH compiler, ordered from highest to lowest. In case two operators have the same precedence, the evaluation is from left to right.

Table 4: Precedence of Operators

Operator	Precedence
Parenthesis ( )	Highest
*** / %**	
+ -	
>> <<	
< > <= >=	
== !=	
&	
^	
|	Lowest

Simulating C Programs in MPLAB

Let's now try to simulate a simple C program in MPLAB. The program is given below and it can be found in ExampleProgram3.asm file. The program calculates the Fibonacci Series by recursively calling Fib() function. The series is obtained by starting with two values; 0 and 1, and then the following values are obtained by adding the previous two values.

```
// 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;
        }
```

To simulate this program in MPLAB:

1. Start a new MPLAB session, create a new project and add the file *ExampleProgram3.c* to your project.
2. Build the project.
3. Select **Debugger** ↳ **Select Tool** ↳**MPLAB SIM.** A set of shortcuts appear on the toolbar as shown below.

4. Go to **View Menu** → **Watch**. Add the variables F1 through F6 to inspect during simulation.
5. Press the "**Step into**" button one at a time and check the Watch window each time an instruction executes.
6. Keep pressing "**Step into**" until you all the six terms of the series are generated.
7. Reset the simulation, do step 5 above but this time use "**Step Over**", note the difference
8. 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.

Note 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, it needs 2 bytes in memory (16 bits as specified in the Table 2), 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 are 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, hence, 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 in C

❖ 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 Statement

```
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 Statement

```
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;
}
```

Writing C Programs for PIC

The preceding discussion introduced the C language in a broad concept. Now, we present an example on writing C programs for the PIC microcontrollers. Actually, it is fairly simple where besides the user-defined variables, the PIC registers are also used in the context of programs.

As you know, the operation of 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 values can be specified in different ways as shown in the following 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. For example:

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

Example 8 below shows a complete C program that continuously flashes a LED connected to RD0 pin.

```
// 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 Example 8, you can either select PORTD from the ADD SFR drop down menu or choose PORTD bits 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 Functions in C

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.

Delay Functions

<table>
<tr><th>_DELAY</th><th>__DELAY_MS and __DELAY_US</th></tr>
<tr><td>

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;
}
```

</td><td>

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;
}
```

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

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 such as *ABS, POW, LOG, LOG10, RAND, MOD, DIV, CEIL, FLOOR, NOP, ROUND* and *SQRT*. 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
Synopsis #include <stdlib.h> int abs (int j) ***Description*** The abs() function returns the absolute value of the passed argument j.	***Synopsis*** #include <math.h> double pow (double f, double p) ***Description*** The pow() function raises its first argument, f, to the power p.
LOG and LOG10	**RAND**
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.	***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
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.	***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.

```
// 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);
        }
        }
```

Note: The define directive

You can use the **#define** directive to give a meaningful name to a constant in your program. The syntax is #define constantName Value.
Example: #define COUNT 1000

Labsheet 5

University of Jordan
School of Engineering
Department of Computer Engineering
Embedded Systems Laboratory 0907334

Using HI-TECH C Compiler in MPLAB

Name:

Student ID:

Section (Day/Time):

COMPUTER NAME:

UNIVERSITY OF JORDAN
SCHOOL OF ENGINEERING
DEPARTMENT OF COMPUTER ENGINEERING
EMBEDDED SYSTEMS LABORATORY CPE0907334

Labsheet5: Using HI-TECH C Compiler in MPLAB

Name: **Student ID:**
Section:

(Pre-lab) Part 1: Code Analysis Skills

1) Create a new project in MPLAB IDE with the following steps:

a. Select the PIC16F877A as the device.
b. Select HI-TECH Universal ToolSuite as the Language Toolsuite.
c. Add the file labsheet5.c to your project
d. Build the project. There should be no errors.

2) Read and simulate the given C code and answer the questions which follow.

a. What is the size of the variables ***a*** and ***i*** in bits?

b. What will happen if you move (cut/paste) the "***char a;*** "and place it in the initial function? Why?

c. What does this operator mean "**<<** "which used in the second for loop "**a=a<<1;** "? What is the instruction in PIC assembly language which performs the exact functionality?

d. What is the task of the **__delay_ms(100)** function ?

e. Rewrite the following C statement "**if (a==00)**"PIC assembly language. Assume a is a GPR.

Part 2: Code Writing Skills (1)

Given the circuit in the Labsheet_5 Proteus Circuit Proteus Part 2 project, it is required to write a program to display the numbers 0 to 8 continuously on the 7-segment display. Your code should have at least two functions: initial and main functions such that:

- The initial function is used to initialize all ports, SFRs and GPR's used in the program and this function is only executed once at the program startup.
- The main function contains all the functions which perform the tasks of the system.
- The nature of the code requires the program to run continuously, i.e. the program code will loop through specific functions which implement the system task.
- Notice that the 7-segment display is common-anode and is connected to PORTD such that segment a is connect to RD0, segment b is connected to RD1 … . The following table lists the 7-segment codes for numbers 0-6. Complete it with the codes of numbers 7 and 8 and use it in your program.

7 Segment Display	Number
0b11000000	0
0b11111001	1
0b10100100	2
0b10110000	3
0b10011001	4
0b10010010	5
0b10000010	6
0b	7
0b	8

Hints

- You need to define a variable to store the current number to be displayed. This variable is incremented by 1 to go to the next number. Once it is 9, it should be cleared.
- The value of the variable is converted to 7-segment code and displayed on PORTD.
- You should put some delay between displaying successive numbers in order to see them.
- There should be an infinite loop in the program to repeat the operation.

Show the simulation in Proteus to the lab engineer and copy your code below.

Part 3: Code Writing Skills (2)

Modify the circuit in Labsheet_5 Proteus Circuit Proteus Part 2 project that you used in Part 2 by adding a switch that is connected to RB0 using a pull-up resistor. Afterwards, modify the code you wrote in Part 2 such that it checks if the switch is closed or opened. **If it is open**, the system counts and displays the value as required; however, when it is closed, the system flashes all the segments continuously.

Show the simulation in Proteus to the lab engineer and copy your code below.

University of Jordan
School of Engineering
Department of Computer Engineering
Embedded Systems Laboratory 0907334

6 Experiment 6: Timers

Objectives

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

- hardware timing modules provided by the PIC 16F877A.
- the concept of 7 segment multiplexing.

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.

Prepared by Dr. Ashraf Suyyagh and Eng. Enas Ja'ra – Revised by Prof. Iyad Jafar

1. Review of Timer0 Operation

Hardware timers are special components that are usually available in most microcontrollers. They can be used for counting and timing purposes, which are very important and frequent operations in embedded systems. As you learned in the course, the PIC16F877A microcontroller has three timers: Timer0, Timer1 and Timer2. In this quick introduction, we will review the operation of Timer0. You are required to read Appendix 1 in this experiment to understand the operation of Timer2.

Timer0 is an 8-bit counter/timer. The block diagram of this timer is shown in Figure 1. As you can see, at the heart of this block is the 8-bit counter TMR0 (address 0x01) which is used to store the count value. The value in this register is incremented by one of two clock sources. The **first source** is the signal that is observed on pin RA4/T0CKI. When this source is selected, the value can be incremented on every rising or falling edge that is received on RA4/T0CKI. In this case, Timer0 is operating in the counter mode and it is basically counting the edges. These edges can be the output of a switch or sensor.

The **second source** that can be used to trigger the increment of the count value in the TMR0 register is internal clock (Fosc/4). When this source is selected, Timer0 is operating in timer mode. In this mode, the value in TMR0 register is incremented every one cycle of Fosc/4. Whenever the value of TMR0 register reaches 255, i.e. the maximum value for an 8-bit register, the register is cleared and the Timer0 Interrupt Flag (T0IF) in the INTCON register is set. This event basically marks an overflow of Timer0. In this experiment, we focus on using Timer0 in the timer mode.

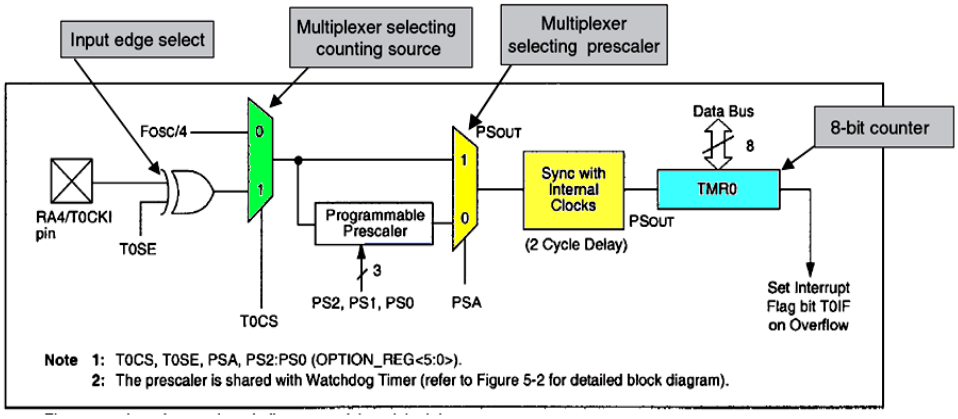


Figure 1. Timer0 block diagram.

So, how can we use Timer0 to measure time? As mentioned earlier, the TMR0 register is incremented on every cycle of Fosc/4. Accordingly, we can calculate the elapsed time using

$$Time = N \times \frac{4}{Fosc} \quad (1)$$

where N represents the number of times Timer0 was incremented. Assuming that Timer0 starts from zero and knowing that TMR0 is 8-bit, then the maximum time that can be measured before the timer overflows is $2^8 \times \frac{4}{Fosc}$ or $\frac{1024}{Fosc}$. What if we want longer time?

One solution is to reduce Fosc; however, this might not be possible. Another solution is to use the Programmable Prescaler block that is shown in Figure 1. This block is basically a frequency divider circuit that can be used to scale down Fosc/4, i.e. scaling up the period 4/Fosc, by a certain prescaler value. In other words, this makes the clock observed by Timer0 slower. Accordingly, the time measured by Timer0 can be calculated using

$$Time = N \times \frac{4}{Fosc} \times prescale\ value \quad (2)$$

In order to use the Timer0 block, we need to configure it using the OPTION_REG which has all the bits needed related to Timer0 such as choosing the source of the clock and specifying whether the prescaler hardware is used with Timer0 or not. Figure 2 shows the OPTION_REG.

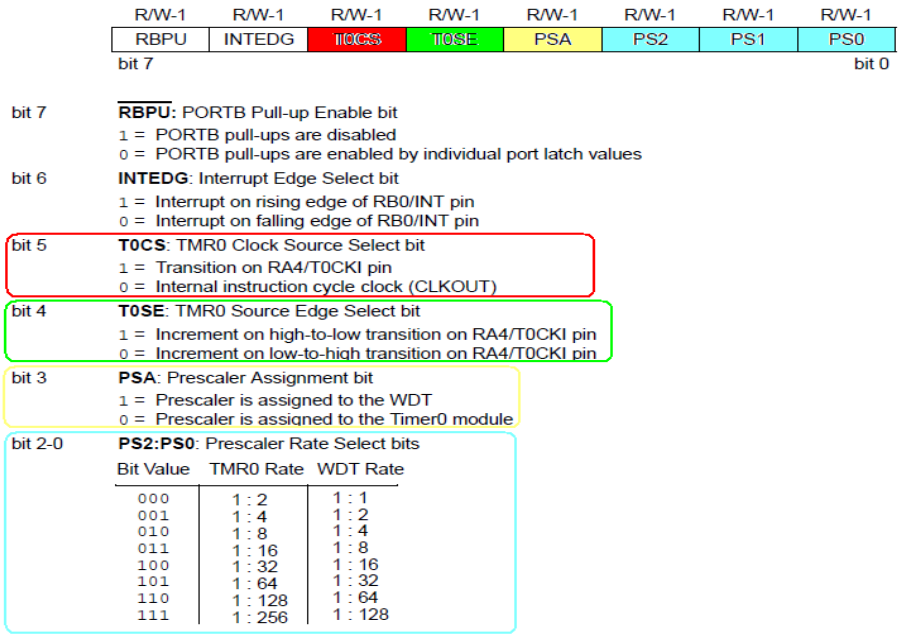


Figure 2: The OPTION_REG register.

Let's see how to configure and use Timer0 to generate a time delay of 0.5 second on PIC16F84A microcontroller with Fosc of 12.8 KHz. The first thing to do is some calculations to figure out the number of increments needed (N) and the preclear value; if it is needed. From Equation (2), we have

$$Time = N \times \frac{4}{Fosc} \times prescale\ value$$

$$0.5 = N \times \frac{4}{12.8 \times 10^3} \times prescale\ value$$

$$1600 = N \times prescale\ value$$

This leaves us with one equation with two unknowns: *N* and the *prescaler value*. Luckily, we can solve this equation by trying different values of the prescaler value that are listed in the table in Figure 2. The possible options that give an 8-bit integer value for N are given in the table below.

Prescaler	N	Note
1	1600	Value can't be used since N is 255 maximum
2	800	
4	400	
8	200	Value can be used
16	100	
32	50	
64	25	
128	12.5	Value has to be truncated since N is integer. Okay to use if the required delay does not need to be accurate
256	6.25	

Let's pick the *prescaler* to be 16. This implies that *N* is 100. Remember that N is the number of increments to be performed in order for Timer0 to overflow, i.e. it reaches 255; thus, the TMR0 register has to be initialized to count from 256-N. Hence, TMR0 in our case should be initialized to 156. To use the prescaler with Timer0, we need to clear the PSA bit to use the Prescaler hardware with Timer0, store $(011)_2$ in the PS2, PS1 and PS0 bits, and clear the T0CS bit in the OPTION_REG to select the internal clock as Timer0 clock.

With these values in hand, we can now write a program to configure and use Timer0 to generate a 0.5s delay as shown below. The program basically flashes an LED that is connected to RB1. The time between flashing is done using the DELAY subroutine that uses Timer0. Every time the subroutine is called, TMR0 is initialized to D'156' and the OPTION_REG is configured as required. Afterwards, the subroutine enters the waiting loop L1 in which it checks T0IF. Try to compile this code and simulate it using the circuit available in **Timer0 Example Proteus Circuit**.

```
        #include p16f84A.inc
        org     0x0000

main    call    initial

repeat
        BANKSEL PORTB
        bsf     PORTB, 1        ; turn on LED
        call    DELAY           ; delay 0.5 sec
        bcf     PORTB, 1        ; turn off LED
        call    DELAY
        goto    repeat

initial
        BANKSEL TRISB
        bcf     TRISB, 1        ; RB1 is output
        return

DELAY
        BANKSEL TMR0
        movlw   D'156' ;        ; preload T0, it overflows after 156 counts
        movwf   TMR0
        BANKSEL OPTION_REG
        movlw   B'00000011'     ;set up T0 for internal clock, prescale by 16
        movwf   OPTION_REG
        BANKSEL TMR0
L1      btfss   INTCON,T0IF     ;test for Timer Overflow flag
        goto    L1              ;loop if not set (no timer overflow)
        bcf     INTCON,T0IF     ;clear Timer Overflow flag
        return
        end
```

This code can be written in C language as shown below. Try to analyze the code to get better understanding on how to use C in writing programs.

```
    #include <htc.h>
    // Function prototypes
    void initial();
    void delay();
    // main
    void main() {
        initial();
        // infinite loop to perform flashing
        while(1){
            PORTB = PORTB | 0B00000010 ; // output 1 on RB1
            delay() ;
            PORTB = PORTB & 0B11111101 ; // output 0 on RB1
            delay() ;
        }
    }
    // initialization
    void initial(){
        TRISB = 0B00000000 ;        // RB1 is output
        return ;
    }
    // Delay
    void delay(){
        TMR0 = 156 ;                // initialize TMR0
        OPTION = 0b00000011 ;       // select Fosc/4, Prescalar = 16
        while(T0IF==0) ;            // loop to wait until T0IF is 1
        T0IF = 0 ;                  // clear T0IF
        return ;
    }
```

2. The Stopwatch Example

In this section, we discuss an example of using Timer0 to design a simple stopwatch system. The system uses a PIC16877A microcontroller running at 4 MHz and is connected through PORTC and PORTD to two common-anode 7-segment displays to display the current count. The 7-segment display that is used to show the most significant digit of the count is connected to PORTC while the other 7-segment is connected to PORTD to show the least significant digit (In order to save the ports, you may have the two displays share the same port and use 7-segment multiplexing technique that you learnt in class. This technique is reviewed in Appendix 3). Additionally, a pushbutton START/STOP is connected to RB0 to control the operation of the stopwatch. Figure 3 shows the schematic diagram of the system. It is available in the **Stopwatch Proteus Circuit** file.

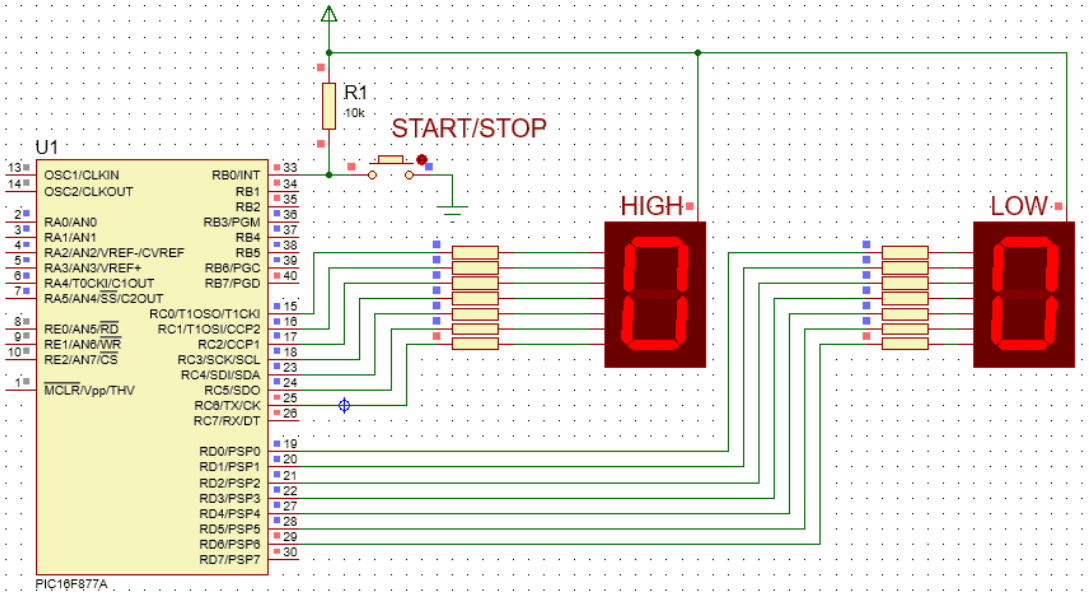


Figure 3: Stopwatch Proteus circuit.

The system operates as follows:

1. When the system starts, it displays 00 on the two displays and waits the user to press the START/STOP button.
2. When the user presses the button, the system starts counting 00, 01, 02, … 59, 00, 01, 02 … such that each increment takes one second. This process is repeated indefinitely until the user presses the button again. Timer0 is to be used for timing.
3. When the user presses the button again, the system pauses counting. However, when he presses it again, counting resumes. In other words, the system toggles between two states; counting and pause, whenever the START/STOP button is pressed.

The first thing to do is to perform some calculations to figure out the configuration of Timer0 to count for 1 second when F_{OSC} is 4 MHz. Using Equation (2), we have

$$1 = N \times \frac{4}{4 \times 10^6} \times prescale\ value$$

If you try different values for the prescaler, you will conclude that we can't find a valid 8-bit value for N. **So, how to solve this without changing F_{OSC}?**

We can use what we call Software Postscaler technique. The idea is simple. Basically, we will allow Timer0 to overflow certain number of times to generate the required one second time. Let's call the

required number of times for Timer0 to overflow the software postscaler P_S. Hence, Equation (2) becomes

$$Time = N \times \frac{4}{Fosc} \times P_H \times P_S \quad (3)$$

where P_H a is the hardware prescaler that we discussed in the previous section. Accordingly, and the equation becomes

$$1 = N \times \frac{4}{4 \times 10^6} \times P_H \times P_S$$

or

$$1000000 = N \times P_H \times P_S$$

Now, we need to find the values for N, P_H and P_S under the constraint that N and P_S are 8-bit integers, and P_H is one of the allowed values for the hardware prescaler as shown in Figure 2. Trying different values, we find that N = 250, P_H = 32 and P_S = 125 gives the required time. In other words, we need to initialize TMR0 with (256-250=6), use a prescaler of 32 (PS2:PS0 = 100_2) and count 125 overflow instances of Timer0. The numbers imply that Timer0 will overflow every $250 \times 4 \times 32/4 \times 10^6$ which is 8 ms. Counting 125 overflow instances gives the required time of 1 second.

Next, we need to design the flow of our program. In our design, we will assume the following:

1. We will use the External Interrupt on RB0 to detect when the user presses the pushbutton. So, we will enable this interrupt source in the INTCON register.
2. We will define the variable START_STOP to store whether the system is counting or stopped. This variable stores 0x00 when the system is stopped and 0xFF when the system is counting. This variable is complemented inside the ISR whenever the START/STOP pushbutton is pressed to change to state of the system.
3. We will use Timer0 Overflow Interrupt to know when it overflows. So, we need to enable this interrupt in the INTCON register.
4. We will define the variable SEC_CALC to count whether Timer0 has overflown 125 times. This variable is incremented by 1 in the interrupt service routine ISR whenever Timer0 overflows until it reaches 125. At this moment it should be cleared.
5. We will use two variables LOW_DIGIT and HIGH_DIGIT to store the two digits to be displayed on the two 7-segment displays. The value of LOW_DIGIT is incremented when SEC_CALC is 125 and is cleared when it reaches 9. The HIGH_DIGIT is incremented when LOW_DIGIT is 9 and it is cleared when its value is 6. This will perform the counting operation from 0 to 59 as required.
6. The LOW_DIGIT and HIGH_DIGIT values are converted to 7-segment codes as we did in Experiment 3.
7. The main program will be responsible for initialization and displaying the count value. Afterwards, it enters an endless loop to update the displays and wait for interrupts. All the operations on the variables and displaying the value on the displays are implemented inside the ISR by calling other subroutines.

The flowchart in Figure 4 shows the operation of the main program and the ISR. Study this flowchart carefully before looking at code that is presented next. The branching to and from the ISR is indicated by the dashed arrows.

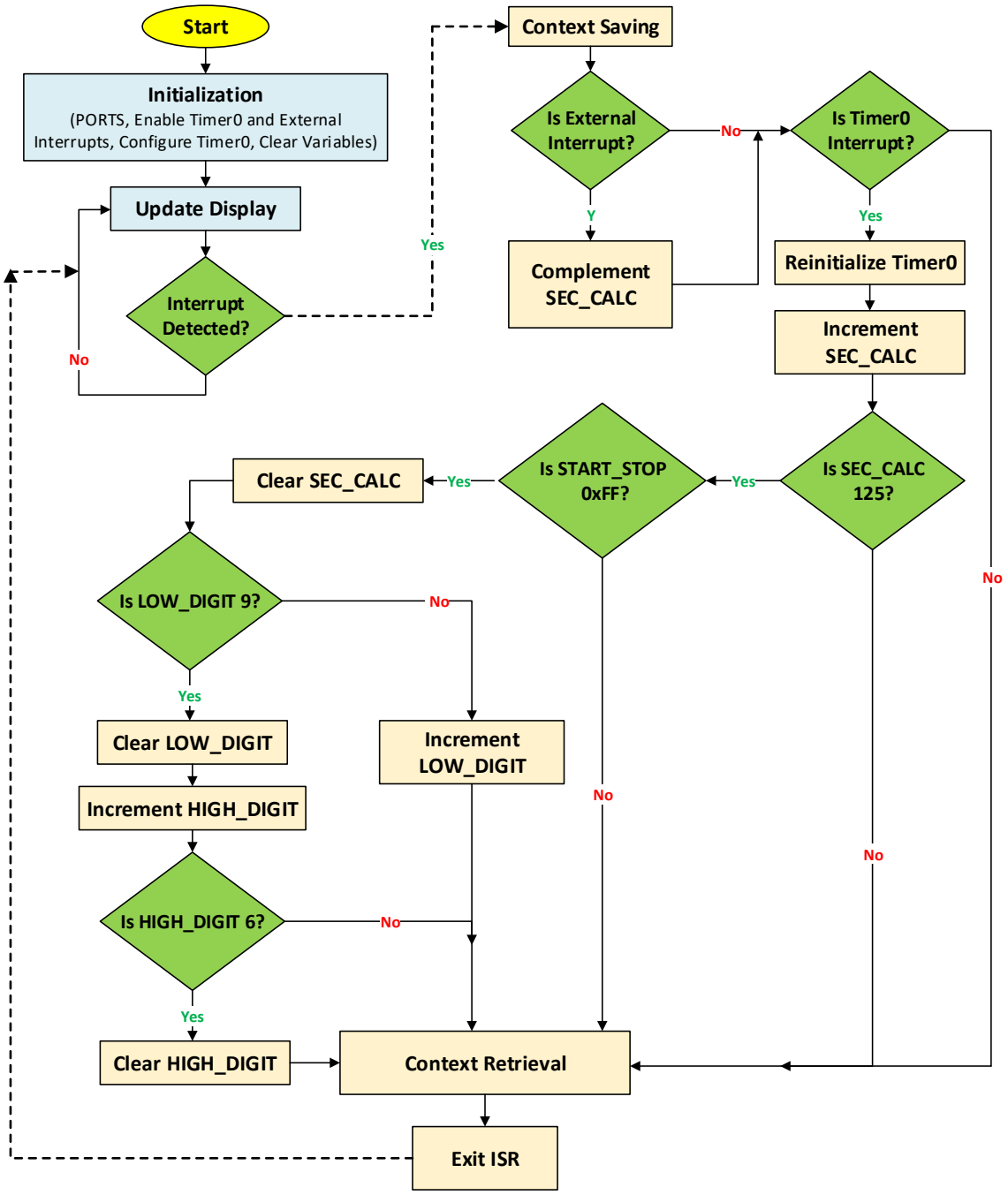


Figure 4: Flowchart of the system.

The assembly code that implements this flowchart is given below. Note the following while studying the code:

1. We check the source of interrupt at the beginning of the subroutine since we are using two interrupt sources; Timer0 and External Interrupt. Based on the source of interrupt we change the flow of execution to the code the service that interrupt source.
2. We pushed the Working register to location tempW at the beginning of the code that services Timer0 to preserve it, since this code modifies the Working register. The value of W is restored at the end this code.

```
;*******************************************************************************************
; Connections:
;       Input:
;           Pushbutton : RB0
;       Output:
;           7-Segment Least Signigicant Digit A-G: PORTD 0-6
;           7-Segment Most Signigicant Digit A-G: PORTC 0-6

;*******************************************************************************************
    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
        tempW               ;location save W during subroutine call (Context saving)
    ENDC
;*******************************************************************************************
    ORG     0X0000
    GOTO    MAIN
    ORG     0X0004
    GOTO    ISR
;*******************************************************************************************
MAIN
    CALL    INITIAL
MAINLOOP
    CALL    DisplayClock
    GOTO    MAINLOOP
;*******************************************************************************************
INITIAL
    BANKSEL TRISA
    CLRF    TRISD               ;PORTD is output
    CLRF    TRISC               ;PORTC is output
    MOVLW   01                  ;RB0 as input (External Interrupt enabled), RB1-RB7 as
    MOVWF   TRISB               ;outputs

    BCF     INTCON, INTF        ;TMR0 and External Interrupts Enabled, their flags
    BCF     INTCON, T0IF        ;cleared
    BSF     INTCON, INTE
    BSF     INTCON, T0IE;>>>>>>>>>
    BSF     INTCON, GIE
    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 ri
                                ;edge

    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

    RETURN
;*******************************************************************************************
ISR
    BTFSC   INTCON, INTF        ;External Interrupt has higher priority
    GOTO    START_STOP_CODE
    GOTO    TMR0_CODE
;*******************************************************************************************
START_STOP_CODE
    BCF     INTCON, INTF        ;clear external interrupt flag
    COMF    START_STOP, F       ;thus halting or starting the stopwatch.
    RETFIE
;*******************************************************************************************
TMR0_CODE
    MOVWF   tempW               ;save W temporarily
    BCF     INTCON, T0IF        ;Clear TMR0 Flag
    BANKSEL TMR0
    MOVLW   0X06                ;Reinitialize TMR0
    MOVWF   TMR0
```

```
80
81          BANKSEL OPTION_REG
82          MOVLW   0XD4
83          MOVWF   OPTION_REG
84
85          BANKSEL TMR0
86          INCF    SEC_CALC, F         ; check if TIMER0 has overflown 125 times
87          MOVLW   .125                ;Assuming a clock of 4MHz, we need
88          SUBWF   SEC_CALC, W         ;250 * 32 * 125 = 1x10^6 µs = 1 Sec
89          BTFSS   STATUS, Z
90          GOTO    ENDTMR0             ;Not 1 Sec yet
91          BTFSC   START_STOP, 0
92          CALL    UPDATE_DIGITS       ;if one second passed, update digits
93      ENDTMR0
94          MOVF    tempW,W             ; restore W
95          RETFIE
96      ;*********************************************************************************
97      UPDATE_DIGITS
98          CLRF    SEC_CALC            ;Cleared so as to count the next 1 sec correctly
99          MOVF    LOW_DIGIT, W        ;If previous low digit is not 9, increment low digit by one
100         SUBLW   0X09                ;else, increment high digit by one and clear low digit
101         BTFSC   STATUS, Z
102         GOTO    UPDATE_HIGH_DIGIT
103         GOTO    UPDATE_LOW_DIGIT
104
105     UPDATE_LOW_DIGIT
106         INCF    LOW_DIGIT, F
107         GOTO    END_UPDATE
108
109     UPDATE_HIGH_DIGIT
110         CLRF    LOW_DIGIT
111         INCF    HIGH_DIGIT, F
112         MOVF    HIGH_DIGIT, W
113         SUBLW   6               ; if high digit reaches 6 (that is number = 60, 1 Minute),
114         BTFSC   STATUS, Z       ;reset
115         CLRF    HIGH_DIGIT
116     END_UPDATE
117         RETURN
118     ;*********************************************************************************
119     DisplayClock
120         BANKSEL PORTC
121         MOVF    LOW_DIGIT, W
122         CALL    Look_TABLE
123         MOVWF   PORTD           ; output LSD
124
125         MOVF    HIGH_DIGIT, W
126         CALL    Look_TABLE
127         MOVWF   PORTC           ; output MSD
128
129         RETURN
130     ;*********************************************************************************
131     Look_TABLE
132         ADDWF   PCL, F
133         RETLW   B'11000000'     ;'0'
134         RETLW   B'11111001'     ;'1'
135         RETLW   B'10100100'     ;'2'
136         RETLW   B'10110000'     ;'3'
137         RETLW   B'10011001'     ;'4'
138         RETLW   B'10010010'     ;'5'
139         RETLW   B'10000010'     ;'6'
140         RETLW   B'11111000'     ;'7'
141         RETLW   B'10000000'     ;'8'
142         RETLW   B'10010000'     ;'9'
143
144         END
```

3. How to Simulate the Stopwatch Example in MPLAB?

You have learnt in Experiment 3 that we can use the Stimulus tool to simulate external inputs to the microcontroller. The Stimulus option is available in the Debugger menu as shown in Figure 5. When you select New Workbook, a new window will appear where you can add the pins to stimulate and specify type action to happen on this pion when stimulated.

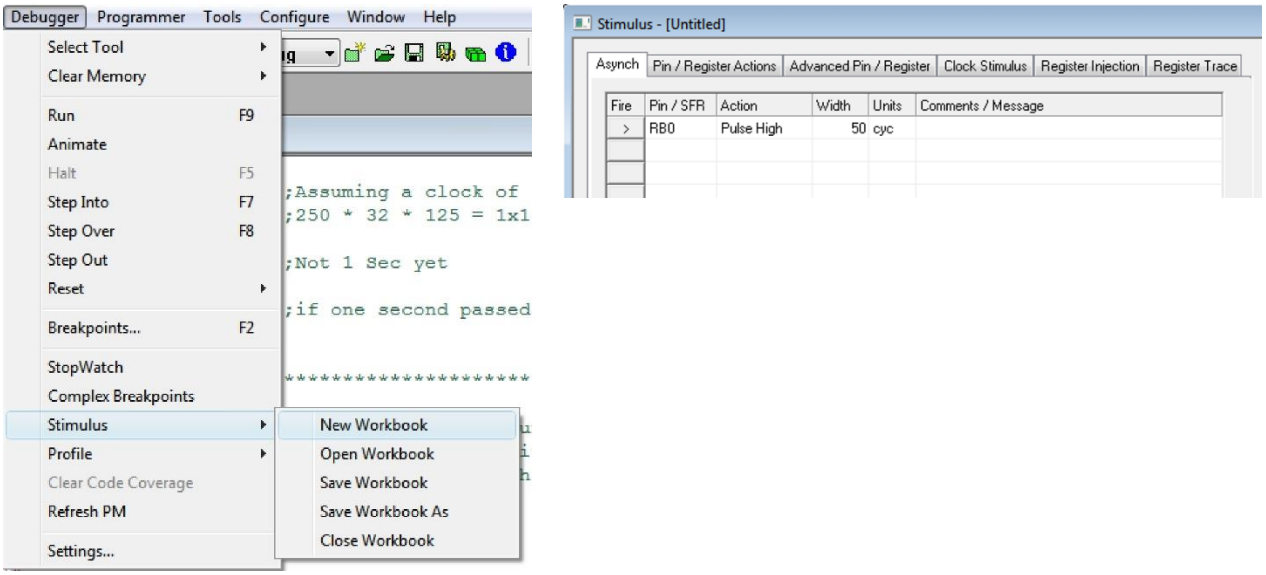


Figure 5. Stimulus window.

In our system, we are observing one input; which is RB0 which is connected to the START/STOP pushbutton. Specifically, we want to generate a rising edge on this pin wherever the pushbutton is pressed. So, in the workbook window, we add RB0 pin and specify the action to be Pulse High.

Now to perform the simulation:

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?

Remember to set the clock to 4 MHz in the Debugger-> Settings menu. Also, make sure that the Watchdog Timer is off. To do so, select Configure-> Configuration Bits and put the Watchdog Timer in the OFF mode. Read more about the Watchdog Timer in Appendix 2.

Appendix 1: Timer2 Module

Timer2 is another timer module that is available in the PIC16F877A microcontroller. The block diagram of this timer is shown in Figure 6. Similar to Timer0, it is an 8-bit timer. However, it can be operated in timer mode only as it can be triggered by the internal clock (Fosc/4). Also, Timer2 has prescaler and postscaler hardware that can be used to extend the time generated by Timer2.

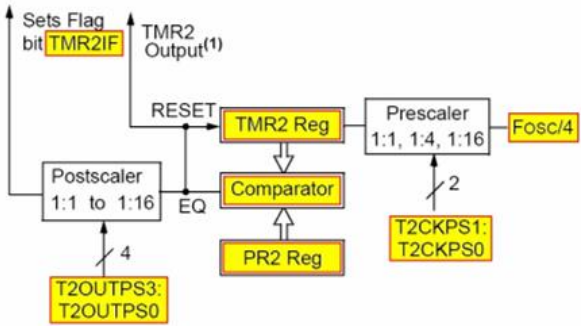


Figure 6: Timer2 block diagram.

Timer2 has two 8-bit data registers: TMR2 and PR2. TMR2 is used to store the initial count value while PR2 is used to store the final count value. Whenever the value in TMR2 register equals that in PR2, TMR2 is cleared and the Timer2 Interrupt Flag (TMR2IF) is set to indicate that. Note that you need to set the T2ON bit in the T2CON in order to force Timer2 to start counting.

The prescaler hardware of Timer2 is similar to that of Timer0. It basically scales-down the clock of Timer2 by some factor. The value of the prescaler can be specified using the T2CKPS1 and T2CKPS2 bits in the T2CON register that is shown in Figure 7.

On the other hand, **the postscaler** in Timer2 delays setting the TMR2IF for specific number of times that equals the postscaler value. The values of the postscaler can be set using the TOUTPS3:TOUTPS0 bits in the T2CON register.

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

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

bit 6:3 **TOUTPS3:TOUTPS0**: Timer2 Output Postscale Select bits

0000 = 1:1 Postscale
0001 = 1:2 Postscale
•
•
•
1111 = 1:16 Postscale

bit 2 **TMR2ON**: Timer2 On bit

1 = Timer2 is on
0 = Timer2 is off

bit 1:0 **T2CKPS1:T2CKPS0**: Timer2 Clock Prescale Select bits

00 = Prescaler is 1
01 = Prescaler is 4
1x = Prescaler is 16

Figure 7: T2CON register.

Accordingly, the time that takes Timer2 to reach the value in PR2 register and set the TMR2IF is given by

$$Time = (PR2 + 1) \times \frac{4}{Fosc} \times prescale\ value \times postscale\ value \quad (4)$$

Appendix 2: Watchdog Timer

The Watchdog Timer (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.

Basically, it is timer similar to other counters in the PIC, but it has its own clock. When this timer is enabled and it overflows, it resets the microcontroller. The WDT can be enabled/disabled setting/clearing the WDTE bit in the Configuration Word. If you decide to enable the WDT, then you need to clear it regularly in your program to avoid resetting the PIC unintentionally when the program is running normally. To do so, you should use the CLRWDT instruction.

So, how long does it take the WDT to overflow?

The PIC data sheet specifies that the WDT has a period from start to finish of 18ms. This is dependent 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. For the sake of simplicity, we will assume that the WDT resets every 18ms.

However, make this longer using the Prescaler hardware that we discussed in Timer0. This prescaler can be assigned to WDT instead of Timer0 to scale down the WDT clock; hence extending the WDT overflow. The assignment can be done using the PSA bit in the OPTION_REG which is given in Figure 2. Notice that different values of the prescaler are used when it is assigned to the WDT. The table below lists the possible time-out periods for the WDT when different values are used.

PS2,PS1,PS0	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	1.1s
1,1,1	1:128	2.3s

Example. Suppose that want the WDT to reset the PIC after about half second. From the table above, the closest value is 0.567s; hence we need to select the prescaler to be 101, i.e. the PS bits are 101. So, we write

```
BANKSEL     OPTION_REG      ; make sure we are in bank 0
CLRWDT                      ; reset the WDT and prescaler
MOVLW       B'00001101'     ;Select the new prescaler value and assign to WDT
MOVWF       OPTION_REG
```

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.

Appendix 3: 7-Segment Multiplexing

The way we designed the Stopwatch system used separate ports to interface the two 7-segment displays. However, this might not be efficient in case we have more devices to interface to the PIC. A solution of this is connect both displays to the same port and display the value on each display for a short period of time repeatedly. This will give the user the illusion that both displays are on. This technique is called **multiplexing**. Figure 8 shows an example of multiplexing two displays.

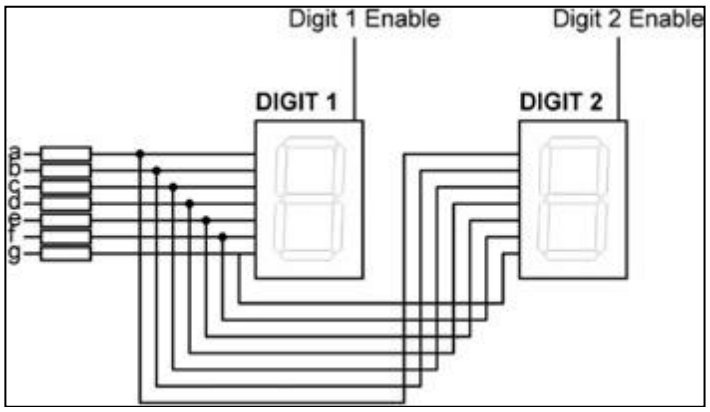


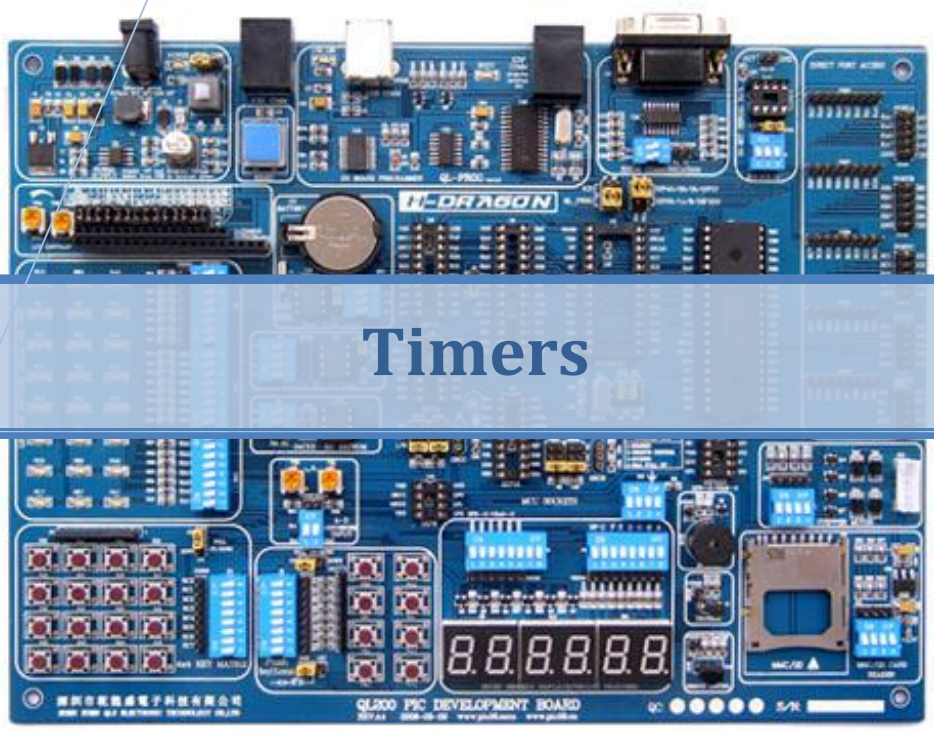
Figure 8: 7-Segment multiplexing.

In this example the LED segments of all the digits are tied together. So, if you send date to any one of the segment, it will be displayed on both segments! To avoid that, the common pins (Enable) 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 file Stopwatch Multiplexing Code.ASM contains the code required to implement the stopwatch operation using 7-segment multiplexing. Try to compile this code and use it in the Stopwatch Multiplexing Proteus Circuit to investigate the operation of multiplexing.

Labsheet 6

University of Jordan
School of Engineering
Department of Computer Engineering
Embedded Systems Laboratory 0907334



Timers

Name:

Student ID:

Section (Day/Time):

COMPUTER NAME:

UNIVERSITY OF JORDAN
SCHOOL OF ENGINEERING
DEPARTMENT OF COMPUTER ENGINEERING
EMBEDDED SYSTEMS LABORATORY CPE0907334

Labsheet6: Timers

Name: **Student ID:**
Section:

(Pre-lab) Part1: The operation of the Timer0 Module and the related OPTION_REG settings

We want to create a delay of 1.6 ms in some program using the Timer0 module in PIC16F877A microcontroller and an oscillator with a value of 8 MHz. Answer the following questions.

Q1) What is the internal frequency?

Q2) Find the instruction cycle time?

Q3) Using equation 3 in the tutorial, find suitable values for N, P_H and P_S to generate this delay.

Q4) What are the values of the following registers: TMR0 and OPTION_Reg in order to generate the time of 1.6 ms?

TMR0 = 0x
OPTION_REG = 0x

Part2: Code Modification

We need to modify the experiment code in the Stopwatch Code.ASM file such that:

1. The system should count from 00 to 19 instead from 00 to 59.
2. The time step between successive values is 0.5s instead of 1s.
3. Oscillator value is still 4MHz.

Answer the following questions.

Q1) What is the internal frequency?

Q2) Find the instruction cycle time?

Q3) Using equation 3 in the tutorial, find suitable values for N, P_H and P_S to generate this delay. Note that Ps is the value of SEC_CALC in the code.

; copy and paste your assembly code here and simulate the program in the Stopwatch Proteus Circuit. Show the simulation to the lab engineer.

Part 3: Using Timer2 in the Stopwatch

We need to modify the system such that instead of using Timer0, we want to use Timer2 for timing the stopwatch. The system should operate as follows:

1. The system should count from 00 to 25 instead from 00 to 59.
2. The time step between successive values is 0.5s instead of 1s.
3. Oscillator value is 4MHz.

Remember that Timer2 has the period register (PR2) and a postscaler hardware in addition to the prescaler. Read Appendix 1 in the tutorial.

What are the values of the following registers in order to generate the time of 0.5 second?

PR2: 0x
SEC_CALC:
Timer2 Prescaler counters: D' '
Timer2 Postscaler counters: D' '
T2CON: B' '

Now, write a C program to implement the stopwatch system using Timer2. You can start with the incomplete code that is available in the Labsheet 6 Part 3.c file.

Notice that we defined the array lookTable that contains the 7-segment codes for number 0 through 9. You can use this table to convert the LOW_DIGIT and HIGH_DIGIT values to their segment codes by writing lookTable[LOW_DIGIT] and lookTable[HIGH_DIGIT].

```
; copy and paste your assembly code here and simulate the program in the Stopwatch Proteus Circuit. Show the simulation to the lab engineer.
```

University of Jordan
School of Engineering
Department of Computer Engineering
Embedded Systems Laboratory 0907334

7

Experiment 7: 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.

Written by Eng. Enas Jaara – Revised by Prof. Iyad Jafar

1. Introduction

In the real world, most of the signals sensed and processed by humans are analog signals. In order to store, process and use these signals in digital systems, these signals have to be converted into digital format. The process that is used for this purpose is called analog-to-digital conversion (ADC) and it uses a special hardware call the analog-to-digital converter.

The ADC process is basically two steps:

1. **Sampling:** in this step, the ADC takes a sample of the input signal. This is done by closing a switch (Sampling Switch) to connect the signal to a capacitor (Hold Capacitor) to store the sample voltage. Since the capacitor is not ideal, we need to wait for the capacitor to charge/discharge before opening the Sampling Switch to disconnect the capacitor. This time is called the acquisition time.
2. **Conversion:** in this step, the voltage on the Hold Capacitor, which represents the sample value, is converted using special hardware into n-bit binary value. The time required to convert the sample is called the conversion time.

Assuming that the sample value is to be represented using n bits, then the ADC basically divides a finite range voltage (Input Range $[V_{ref-},V_{ref+}]$) into 2^n subranges such that each of these subranges has a length of $(V_{ref+}- V_{ref-})/2^n$, which we call Resolution. The binary code of the sample is determined by knowing the subrange that the sample belong to.

For example, consider a 3-bit ADC with $[V_{ref-},V_{ref+}]=[0,4]$. Table 1 shows the subranges and the corresponding binary value to be assigned to any sample in $[V_{ref-},V_{ref+}]$. Any sample outside the range $[V_{ref-},V_{ref+}]$ is clipped. Notice that the binary value is not necessarily the actual voltage of the sample. It basically represents the number of the subrange to which the sample belong. For example, if the binary value is 101, this implies the sample is in [2.5,3.0] volt. Usually, we assume the actual value to be the minimum of the subrange, i.e. 2.5V in this example.

Table 1: Example of 3-bit ADC

Sub-Range	Binary Value
0 ≤ input voltage < 0.5	000
0.5 ≤ input voltage < 1.0	001
1.0 ≤ input voltage < 1.5	010
1.5 ≤ input voltage < 2.0	011
2.0 ≤ input voltage < 2.5	100
2.5 ≤ input voltage < 3.0	101
3.0 ≤ input voltage < 3.5	110
3.5 ≤ input voltage < 4.0	111

Alternatively, it is common to assume that the ADC performs a linear mapping from $[V_{ref-},V_{ref+}]$ to $[0,2^n-1]$. Hence, we can calculate the corresponding voltage of the sample using

$$Sample\ Voltage = \frac{V_{ref+}-V_{ref-}}{2^n-1} \times Binary\ Value + V_{ref-} \quad (1)$$

and we can determine the binary value of the sample using

$$Binary\ Value\ Voltage = \frac{2^n-1}{V_{ref+}-V_{ref-}} \times (Sample\ Voltage - V_{ref-}) \quad (2)$$

2. The PIC16F877A ADC

In general, embedded systems are used to sense different physical analog quantities such as temperature, humidity and light. For this purpose, an ADC can be interfaced to the microcontroller to perform conversion. Since this is very common in embedded systems, many microcontrollers have at least one ADC module integrated within it. For example, the PIC16F877A has an 8-channel 10-bit ADC module. This implies that this ADC can be connected to eight different signals and each sample is represented using 10 bits.

Using the ADC module in PIC16F877A microcontroller is similar to using other modules inside the microcontroller, i.e. using special function registers. Specifically, the ADC in PIC16F877A has two control registers; ADCON0 and ADCON1, and two result data registers; ADRESH and ADRESL. The control registers are used to configure the A/D while the result registers are used to store the 10-bit binary value that comes out of the ADC. Table 2 shows these registers.

The ADCON0 register contains a set of bits:

1. **ADON →** Turn on the ADC. By default, it is turned off on power-up to save power.
2. **CHS2:CHS0 →** Select the channel to read the sample from.
3. **GO/$\overline{\text{DONE}}$ →** Start the conversion process once the sample is acquired. This bit is cleared by the ADC once the conversion is complete.
4. **ADCS2:ADCS0 →** Specifying the clock rate of the ADC. Note that ADC2 is in ADCON1 register.

On the other hand, the ADCON1 register has the following bits:

1. **PCFG3:PCFG0 →** Specify whether PORTA and PORTE pins are analog or digital pins.
2. **ADFM→** Format the 10-bit result in the result registers.

Table 2: Control and Data Register of the ADC

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

In order to use the ADC, we need to write instructions to configure its features such the clock rate, the channel being used and the formatting of the result. Please refer to Appendix 1 to learn the details on how to specify the values of different bits in these two registers.

In general, using the ADC follows the steps that are shown in Figure 1. For the ADC in PIC16F877A, note that:

- We need to wait some time before we start a conversion process after turning on the ADC on.
- Closing the sampling switch to start acquiring the sample is done by selecting the input channel.
- The end of conversion is known by checking the GO/DONE bit or the ADC interrupt flag (ADIF).

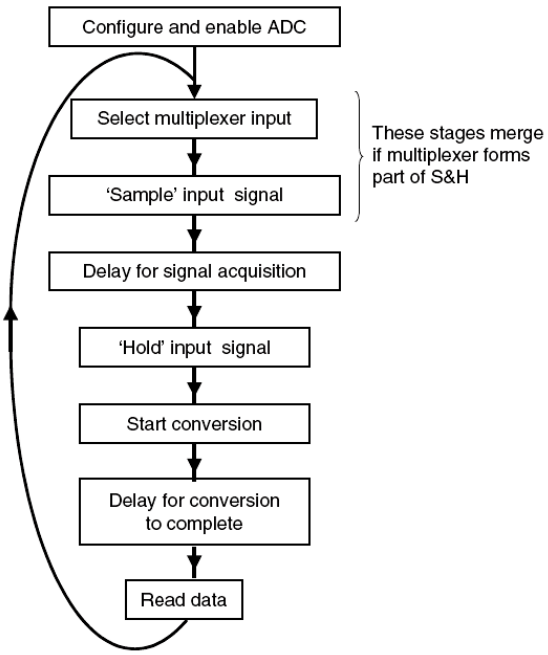


Figure 1: Steps for using ADC.

3. ADC Example

Let's demonstrate the use of the PIC16F877A ADC with Fosc being 2 MHz in an example in which we want to convert the voltage that is coming out of a potentiometer continuously to 10-bit digital value and display the upper 8-bit of the result on three 7-segment display as a BCD value.

The potentiometer is connected to RA0, i.e. we will use channel0 for ADC. The three segment displays are common-anode and are connected to PORTD. We will use 7-segment multiplexing to display the value on the displays (Read Appendix 3 in Experiment 6). To control which display is enabled, we connect the common input of the hundreds, tens and units displays to RB1, RB2 and RB3 pins, respectively.

We will configure the ADC features as follows:

- Turn on the ADC by setting the set ADON bit.
- Choose the analogue channel 0 "AN0" as the analogue input of the AD module by setting CHS2:CHS0 to 000.
- RA0 should be configured as analog input. Set the voltage references V_{ref-} and V_{ref+} to be internal. This can be done by setting PCFG3:PCFG0 bits to 1110 (other options are possible, check Appendix 1).
- The result is to be left justified such that the upper 8-bits will reside in ADRESH and the lower 2 bits will reside in ADRESL. In this program, we will choose to ignore ADRESL and only deal with the upper 8 bits of digitized value to simplify program development. To do so, clear the ADFM bit.
- The ADC clock is set to Fosc/8 by setting ADCS2:ADCS0 bits to 001.
- Hence, ADCON1 should be 0x0E and ADCON0 should be 0x41.

In general, the program will operate such that it continuously uses the ADC to convert the value on RA0 to digital, convert the value in ADRESH from binary to three BCD digits that represent the hundreds, tens and units digits of the corresponding digital value, and then display them on the 7-

segment displays. The flowchart of the system is given in Figure 2. Study the flowchart along with the following code to understand the operation of the system. Try to simulate the program using the ADC Example Proteus Circuit.

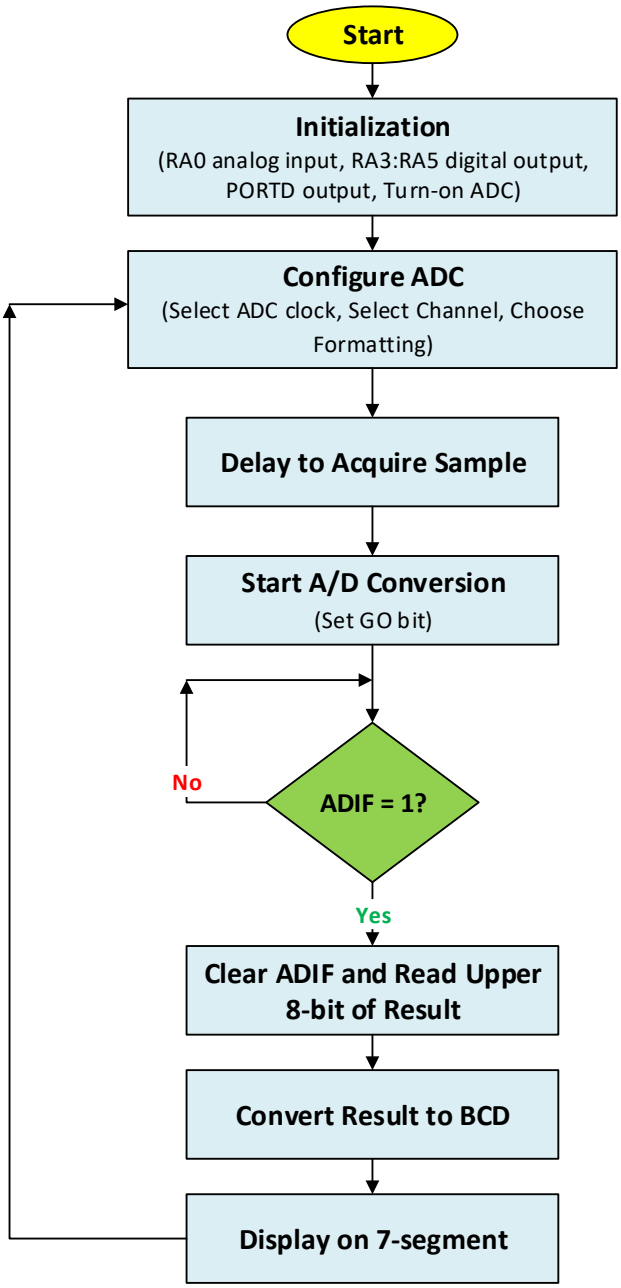


Figure 2: Flowchart of ADC Example.

```
; ********************************************************************************
; ADC Example
; This code reads the voltage from a potentiometer connected to analog
; channel 0 (RA0), converts the lower 8 bits of the result (ADREL) to
; 3-digit BCD code, and displays them on three 7-segment displays using
; 7-segment multiplexing

; Inputs:
; RA0 - analog
; Outputs:
; RB1, RB2, RB3 - digital outputs connectd to the commons of 7-seg displays
; PROTD - connected to segments a-g of the displays
; ********************************************************************************

__CONFIG _DEBUG_OFF&_CP_OFF&_WRT_HALF&_CPD_OFF&_LVP_OFF&_BODEN_OFF&_PWRTE_OFF&_WDT_OFF&_XT_OSC

                #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

;**********************MAIN program**********************
        ORG       00H
MAIN
        BANKSEL     TRISA           ;select bank 1
        MOVLW       01H             ;PORTA bit Number0 is INPUT
        CLRF        TRISD           ;All of the PORTD bits are outputs
        CLRF        TRISB
        BANKSEL     ADCON1
        MOVLW       0x0E
        MOVWF       ADCON1
        BANKSEL     ADCON0
        BSF         ADCON0, 6       ;Set ADCS0
        BSF         ADCON0, 0       ;Set ADON to start ADC
        CALL        DELAY           ;Wait for ADC to start

LOOP                                ;Code to initilize ADC
        CLRF        tens
        CLRF        hundreds
        CLRF        units

        MOVLW       0x41
        MOVWF       ADCON0          ;Select channel 0
        CALL        DELAY           ;Wait to acquire sample

        BSF         ADCON0, GO      ;Start conversion
        BANKSEL     PIR1            ;Poll ADIF flag to wait for conversion
WAIT    BTFSS       PIR1, ADIF
        GOTO        WAIT
        BCF         PIR1, ADIF      ;Clear flag to prepare for next conversion
        BANKSEL     ADRESH
        MOVF        ADRESH, W
        MOVWF       TEMP
        CALL        CHANGE_To_BCD   ;call result convert subroutine
        CALL        DELAY
        CALL        DISPLAY         ;call display subroutine
        CALL        DELAY
        GOTO        LOOP            ;Do it again

;**********************Convert subroutine**************************
; Subroutine to convert the 8-value in Temo to 3 BCD digits
;******************************************************************
```

```
CHANGE_To_BCD
        BANKSEL     PORTA
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*************************
; Subroutine to show the three values on the 7-seg displays
;*********************************************************************
DISPLAY
        MOVF        hundreds,W              ;display Hundreds bit
        CALL        TABLE
        MOVWF       PORTD
        BSF         PORTB,1
        CALL        DELAY
        BCF         PORTB,1

        MOVF        tens,W                  ;display Tens bit
        CALL        TABLE
        MOVWF       PORTD
        BSF         PORTB,2
        CALL        DELAY
        BCF         PORTB,2

        MOVF        units,W                 ;display Units bit
        CALL        TABLE
        MOVWF       PORTD
        BSF         PORTB,3
        CALL        DELAY
        BCF         PORTB,3
        RETURN

;************************** Lookup Table ***********************
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'

```

```
128     ;***************************Delay subroutine***********************
129     DELAY
130             MOVLW       0x0F
131             MOVWF       TEMP
132     L1      DECFSZ      TEMP,1
133             GOTO        L1
134             RETURN
135
136     ;*************************************************************
137             END                             ;program end
138
```

Appendix 1: PIC16F877A ADC Registers

ADCON0 Register 0x1F

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0
ADCS1	ADCS0	CHS2	CHS1	CHS0	GO/$\overline{\text{DONE}}$	—	ADON
bit 7							bit 0

bit 7-6 **ADCS1:ADCS0:** A/D Conversion Clock Select bits (ADCON0 bits in **bold**)

ADCON1 <ADCS2>	ADCON0 <ADCS1:ADCS0>	Clock Conversion
0	00	Fosc/2
0	01	Fosc/8
0	10	Fosc/32
0	11	FRC (clock derived from the internal A/D RC oscillator)
1	00	Fosc/4
1	01	Fosc/16
1	10	Fosc/64
1	11	FRC (clock derived from the internal A/D RC oscillator)

bit 5-3 **CHS2:CHS0:** Analog Channel Select bits
- 000 = Channel 0 (AN0)
- 001 = Channel 1 (AN1)
- 010 = Channel 2 (AN2)
- 011 = Channel 3 (AN3)
- 100 = Channel 4 (AN4)
- 101 = Channel 5 (AN5)
- 110 = Channel 6 (AN6)
- 111 = Channel 7 (AN7)

bit 2 **GO/$\overline{\text{DONE}}$:** A/D Conversion Status bit
When ADON = 1:
1 = A/D conversion in progress (setting this bit starts the A/D conversion which is automatically cleared by hardware when the A/D conversion is complete)
0 = A/D conversion not in progress

bit 1 **Unimplemented:** Read as '0'

bit 0 **ADON:** A/D On bit
1 = A/D converter module is powered up
0 = A/D converter module is shut-off and consumes no operating current

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

ADCON1 Register 0x9F

R/W-0	R/W-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0
ADFM	ADCS2	—	—	PCFG3	PCFG2	PCFG1	PCFG0
bit 7							bit 0

bit 7 **ADFM:** A/D Result Format Select bit
1 = Right justified. Six (6) Most Significant bits of ADRESH are read as '0'.
0 = Left justified. Six (6) Least Significant bits of ADRESL are read as '0'.

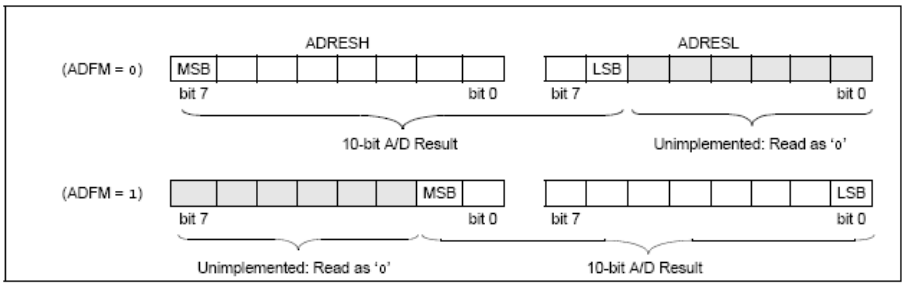
bit 6 **ADCS2:** A/D Conversion Clock Select bit (ADCON1 bits in shaded area and in **bold**)

bit 5-4 **Unimplemented:** Read as '0'

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

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

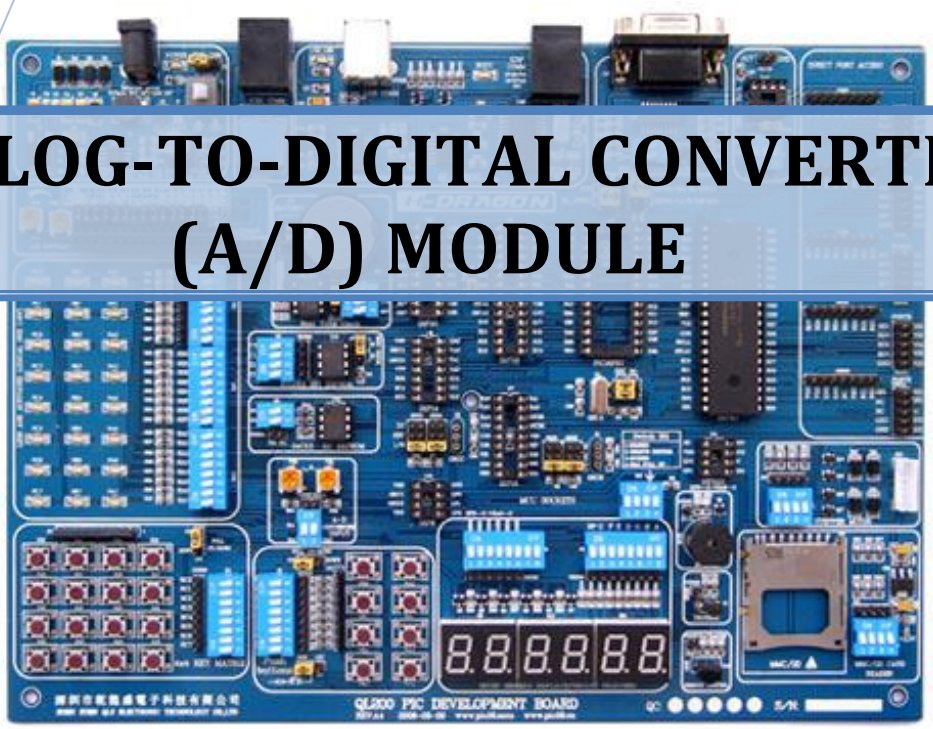
A = Analog input D = Digital I/O
C/R = # of analog input channels/# of A/D voltage references



Labsheet 7

University of Jordan
School of Engineering
Department of Computer Engineering
Embedded Systems Laboratory 0907334

ANALOG-TO-DIGITAL CONVERTER (A/D) MODULE



Name:

Student ID:

Section (Day/Time):

COMPUTER NAME:

UNIVERSITY OF JORDAN
SCHOOL OF ENGINEERING
DEPARTMENT OF COMPUTER ENGINEERING
EMBEDDED SYSTEMS LABORATORY CPE0907334

Labsheet 7: Analog-to-Digital Converter (A/D) Module

Name: **Student ID:**
Section:

Overview

In this experiment, we will use the ADC in PIC16F877A to read the temperature and display it on two 7-segment displays without using 7-segment multiplexing. The sensor that is used to acquire the temperature is the MCP9700. The Proteus circuit is available in the Labsheet 7 Proteus Circuit and is shown in Figure 1.

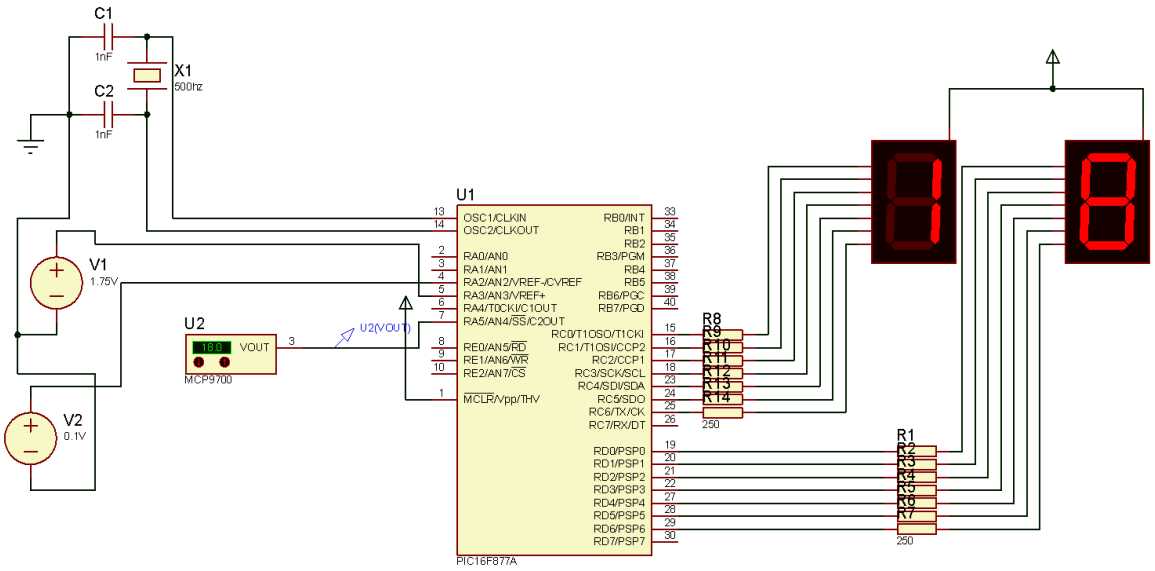


Figure 1: Proteus Circuit.

According to the datasheet, the MCP9700 sensor can measure the temperature in the range of [-40,+125]°C and has a positive temperature coefficient of 10mV/°C, i.e. its output increases/decreases by 10mV when the temperature increases/decreases by 1°C. For example, when the temperature is -40°C, the output voltage is 0.1V and when the temperature increases to-39°C the output voltage becomes 0.11V.

The chart in Figure 2 shows the relation between the sensor output voltage and the measured temperature. From this chart, you can notice that the minimum output voltage is 0.1V which represents the minimum temperature of -40°C and the maximum output voltage is 1.75V which represents the maximum temperature of +125°C.

Hence, and in order to get the best performance of the ADC, we connect AN3 (V_{ref+}) to 1.75V voltage source and we connect AN2 (V_{ref-}) to 0.1V voltage source. Accordingly, the resolution of the ADC is (1.75-0.1)/1024 or 1.6mV/step. In other words, when the input voltage increases/decreases by 1.6mV, the ADC digital output increases/decreases by 1. Table 1 shows the input value range and the corresponding digital output.

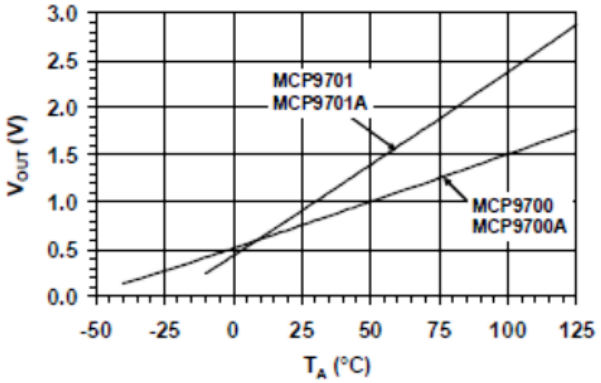


Figure 2: Voltage-temperature relationship for MCP9700.

Table 1: Input Range and Corresponding Digital Output

Input Range	Digital Output
0.1V ≤ input voltage < 0.1016V	0
0.1016V ≤ input voltage < 0.1032V	1
0. 1032V ≤ input voltage < 0.1048V	2
0.1048V ≤ input voltage < 0.1064V	3
0. 1064V ≤ input voltage < 0.1080V	4
0.1080V ≤ input voltage < 0.1096V	5
0.1096V ≤ input voltage < 0.1112V	6
. . . . .	. . . . .
1.7468V ≤ input voltage < 1. 7484V	1022
1.7484V ≤ input voltage < 1.75V	1023

Based on this, we can conclude that when the temperature changes by 1°C the output voltage of the sensor changes by 10mV which causes the ADC digital output to change by

$$10\ \frac{mV}{^oC} \div 1.6\ \frac{mV}{Step} \cong 6\ \frac{Step}{^oC} \qquad (1)$$

In other words, the ADC output increases/decreases by 6 approximately when the temperature increases/decreases by 1°C. We can use this conclusion to simplify reading the temperature when we write our program instead of performing calculations using the equations we have in the tutorial.

Prelab

You are required to modify the program in the ADC Example Code.ASM to read the temperature, convert it to two BCD digits, and display it on the 7-segment displays. The ADC is configured as follows:

- V_{ref+} = AN3, V_{ref-} = AN2
- RA5 or channel4 is the ADC input
- ADC output is **right-justified**
- ADC clock is $F_{osc}/8$

What are the values to be stored in ADCON0 and ADCON1 register?

ADCON0 =	( )$_2$
ADCON1 =	( )$_2$

The system should show only the temperature when it is in [5,35]˚C range, otherwise, the system shows 0xEE on the displays. Using equation (1), **what are the <u>binary and decimal values</u> in ADRESH and ADRESL registers that correspond to the required range? Notice that ADRESH will hold the most significant bits of the conversion result since we chose the result to be right-justified. Remember that a temperature of 5˚C implies 45˚C increments from -40˚C.**

	ADC Result															
Temp.	**ADRESH**								**ADRESL**							
5˚C																
35˚C																

ADRESL =	( )$_{10}$
ADRESH =	( )$_{10}$

In Lab

Modify the code in order to read the digital output of the ADC and then convert to a 2-digit temperature in ˚C displayed on a 2-digit 7-segment display only when the temperature is in the range of 5˚C to +35˚C. In order to do that, you need the following:

- **Reading ADC Result:** You need to read the ADC result from the ADRESL and ADRESH registers and store them in RL and RH locations, respectively.
- **Converting the ADC Result to Temperature:** to convert the ADC result to actual temperature, we need to <u>divide it by 6 then subtract 40</u> (WHY?). However, the ADC result is 10 bits (the most significant two bits in RH and the least significant 8 bits in RL) and our microcontroller is 8-bit. So, we can't do the calculations on the ADC result directly. Alternatively, we express the ADC result in RH::RL by

$$ADC\ Value = RH_1 \times 2^9 + RH_0 \times 2^8 + RL \quad (2)$$

Where RH_1 and RH_0 are bits 1 and 0 in RH, respectively. Mathematically, we can break down the calculation of the temperature to

$$Temperature = \frac{RH_1 \times 2^9 + RH_0 \times 2^8 + RL}{6} - 40 \quad (3)$$

$$Temperature = RH_1 \times 85 + RH_0 \times 44 + \frac{RL}{6} - 40 \quad (4)$$

Accordingly, write the subroutine CONVERT_TEMP to convert the values in RH and RL to actual temperature using equation (4) and store it in location RESULT.

- **Check Temperature:** write a subroutine CHECK_RANGE to check whether the value in RESULT is in [5,35]. If the temperature is in range, store 1 in location VALID. Otherwise, store 0.
- **Convert to BCD:** Modify the CHANGE_to_BCD subroutine to convert the temperature value in RESULT into two separate BCD digits instead of three.
- **Display:** Modify the DISPLAY subroutine to display two digits instead of three without using 7-segment multiplexing. The units digit is shown on PORTD while the tens digit is shown on PORTC. The subroutine should check if VALID is 0 or not in order to show the temperature if it is in range or to show 0xEE otherwise.

```
; Copy and paste your code here
```

Ask your engineer to check the simulation

University of Jordan
School of Engineering
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.

Written by Eng. Enas Jaara – Revised by Prof. Iyad Jafar

1. Introduction

The universal synchronous asynchronous receiver transmitter (USART) is one of serial communication modules available in PIC16F87x microcontroller. When operated in the asynchronous mode, it can be used to send and receive data simultaneously, i.e. full-duplex mode. In this mode, 8-bit data is sent/received along with a START and STOP bits as frame. Optionally, a parity bit can be added to the frame to aid detection of odd errors. Figure 1 shows one frame.

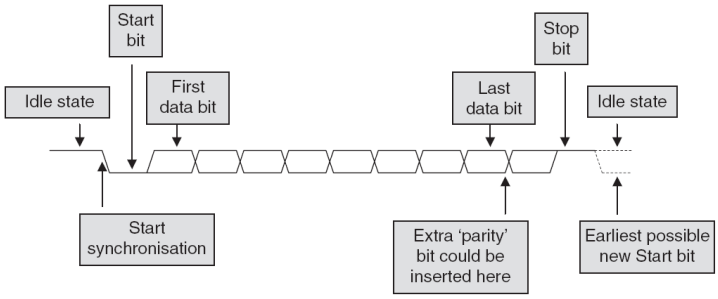


Figure 1: Contents of one frame.

The block diagram of the asynchronous transmitter is shown in Figure 2. There are several registers that are associated with using the asynchronous transmitter. These are listed in Figure 3. In order to use this transmitter, you need to consider the following steps:

1. The output of the transmitter appears on pin RC6; hence we need to clear TRISC<6> bit to configure RC6 as output.
2. Specify the transmission rate by specifying the values of the SPBRG register and BRGH bit in the TXSTA register (more on this later).
3. Enable the USART module by setting the SPEN bit in RCSTA register and configure it in asynchronous mode by clearing the SYNC bit in the TXSTA register.
4. If interrupts are desired, set the TXIE in the PIE1 register, GIE and PEIE bits in the INTCON register.
5. Enable transmission by setting the TXEN in the TXSTA register. This will set the TXIF flag bit to indicate the TXREG is empty.
6. If 9-bit transmission is desired:
    a) Set the TX9 bit in the TXSTA register.
    b) Store the ninth bit in TX9D in the TXSTA register.
7. Store the data to be transmitted to TXREG register to start the transmission.

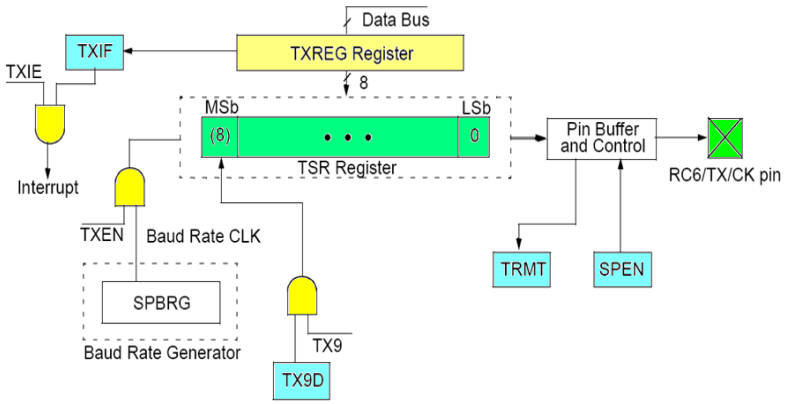


Figure 2: The block diagram of asynchronous transmitter.

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on: POR, BOR	Value on all other Resets
0Bh, 8Bh, 10Bh, 18Bh	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	R0IF	0000 000x	0000 000u
0Ch	PIR1	PSPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000	0000 0000
18h	RCSTA	SPEN	RX9	SREN	CREN	—	FERR	OERR	RX9D	0000 -00x	0000 -00x
19h	TXREG	USART Transmit Register								0000 0000	0000 0000
8Ch	PIE1	PSPIE(1)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000	0000 0000
98h	TXSTA	CSRC	TX9	TXEN	SYNC	—	BRGH	TRMT	TX9D	0000 -010	0000 -010
99h	SPBRG	Baud Rate Generator Register								0000 0000	0000 0000

Figure 3: Registers related to the asynchronous transmitter.

Similarly, the block diagram of the asynchronous receiver is shown in Figure 4 while the related registers are shown in Figure 5. The steps for using the asynchronous receiver are as follows:

1. The input of the receiver is from pin RC7; hence we need to set TRISC<7> bit to configure RC7 as input.
2. Specify the transmission rate by specifying the values of the SPBRG register and BRGH bit in the TXSTA register (more on this later).
3. Enable the USART module by setting the SPEN bit in RCSTA register and configure it in asynchronous mode by clearing the SYNC bit in the TXSTA register.
4. If interrupts are desired, set the RCIE in the PIE1 register, GIE and PEIE bits in the INTCON register.
5. If 9-bit reception is desired, set the RX9 bit in the RCSTA register.
6. Enable the reception by setting bit CREN in RCSTA register.
7. The RCIF flag in PIR1 will be set when reception of one word is complete and an interrupt will be generated if RCIE is set.
8. Read the RCSTA to get the 9th parity bit and determine if any error occurred (OERR, FERR bits).
9. Read the 8-bit received data by reading RCREG.
10. If any error occurred, clear the error by clearing the CREN.

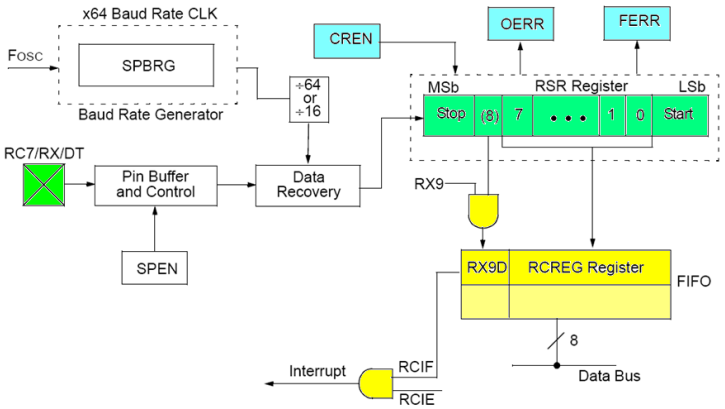


Figure 4: The block diagram of the asynchronous receiver.

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on: POR, BOR	Value on all other Resets
0Bh, 8Bh, 10Bh, 18Bh	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	R0IF	0000 000x	0000 000u
0Ch	PIR1	PSPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000	0000 0000
18h	RCSTA	SPEN	RX9	SREN	CREN	—	FERR	OERR	RX9D	0000 -00x	0000 -00x
1Ah	RCREG	USART Receive Register								0000 0000	0000 0000
8Ch	PIE1	PSPIE(1)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000	0000 0000
98h	TXSTA	CSRC	TX9	TXEN	SYNC	—	BRGH	TRMT	TX9D	0000 -010	0000 -010
99h	SPBRG	Baud Rate Generator Register								0000 0000	0000 0000

Figure 5: Registers related to the asynchronous receiver.

To specify the transmission/reception baud rate, you need to specify the values of the SPBRG register and the BRGH bit in the TXSTA register. These values along with the frequency of the PIC clock are used to determine the baud rate using one of the following formula

$$Baud\ Rate = \begin{cases} \frac{Fosc}{64(SPBRG+1)}, & BRGH = 0 \\ \frac{Fosc}{16(SPBRG+1)}, & BRHG = 1 \end{cases} \quad (1)$$

Alternatively, you can use the tables in the datasheet to find the proper values for SPBRG and BRGH in order to have specific baud rate at certain Fosc.

The information presented previously serves as a quick overview of the USART. You are strongly recommended to review the topic from the textbook or from the datasheet in case you feel that you are missing some details.

2. USART Example

In a certain factory, a modern computerized machine is serially connected to a control computer. The machine has a PIC16F877A microcontroller and uses its Universal Synchronous Asynchronous Receiver Transmitter (USART) module to communicate with the computer. When the machine is powered on, it sends the message "Machine ready to receive commands" to the control room indicating that it is ready to receive commands. After receiving the message by the computer, 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-segment display.

It is required to write a program for the PIC to perform the required operation. The general flow of the program is as follows:

- Initialize I/O, enable interrupts, configure USART settings (baud rate, transmitter and receiver settings).
- Send message to control computer.
- Wait until command is received from control computer. When received, show it on the 7-srgment display.

The following steps details the operation of the program.

Step 1: Initialization

- PORTD will be connected to the 7-segment display to show the received commands. It is configured as output.
- **USART pins**
  - RC6 is used by the USART transmitter. So, it has to be configured as output.
  - RC7 is used by the USART receiver. So, it has to be configured as input.
- **USART Configuration:**
  - We will use the USART in asynchronous mode, so (SYNC = 0).
  - Enable serial port (SPEN = 1), enable receiver (CREN = 1), enable transmitter (TXEN = 1)
  - The Baud rate to be used is 9600 bps. Assuming the PIC is running at 4MHz, 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 communicate at this rate.
- **Interrupts:** the PIC will use the receiver interrupt to know when a command is received. So, we need to set (GIE = 1), (PEIE = 1) and (RCIE = 1).

Step 2: Sending Message to Control Computer

When the machine starts, it should send the message "Machine ready to receive commands" to the host computer. The message has 33 characters and will be stored in a lookup table called Message with the first entry being letter "M" and last entry being letter "s". The table will be accessed 33 times in a loop to read all the letters. The loop variable INDEX is initialized to 0 and is incremented every time a letter is sent. When it reaches 33, this implies that whole message is sent.

In order to send each character serially, it has to be stored to the transmitter register in the USART (TXREG). The character is sent serially if the USART is configured correctly. However, whenever we send a character, we need to make sure that the previous one is sent; otherwise, it will be overwritten. In order to avoid that, we have four different approaches:

- Poll the TXIF interrupt flag found in PIR1 register which is set when the TXREG is empty
- Poll the TRMT flag found in TXSTA register which is set when the data when the transmission of data is completed (This is the approach used in the program).
- Enable the USART transmitter interrupt and write an interrupt service subroutine to send the next character.
- Insert a sufficient delay between writing characters to the TXREG. For example, if the speed is 9600 bps, this implies that the time required to transmit a 10-bit frame is 10/9600, which is 1.041ms approximately.

Step 3: Waiting for Commands from Control Computer

After the whole message is sent, the code goes into an infinite loop waiting to receive commands from the control computer. The commands will be received serially and placed in the RCREG. When a whole frame is received in the RCREG, the RCIF flag in the PIR1 register is set. So, in order to decide whether a command is received or not, we can:

- Poll the RCIF flag in the PIR1 register.
- Enable the USART receiver interrupt and write an ISR to read the RCREG when the interrupt occurs (This is the approach used in the program).
- Read the RCREG periodically at sufficient time interval.

While receiving the commands from the control computer, it is important to check if there are errors in the received data. There three types of errors in serial communication:

- Framing errors – This error 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 bit 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 and test this bit to check if we are receiving the data correctly.
- 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!

- Parity Errors – This error is used to detect odd number of erroneous bit transmissions. This is done by enabling the 9th 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!

Step 4: Displaying Commands

When a command is received by the PIC, it has to display them on a common-anode 7-segment displays. The received commands are basically the numbers 0 through 9. Thus, the program uses a lookup_TABLE table to convert the command into 7-segment code and output it to PORTD that is connected to the display.

The code for the whole program is available in the USART Example.ASM file and it is listed below for your reference. Study the code before you start with the labsheet.

```
;*****************************************************************************************************
; 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 segment 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
__CONFIG _DEBUG_OFF&_CP_OFF&_WRT_HALF&_CPD_OFF&_LVP_OFF&_BODEN_OFF&_PWRTE_OFF&_WDT_OFF&_XT_OSC

;*****************************************************************************************************
    include "p16f877A.inc"
;*****************************************************************************************************
; User-defined variables
    cblock  0x20
            WTemp           ; Must be reserved in all banks
            StatusTemp      ; reserved in bank0 only
            Counter
            BLNKCNT
            INDEX
        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
        bsf         PIE1,RCIE       ; Enable Receiver Interrupt

        banksel     INTCON
        bsf         INTCON, GIE     ; Enable global and peripheral interrupts
        bsf         INTCON, PEIE

        banksel     TRISD           ; PORTD is used to display the received commands
        clrf        TRISD
        clrf        TRISA
        bcf         TRISC, 6        ; Configuring pins RC6 as o/p, RC7 as i/p for
        bsf         TRISC, 7        ; serial communication
        movlw       6
        movwf       ADCON1

        banksel     PORTD
        clrf        PORTD
        clrf        PORTA
        clrf        PORTA
        return
;*********************************************************************************
Main
        call        Initial
MainLoop
        clrf        INDEX           ; Prepare to send first character in the message
                                    ; MSG = 0, then incremented by on to access every
                                    ; character in look up table.
SEND
        movf        INDEX, 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     INDEX
        incf        INDEX, F            ; Move to next character in string
        movlw       .33             ; Check if the whole message has been sent
        subwf       INDEX, 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                        ; push W and STATUS
        btfsc       PIR1, RCIF      ; Check for RX interrupt
        call        RX_Receive
        pop                         ; pop W and STATUS
        retfie
;*********************************************************************************
RX_Receive
        ; 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'
```

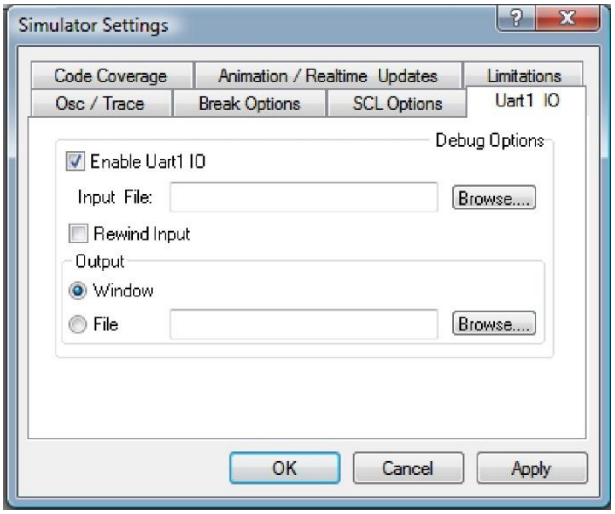
```
159         RETLW       B'10000000'     ;'8'
160         RETLW       B'10010000'     ;'9'
161     ;*****************************************************************************
162     Message
163         addwf       PCL, F
164         retlw       A'M'
165         retlw       A'a'
166         retlw       A'c'
167         retlw       A'h'
168         retlw       A'i'
169         retlw       A'n'
170         retlw       A'e'
171         retlw       A' '
172         retlw       A'r'
173         retlw       A'e'
174         retlw       A'a'
175         retlw       A'd'
176         retlw       A'y'
177         retlw       A' '
178         retlw       A't'
179         retlw       A'o'
180         retlw       A' '
181         retlw       A'r'
182         retlw       A'e'
183         retlw       A'c'
184         retlw       A'e'
185         retlw       A'i'
186         retlw       A'v'
187         retlw       A'e'
188         retlw       A' '
189         retlw       A'c'
190         retlw       A'o'
191         retlw       A'm'
192         retlw       A'm'
193         retlw       A'a'
194         retlw       A'n'
195         retlw       A'd'
196         retlw       A's'
197         END
198     ;*****************************************************************************
```

3. Simulating and Testing in MPLAB

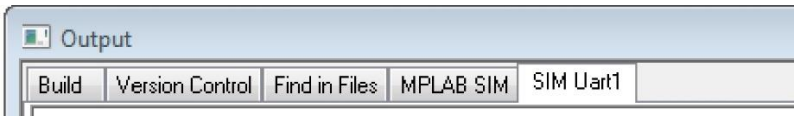
At first glance, you might think that you cannot test your code unless you have a physical control PC and a machine at home!! Surely this is not feasible. Therefore, we will now introduce you to testing USART serial communication in MPLAB IDE.

To test transmitting the data from the PIC, do the following after you build your project:

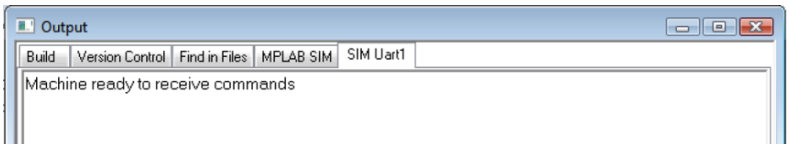
1. From the *Debugger* Menu, *Select Tool* → *MPLAB SIM*.
2. From the *Debugger* Menu, Settings → select *UART1 IO*.
3. The following screen will show up.



4. Select *Enable UART IO*.
5. Choose to show the output on *Window*. Click OK.
6. 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.

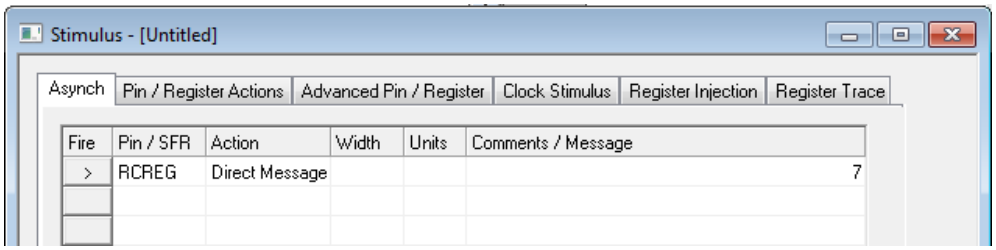


Now run the program. You will see that the message has appeared in the Uart1 IO window which we have already enabled.



To test receiving the commands from the computer, we will use the Stimulus tool that we introduced in Experiment 3. The procedure will be revisited here again:

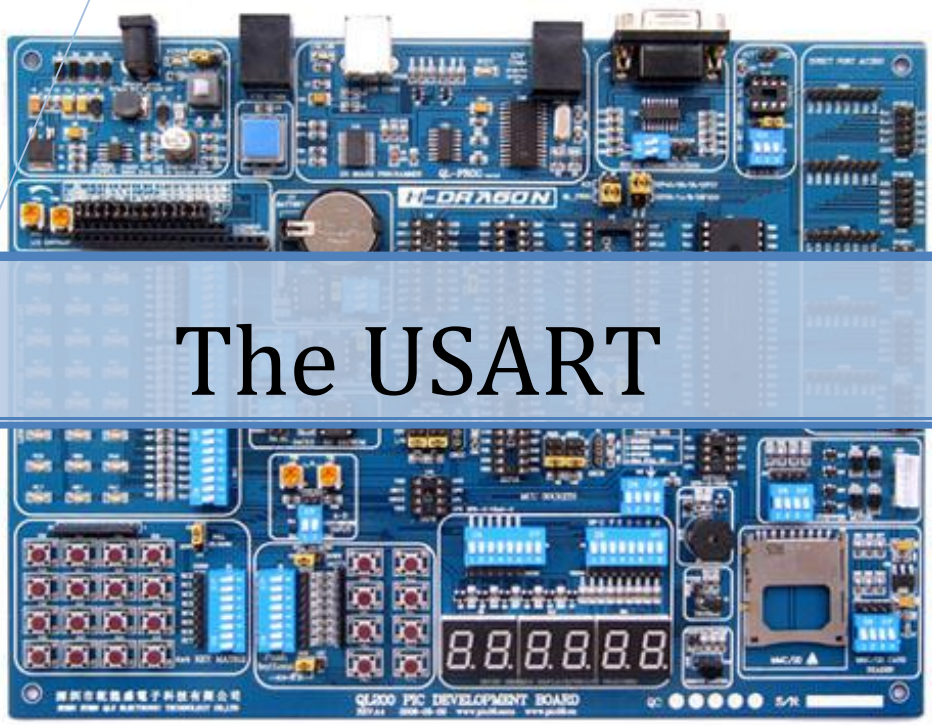
1. Debugger → Stimulus →New Workbook
2. In the ***Async*** tab choose ***RCREG***, and set the action as ***Direct Message,*** in the Message field type in the character you wish to send as shown below.



3. Place a break point at instruction **goto IntService.**
4. Run the program and wait until the message "Machine is ready to receive commands" is shown on the output window.
5. Now, click on Fire. The program execution stops at goto IntService. Step into the code.
6. Once you finish stepping in the RX_Receive subroutine, you should see that PORTD has the value of "11111000" which is the code for 7.

Labsheet 8

University of Jordan
School of Engineering
Department of Computer Engineering
Embedded Systems Laboratory 0907334



The USART

Name:

Student ID:

Section (Day/Time):

COMPUTER NAME:

UNIVERSITY OF JORDAN
SCHOOL OF ENGINEERING
DEPARTMENT OF COMPUTER ENGINEERING
EMBEDDED SYSTEMS LABORATORY CPE0907334

Labsheet 8: The USART

Name: **Student ID:**
Section:

(Pre-lab) Part1: The operation of the USART Module and the related TXSTA, RCSTA and SPBRG settings

Q1) What are the values of the following registers: TXSTA, RCSTA and SPBRG that should be initialized with, to configure USART module as follow:

1. Setting the baud rate to 1200 with low speed.
2. Enable s continuous receive and transmitter module.
3. 8-bit transmission and reception.
4. Set up the USART port for Asynchronous operation.

Register	7	6	5	4	3	2	1	0
TXSTA								
RCSTA								
SPBRG								

Q2) Write the necessary instruction(s) required for enabling the USART receiver interrupts.

Q3) For the USART example explained in the tutorial, what changes should be made to the code in order to send the message "Machine is Ready"?

Part 2: Coding

In the Labsheet 8 Proteus Circuit, you will find two microcontrollers; PIC1 and PIC2, that have 4 MHz clock and are supposed to communicate serially. PIC1 has four switches connected to the lower 4 bits of PORTB while PIC2 has two switches connected to RC1 and RC0. PIC1 has a green and red LEDs connected to RC1 and RC0, respectively, and PIC2 has a 7-segment display connected to PORTB.

When the system starts, PIC1 is supposed to read the four switches and send them to PIC2 serially and then waits for PIC2 to send an acknowledgment. If the received acknowledgement contains 0xCC, the green LED is turned on. In case the acknowledgment has a value of 0xEE, the red LED is on. PIC1 repeats these operations indefinitely.

When PIC2 receives the values, it should read the two switches connected to it and then add the read value to the received value. If the result is less than 10, then it is displayed on the 7-segment display connected to PORTB of PIC2 and an acknowledgement message containing the value of 0xCC is sent to PIC1. Otherwise, PIC2 should display 0xE on the display and send an acknowledgement message with value 0xEE to PIC1.

Hints:

- **Note that the transmitter and the receiver in the USART in PIC1 and PIC2 should be enabled.**
- **Use baud rate 19200 assuming BRGH=1 and Fosc = 4.000 MHz.**

Ask your engineer to check the run.

Labsheet 3-B

University of Jordan

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

Lab 3 Hardware Exercise

Name:
Student ID:
Section (Day/Time):

Lab 3 Hardware Exercise

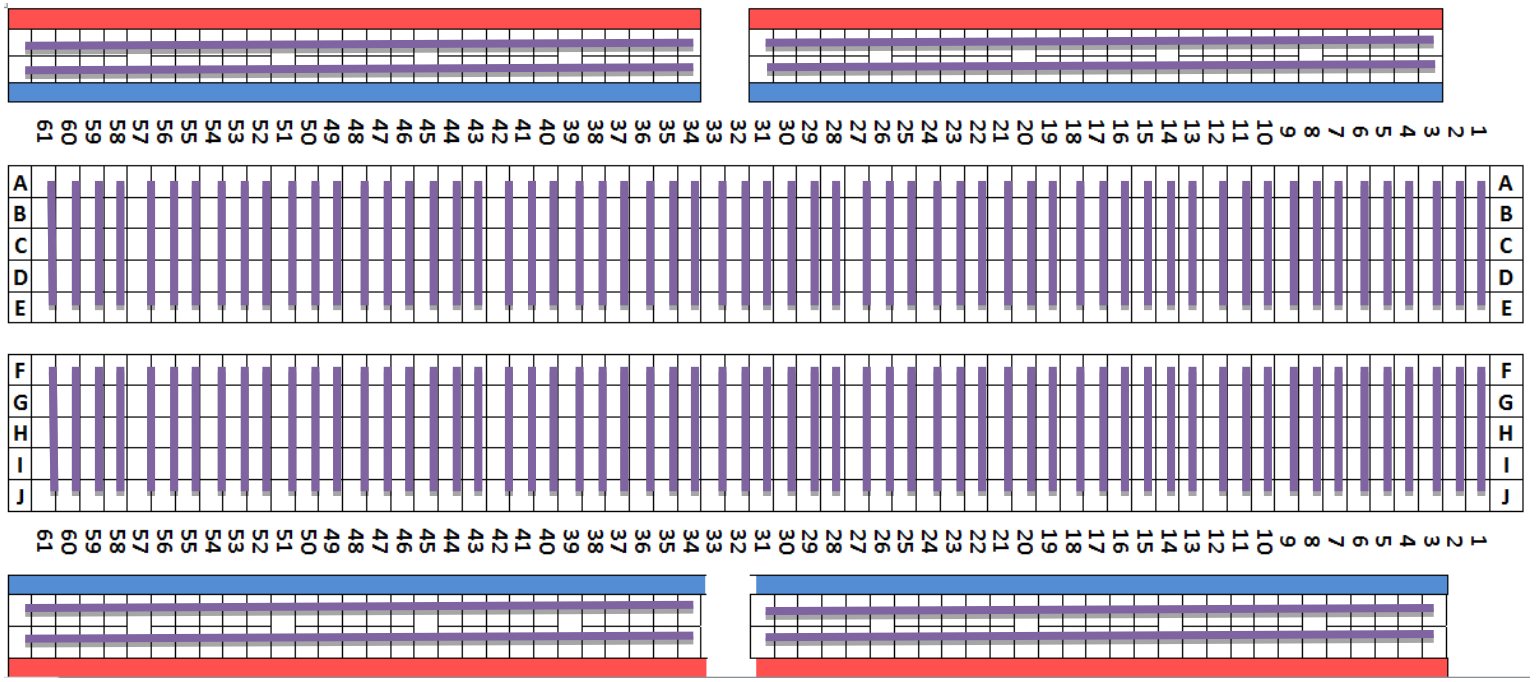
In this tutorial we will guide you through the steps to build your first hardware circuit, it is a simple circuit based on 16F84A PIC which drives a 7-segment display to show the numbers from 0 to 9 continuously. **The PIC is already programmed and placed for you!**

This basic circuit uses the following components (you should have read the "Guide to Hardware I" by now and familiarized yourself with all the hardware components listed before coming to the lab!):

We will use a 16F84A PIC, A 7805 regulator, two 22pF capacitors, a 7-segment display and required resistors!

Since this is your first hands-on experience with hardware, and to ensure that the circuit works with you we will specify all the interconnects you need, this is necessary for many students have no basic foundation in electronics or basic circuit construction.

The Breadboard layout below is to refresh your knowledge about breadboard structure: (The line in Purple represents that these spaces are internally one node)



Follow the following steps to construct the circuit

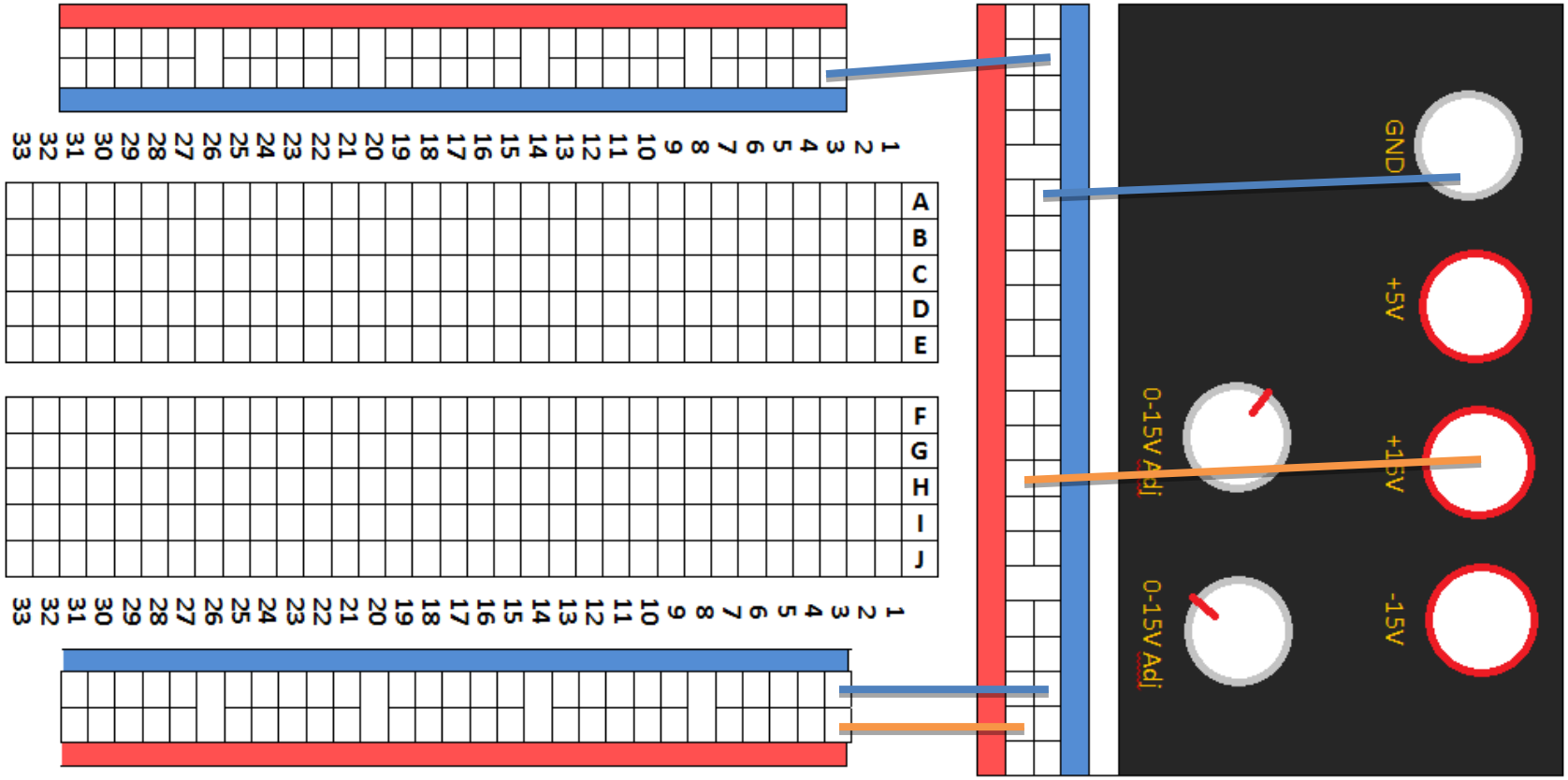
Step 1: Ensure that the power connections are as follows, We will use the +15V adjustable power knob as Input power source to the regulator, place the knob at the direction indicated, this means an approximate power of 7 – 8 volts!

In this experiment we will use the following color coding:

Blue: to connect to ground!

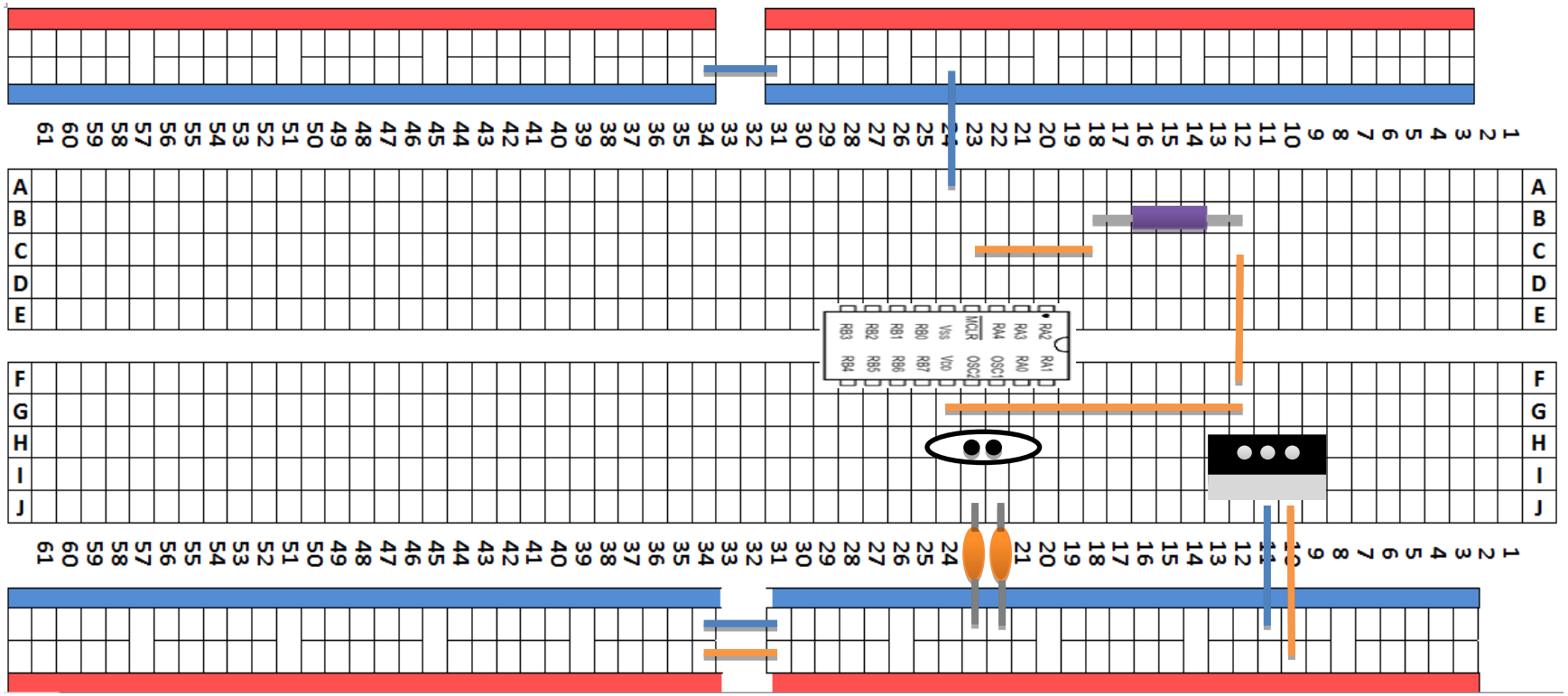
Orange: To connect to power supply

Green: to connect between all other components!

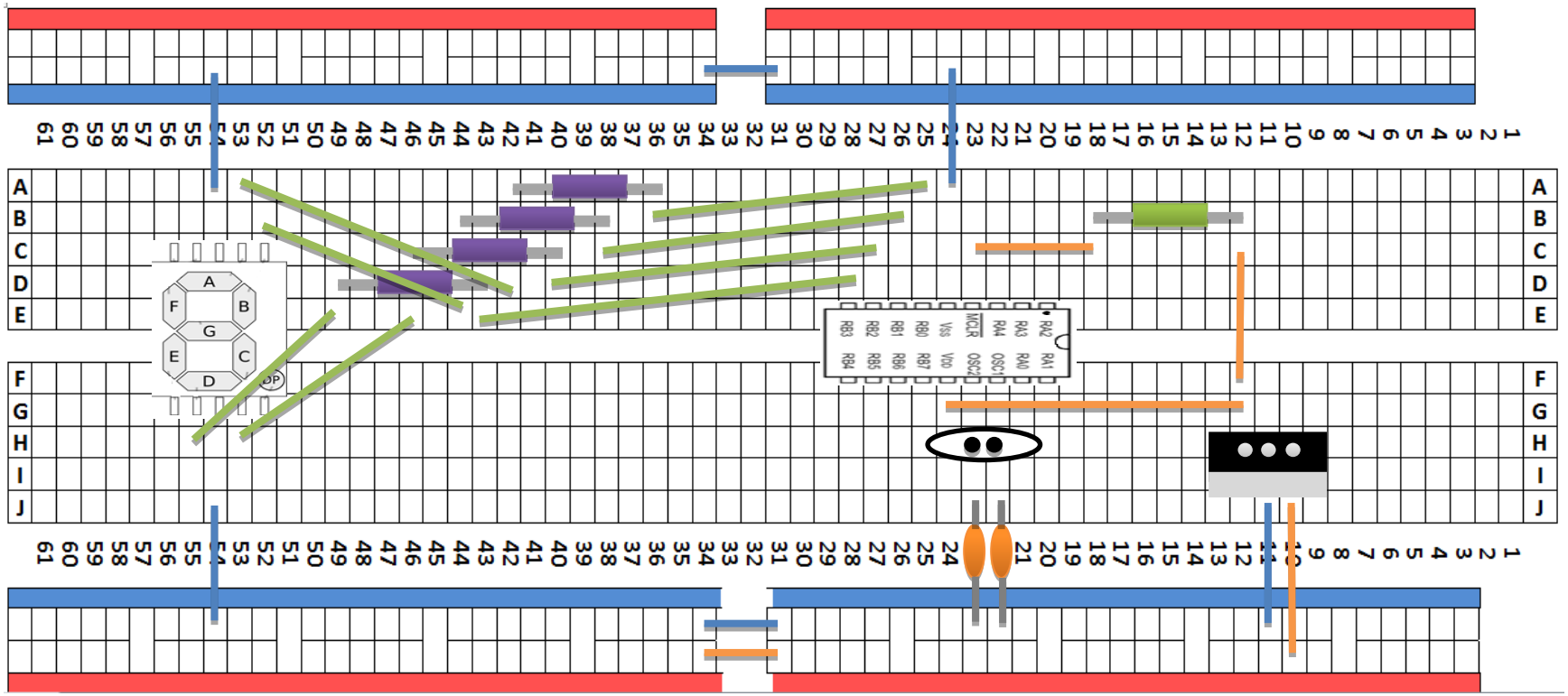


Step 2: Interfacing all the necessary components to power up the PIC and provide oscillation! Connect the 7805 regulator, the 4MHz oscillator (with the required 22pF capacitors) and the MCLR pin as follows:

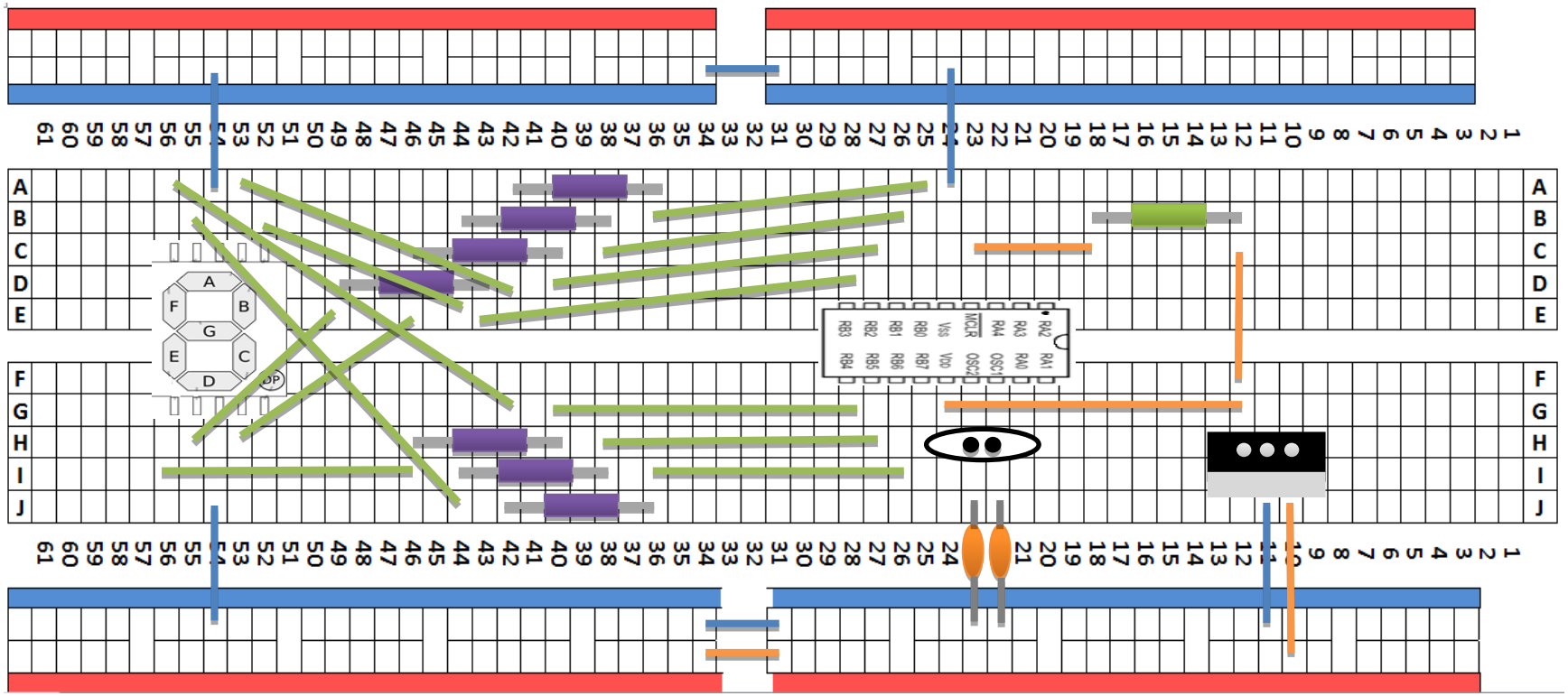
The dots inside the regulator and the oscillator indicate that the pins are directly below them such that you will know where exactly to place them!

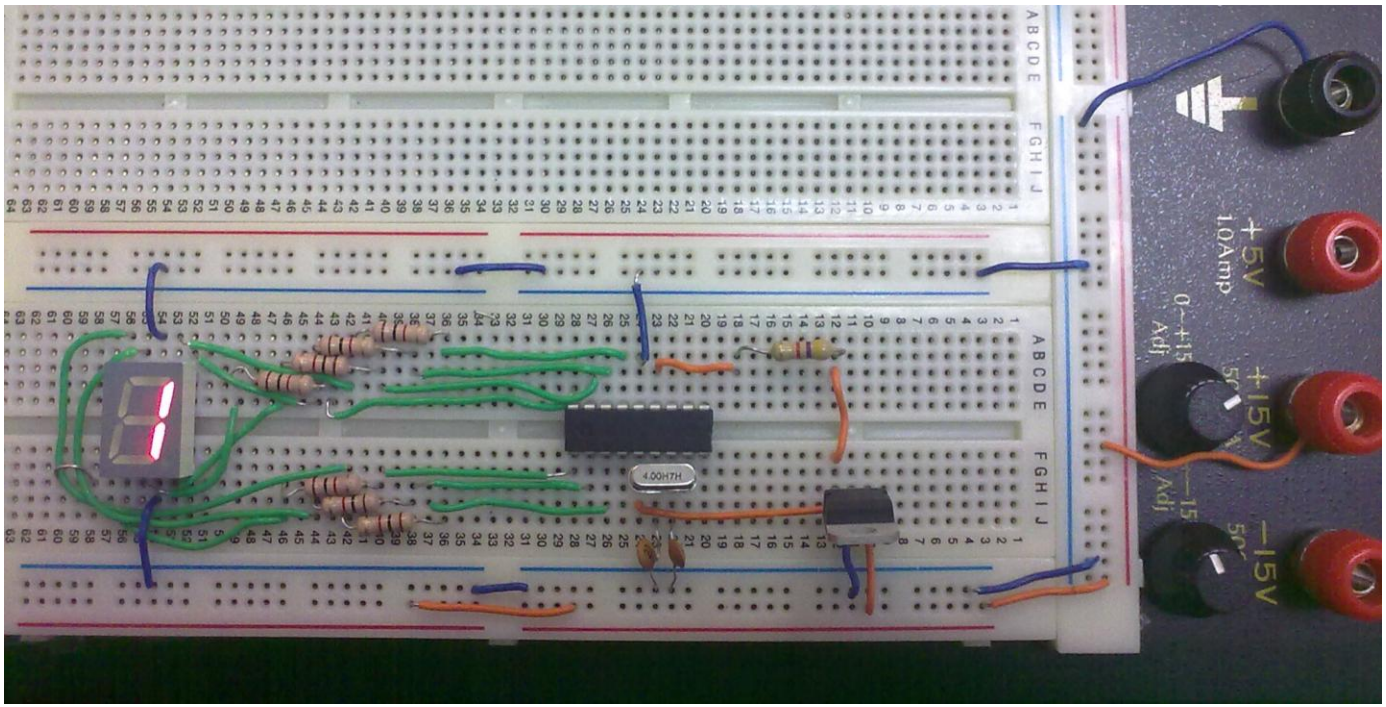


Step 3: Interfacing RB0 – RB3 to segments A to D! Note that the resistors are needed to limit the current to the LEDs making up the 7-segment such that it will not burn out (Note: some 7-segments have high internal resistors values and do not necessarily require external resistors!)



Step 4: Interfacing RB4 – RB6 to segments E to G!





Circuit on breadboard

Your circuit on breadboard should be similar to this picture

Ask your engineer to check the circuit.

GUIDE TO HARDWARE I

Required Study Material

Prepared by Engrs. Ashraf Al-Suyyagh – Enas Ja'ra

We present this guide to serve as a reference for many electrical and electronic components which students are to use throughout the lab course and the final project. We attempt to cover all required material such that all students will be on the same level of basic knowledge. The components are presented in the context of their use in embedded systems where they are interfaced with microcontroller devices.

1. CLOCKING SOURCES

Microcontrollers need clocking sources to synchronize their functions. An ideal clock is a square wave function. There are two components which provide such capability: oscillators and resonators. Basically the faster the clocking source; the faster the processing speed. However, fast processing also requires more current and therefore generates more heat. PIC 16F series can operate up to 20MHz whereas the 18F series has an operation speed of up to 40MHz. Details of operation principles, interfacing and calculations are presented in these following subsections

1.1. Oscillators

Oscillators usually built from crystal (most notably Quartz crystal– as in wristwatches) have simple operation principle: *The use of the mechanical resonance of a vibrating crystal of piezoelectric material to create an electrical signal with a very precise frequency*. To make things clear: we will start to define terms and simplify operation principles; a piezoelectric material is one which has the characteristic of changing shape when voltage or an electric field is applied to it then reverts back to its original shape once the induced voltage/electric field is removed. While switching back to its original state, the material itself generates an electric field and thus a voltage with very precise frequency which is the oscillator frequency. So simply, to get the oscillator to work, feed it with voltage at one pin and use the output frequency from another. In this manner, the oscillator can be modeled as an RLC circuit with a specific resonance frequency as you should have already learnt in the Circuits II course. Yet, since crystals are **mechanical** devices which vibrate at their resonance frequency, they are not that precise, that is, they don't produce an ideal square wave function of which the high level of the signal is fixed at 50% of the period's time, and instead it can be any time in between 40% - 60% of the period. Therefore, to account for such errors, the clock signal is divided by a certain fixed value to minimize error effects, in the PIC MCUs, the input clocking source is divided by four. In that manner, every four pulses of the original signal will generate one pulse in the new signal.

Figure 1 below further clarifies the idea; the top signal is an ideal square wave signal, the duty cycle of which is exactly half the period. Notice how the middle signal – the actual signal as generated from a clocking source – deviates from the ideal, the duty cycle differs (red). By dividing the cycle by a fixed value – bottom signal – we minimize the errors for the new signal hides the frequent changes in between successive pulses.

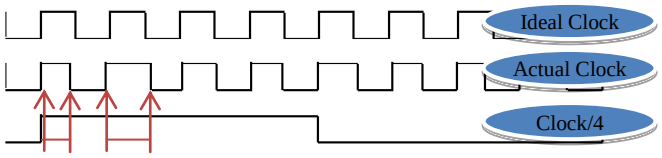


Figure 1 - Clock

Oscillators come in variety of form factors and speeds. Figure 2 shows two common form factors of oscillators, the 2-pin oscillator (left) and 4-pin oscillator (right). You can place the 2-pin oscillator in either direction on the OSC pins of the microcontroller, the 4-pin oscillator only uses three pins and the forth is not connected, the pins are GND, Vcc and output. Refer to the datasheet of the oscillator to determine which pin is which.

Figure 2 - Two Common Shapes of Oscillators

After connecting an oscillator, one should explicitly specify to the PIC which oscillator speed range and type it should expect. This option can be set in either the MPLAB configuration bits window prior to programming or by explicitly specifying it in the configuration word in the source code. There are four options:

- **XT** – Crystal: 1-4 MHz
- **HS** – High Speed: >= 4 MHz, and with ceramic resonators.
- **LP** – Low Power: <= 200 KHz,
- **RC** – Resistor-Capacitor (if you build the resonance circuit by yourself)

Along with the crystal, two capacitors of approximately (10-33) pF are required, crystal needs loading capacitors to work at the exact operating frequency (i.e. to get a stable oscillation from the crystal oscillator) and for noise immunity. An important note though is that the operating frequency is not fixed and that it varies with temperature. The advertised frequency is usually specified at room temperature 25°C, clocks slow down when temperatures increase or decrease from the nominal room temperature. For accurate timing one needs to know the operating frequency at different temperatures, for this we can use the following formula (assume all other parameters are at their recommended values):

$$F = F_0(1 - P\ x\ (T\text{-}T_0)^2)$$

Where:

- **T** is the expected temperature in Celsius
- $\mathbf{T_0}$ room temperature 25°C
- $\mathbf{F_0}$ advertised oscillator frequency
- **F** actual frequency at temp T
- **P** is the frequency stability coefficient (obtained from datasheet – units in ppm)

Example:

A 32 kHz oscillator with a frequency stability coefficient of 0.004 ppm running in arid environment where average temperature is 35°C will actually have an oscillation frequency of:

$32000(1 - 0.004\ (35\text{-}25)^2) = 19.2\text{kHz}!!$

From the above example, we clearly show the importance of considering temperature effects upon the frequency of operation.

Figure 3 shows a typical crystal interfacing to a PIC

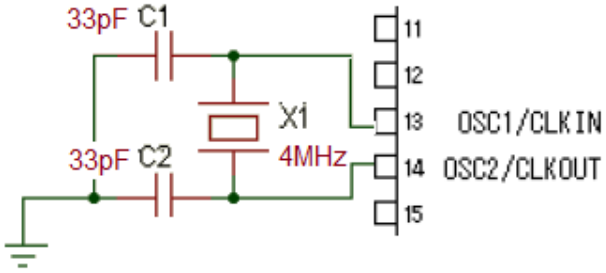


Figure 3 - Interfacing a 4MHz Crystal to PIC16F877A

1.2. Resonators

Resonators are made of high-stability piezoelectric ceramics and share the same operating principles of oscillators but differs in that it consists of a voltage-variable capacitor that acts in some ways like a quartz crystal. The thickness of the ceramic substrate determines the resonance frequency of the device. Resonators have either two or three pins. They need not loading capacitors. They have a similar connection as the oscillator in Figure 3, if a three lead resonator is used the middle pin is connected to GND. Figure 4 shows a typical resonator.

Figure 4 - A Typical Resonator

2. REGULATORS

A voltage regulator is an electrical regulator designed to automatically maintain a constant voltage level at the output given varying voltage at the input. Depending on the part number and manufacturer specifications, regulators can take a limited range of input voltages and produce a limited range as well. Regulators most often have metallic heat sinks attached to dissipate heat more efficiently. Many commercial regulators regulate fixed voltages, commonly 3, 5, 9, 12 and 15 volts. One must be cautious to the input and output currents to and from the regulator, too much input current than specified will overheat and eventually burn the device. Too much load current will have the effect of regulator output voltage drop!

There are two main series of regulators: the 78xx and 79xx series. The 78 represents a family of regulators which regulates positive voltages and the 79 family regulates negative ones. The xx part is the output voltage of the device.

Examples:

- 7805: 5V DC Regulator
- 7905: -5V DC Regulator
- 7808: 8V DC Regulator
- 7909: -9V DC Regulator

Regulators have three pins, one connected to the input voltage source, another to the circuit and the middle one is shared ground in between the input source and the circuit. Figure 5 shows a typical regulator.

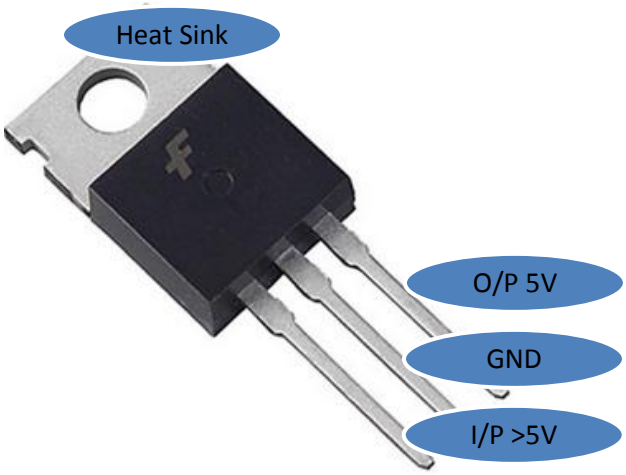


Figure 5 - A Typical 7805 Regulator

2.1. More on Regulator Heat Sink

The heat sink is a component designed to lower the temperature of an electronic device by dissipating heat into the surrounding air. As a general rule the input voltage should be limited to 2 to 3 volts above the output voltage. The LM78XX series can handle up to 36 volts input, be advised that the power difference between the input and output appears as heat. So heat which will be dissipated by the chip during the voltage regulation process. This can cause the chip to heat up, and so a heat sink is often used to speed up heat removal and prevent overheating.

Figure 6 shows a typical in-circuit connection for the 7805 regulator. A couple of coupling capacitors (between 10 uF and 47 uF) are required on the input (V-IN) and output (V-OUT) and connected to ground. Coupling capacitors are used for good regulation and to reduce unwanted AC signals riding on DC supply circuits (Noise)

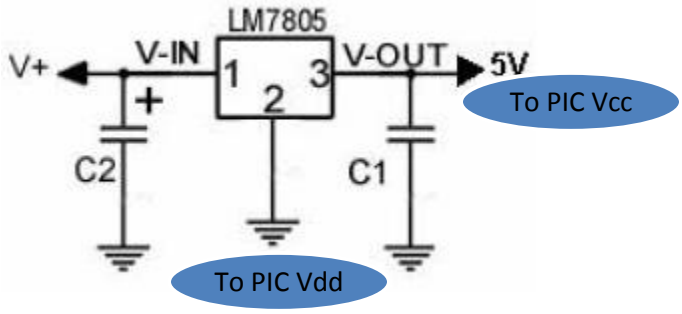


Figure 6 - 7805 Circuit Diagram

3. PIC RESET CIRCUIT

As you should have already learnt in the course, PIC MCUs already have a master clear pin called MCLR, keep in mind that this is an active low pin. Therefore, PIC reset circuitry is simply constructed by wiring a switch to MCLR, and when pushed gives logical '0' or GND to this pin. This has an effect of resetting the microcontroller, clearing all RAM and starting program execution from the beginning. A **pull up resistor** circuitry is used to hold the input at logic "1" state as long as the reset button is not pressed. Figure 7 shows the circuit diagram of the reset circuitry.

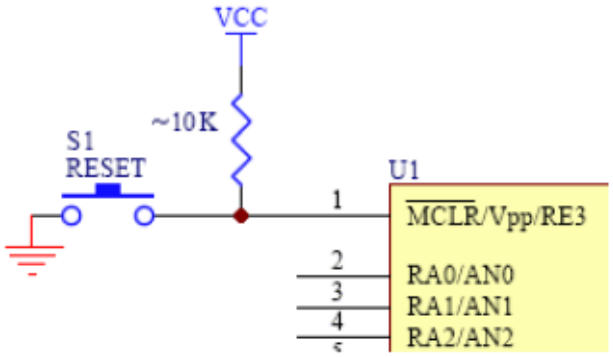


Figure 7 - PIC Reset Circuit

4. Pull-Up and Pull-Down Resistors

Pull-up resistors are used in electronic logic circuits to ensure that inputs to logic systems settle at expected logic levels if external devices are disconnected. The idea of a pull-up resistor is that it weakly "pulls" the voltage of the wire it's connected to towards 5V (or whatever voltage represents logic "high"). However, the resistor is intentionally weak (high-resistance) enough that, if something else strongly pulls the wire toward 0V, the wire will go to 0V. Pull-down resistors operate in a similar fashion where they are initially pulled down to logic 0 through a connection to ground, but when a source pulls it up toward logic high it will change state. Pull up and pull down resistors are used with switches and push buttons to fix the state of the pin connected to the switch at a predetermined state and not be kept floating. Pull up and pull down resistors take a minimum value of 4.7kΩ.

5. Light Emitting Diodes

Light emitting diodes or LEDs are semiconductor light sources used as indicator lamps in many electronic devices. Modern versions are available across the visible, ultraviolet and infrared wavelengths, with very high brightness and come in a variety of shapes and sizes. The physics behind LED operation is covered in the Electronics I course and will not be offered here. Figure 8 shows the different color spectrum of LEDs.

Figure 8 - Different Colors of LEDs

In order to switch a LED on, forward current must pass from the anode to the cathode, but how to determine which pin of the LED is anode and which is cathode, generally, there are two ways:

1. The longer lead is anode, the shorter is cathode.
2. The cathode has a flat surface as shown in Figure 9

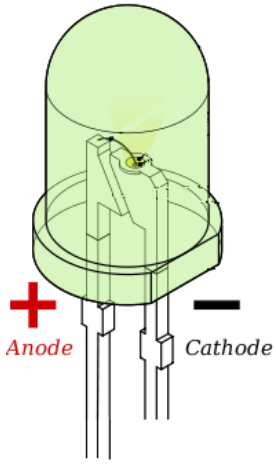


Figure 9 - Determining Cathode and Anode in LED

Resistors with values in between 220Ω to 1kΩ are placed in between the voltage source (often 5 to 9V or even more) and the anode to limit the current entering the LED or else it will burn. The lesser the resistor value, the brighter the LED shines (Ohms Law). In this case these resistors are called current limiting resistors.

Figure 10 shows how to interface a LED to the PORTC pin 1

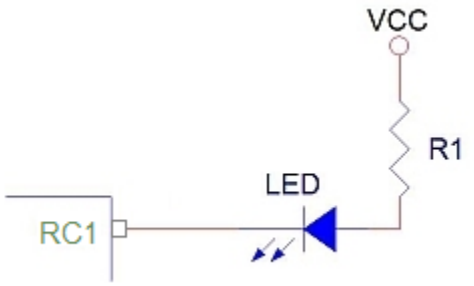


Figure 10 - LED Interfacing

Not only are LEDs used as discrete components but are also the building blocks of LED Matrices and 7-Segment displays. Figure 11 shows an example of a LED matrix.

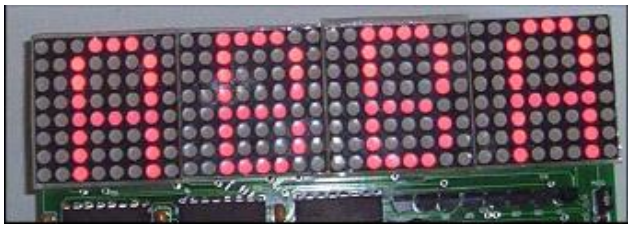


Figure 11 - LED Matrix

6. Seven-Segment Displays

A Seven-Segment display, as its name implies, is composed of seven segments (or technically of seven LEDs) which can be individually switched on or off. This ability to individually control each segment and the layout in which these segments are distributed allows for the representation of the numerals and some characters. If the anode ends of all LEDs are connected together, it is called common anode display. If the cathodes of all LEDs are connected together, it is called a common cathode display. To switch a LED on in a common cathode configuration, you have to send logic high to the segment pin. Conversely, to switch a LED on in a common anode configuration, you have to send logic low to the segment pin.

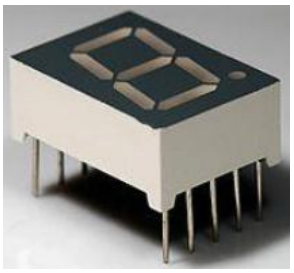


Figure 12 - A Single Unit 7-Segment Display

Seven-Segment displays can be purchased in single units encompassing one, two, three or even four displays in the same package. Digit Multiplexing techniques are widely used to allow for the multiple displays to share the same segment pins simultaneously while each displaying a different numeral or character. Figure 12 shows a typical single unit seven segment display while Figure 13 shows the typical layout of segment pins for the common cathode and common anode configurations.

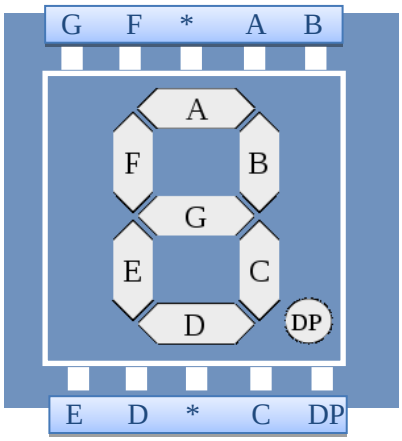


Figure 13 - Seven-Segment Display
*Means Vcc for common Anode, and GND for common Cathode

Interfacing a Seven-Segment display is independent of the type of the module, whether it is common anode or common cathode, only the logic level sent to the display differs. Finally, since the display is basically LEDs, current limiting resistors are used for each segment.

7. Switches and Push Buttons

Switches and pushbuttons have similar operation, to switch the input level between two alternating levels, the only difference is that a push buttons only retains

the level as long as it is pressed and reverts back to its prior state once the press effect is gone.

7.1. Switches

A switch is an electrical component which can break an electrical circuit, interrupting the current or diverting it from one conductor to another. There are four types of switches:

- SPST Single Pole, Single Throw: SPST is simple on-off switch. This type is simply used for turning something on and off
- SPDT Single pole, double throw: SPDT switches are useful if you want to supply some instrument with two different voltages or divert current between two different paths.
- DPST Double pole, single throw: A Double-pole Single-throw switch is simply two SPST switches together. It allows you to switch two separate circuits on and off at once.
- DPDT Double pole, double throw: DPDT switches have six terminals and allow one to switch poles between two different circuits.

Figure 14 below shows typical SPDT switch.

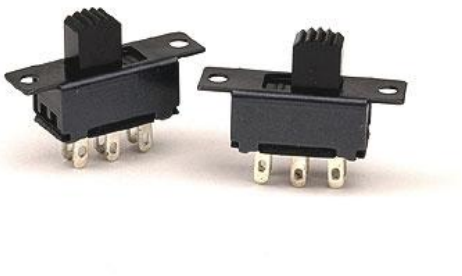


Figure 14 - SPDT Switches

Switches use pull-up or pull down resistors to hold the input voltage supplied to the PIC at a predetermined level, only when the switch is used does the voltage level change. Figure 15 shows the circuit diagram of interfacing an SPST switch to PIC. Note that the pull up or pull down resistors take a minimum of 4.7kΩ, you can determine the exact value using Ohm's Law.

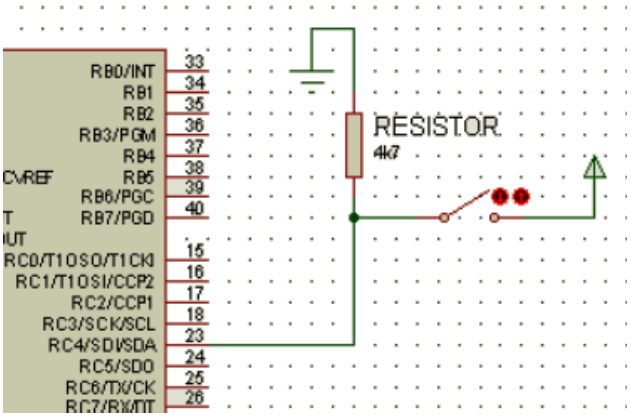


Figure 15 - Interfacing an SPST Switch – Same circuit is used to interface a push button.

7.2. Push Buttons

There are two types of push buttons:

- Normally closed push button (abbreviated NC) is one that normally gives logic one and when pressed gives logic zero.
- Normally open push button (abbreviated NO) is one that normally gives logic zero and when pressed gives logic one.

Push buttons are interfaced in the exact same way as an SPST switch shown in Figure 15. Figure 16 shows a push button.

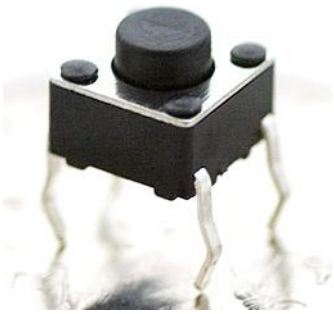


Figure 16 - A Push Button

7.3. Mechanical Switch De-bouncing

Push-buttons and switches are often used to provide input to digital systems. However, mechanical switches do not open or close cleanly. When a switch is pressed it makes and breaks contacts several times before settling into its final position. This causes several transitions or "bounces" to occur. To correct this situation a **de-bounce** circuit is connected to the switches, thus removing the

series of pulses generated by the mechanical action of the switch. Figure 17 shows a circuit which suffers from bouncing effects.

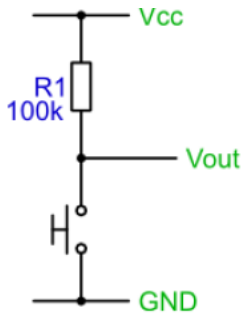


Figure 17 - A Circuit Suffering from Mechanical Bouncing Problem

Figure 18 shows an oscilloscope captured image clearly showing the bouncing effect.

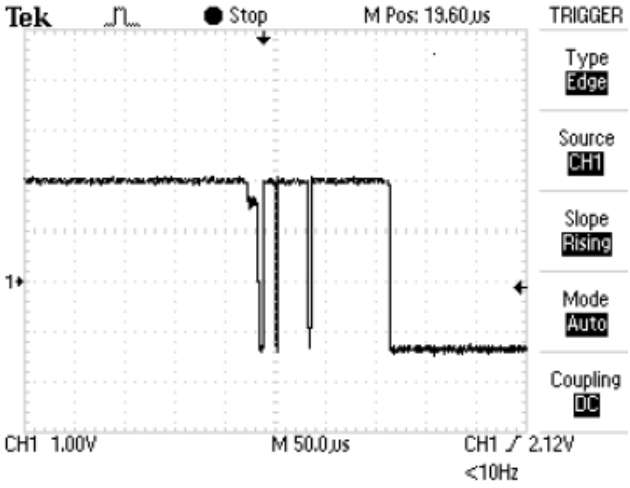


Figure 18 - Bouncing Effect. Note that it approximately lasts for 150 us

Solutions

There are two solutions to the bouncing problem: hardware and software approaches.

Hardware de-bouncing: The most basic circuit used to de-bounce a switch is shown in Figure 19. It consists of a resistor and a capacitor in series. The resistor and capacitor values must be chosen such that the RC time constant is greater than the bounce time. The output is then connected to a Schmitt trigger. At the start of operation the capacitor is charged to Vcc and the output is at 5 volts, when the switch is closed, the capacitor starts discharging smoothly and this filters out the bounces. The Schmitt trigger is a comparator which gives a high output if the input value is over a certain threshold, and a low output if below. In this case, the Schmitt trigger is necessary because the smoothed out value from the capacitor is neither high nor low but an exponential signal which digital devices don't understand, therefore it is up to the Schmitt trigger to convert it to logic highs and lows.

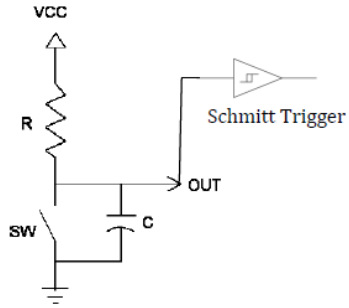


Figure 19 - A Hardware De-Bouncing Solution

Software de-bouncing: The basic idea is to read the switch input signal after some time interval guaranteed to be larger than the duration which the bouncing lasts and thus skip any short-lived bounces. In. Figure 18, one can read the signal after 200 µs.

Which is better: Software or Hardware de-bouncing?

It depends on your application needs; if time is critical and speed is important, you need not waste cycles in generating delays and therefore hardware solutions are preferable. If, however, you are developing a simple small-scale project where you want to reduce the hardware costs, then the software approach is better. All in all, you need to compromise and choose depending on your application and development needs

Introduction to Proteus

Prepeared by Eng.Enas Jaara

The PROTEUS Environment:

Proteus PIC Bundle is the complete solution for developing, testing and virtually prototyping your embedded system designs based around the Microchip Technologies ™ series of microcontroller. This software allows you to perform *schematic capture* and to *simulate* the circuits you design.

A demonstration on the use of *PROTEUS* will be given to you on this lab session, after that; you are encouraged to learn to use the software interactively.

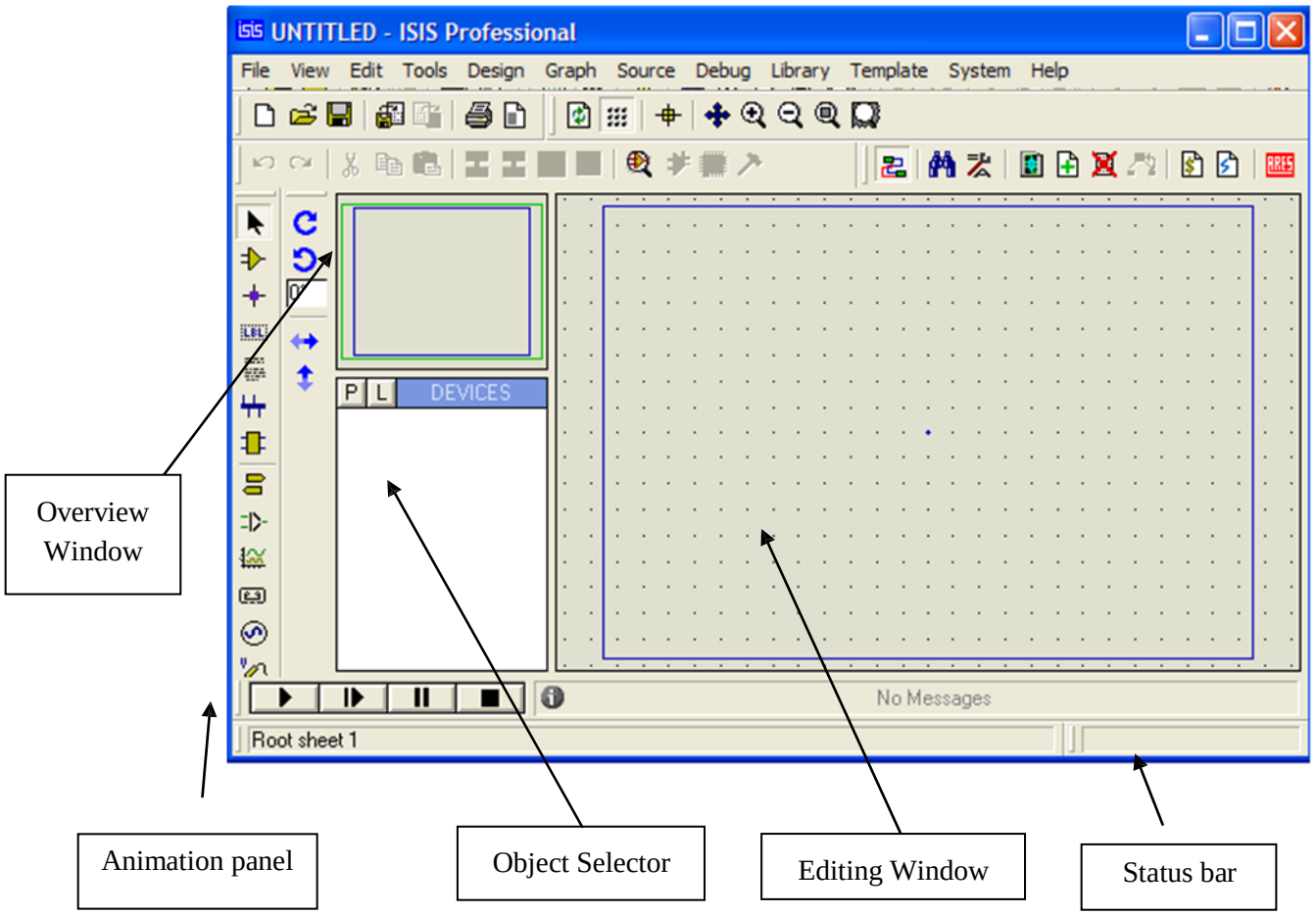


Figure 1. A screen shot of the Proteus IDE

Proteus How to Start

Drawing the Circuit

Start a fresh design, select New Design from File menu then the Create New Design dialogue now appears as shown in Figure 2 and 3. Select Default and press OK.

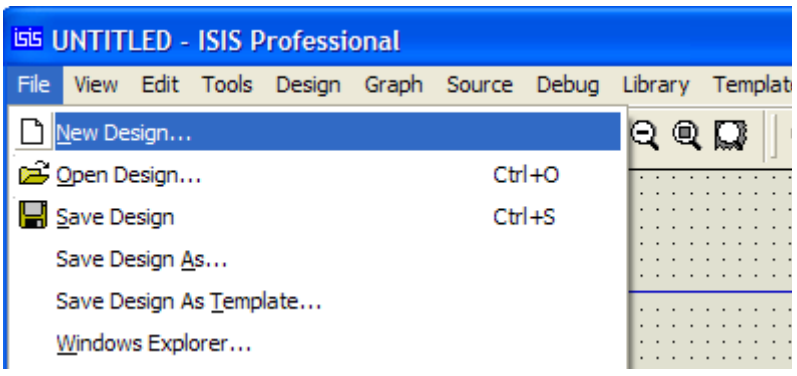


Figure 2

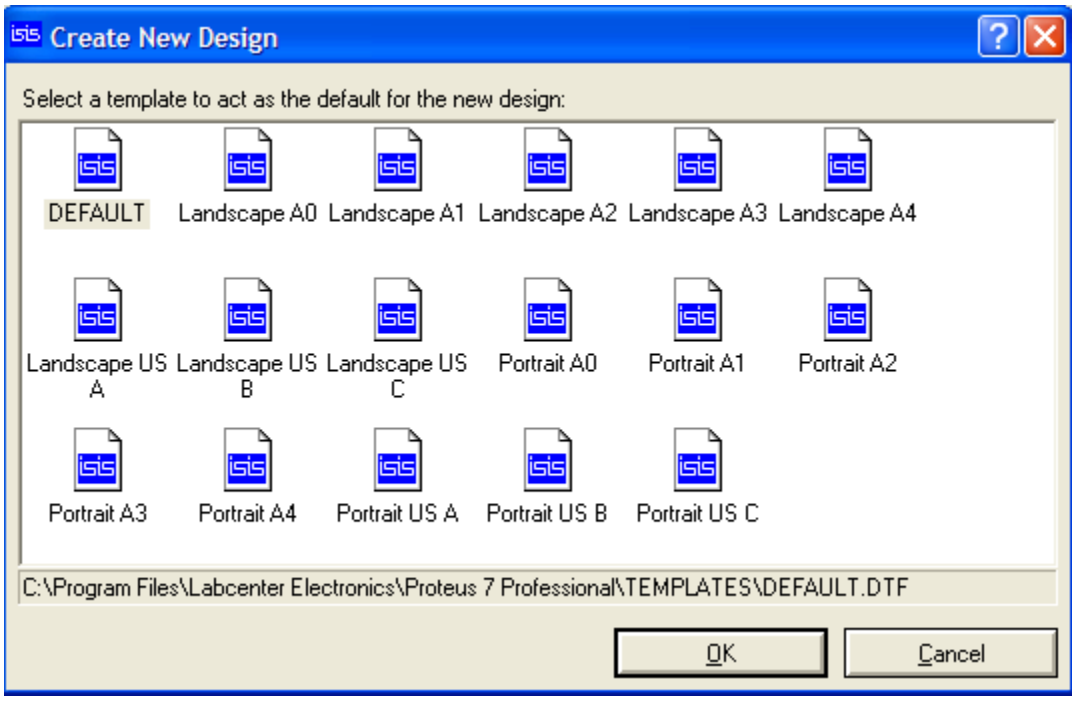


Figure 3

From the Library menu select Pick Device/Symbol see Figure 4 or Left click on the letter 'P' above the Object Selector as shown in Figure 5 to launch the Library Browser or Press the 'P' button on the keyboard. The Library Browser will now appear over the Editing Window see Figure 6.

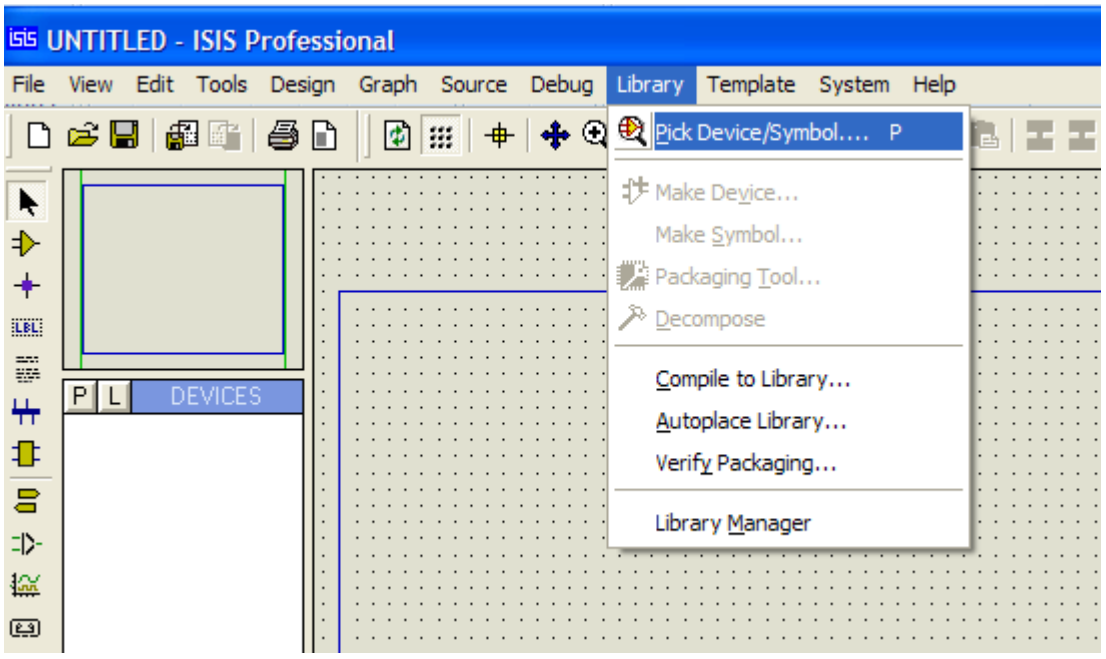


Figure 4

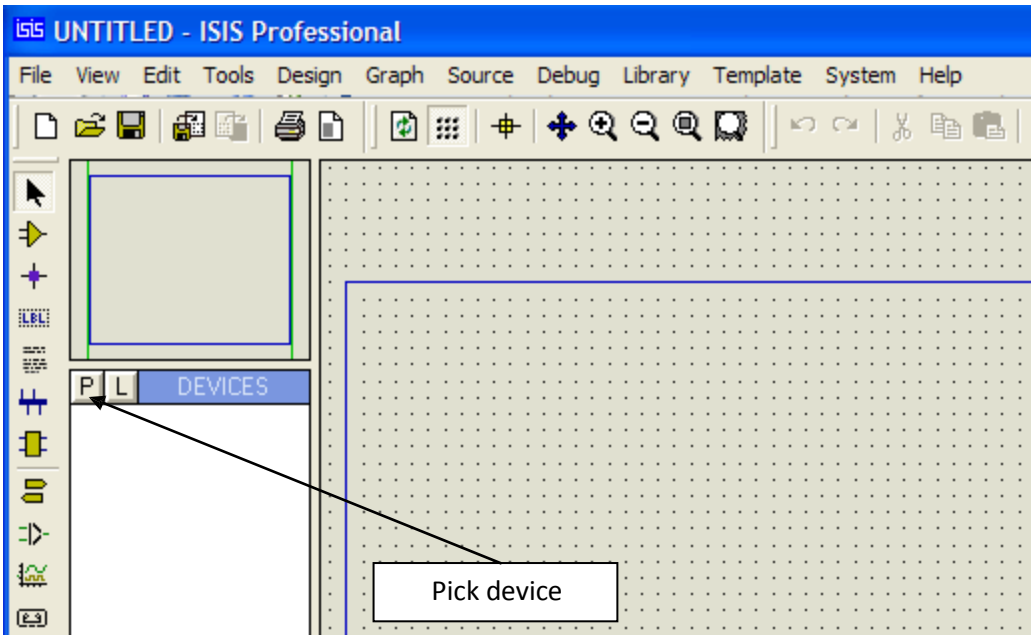


Figure 5

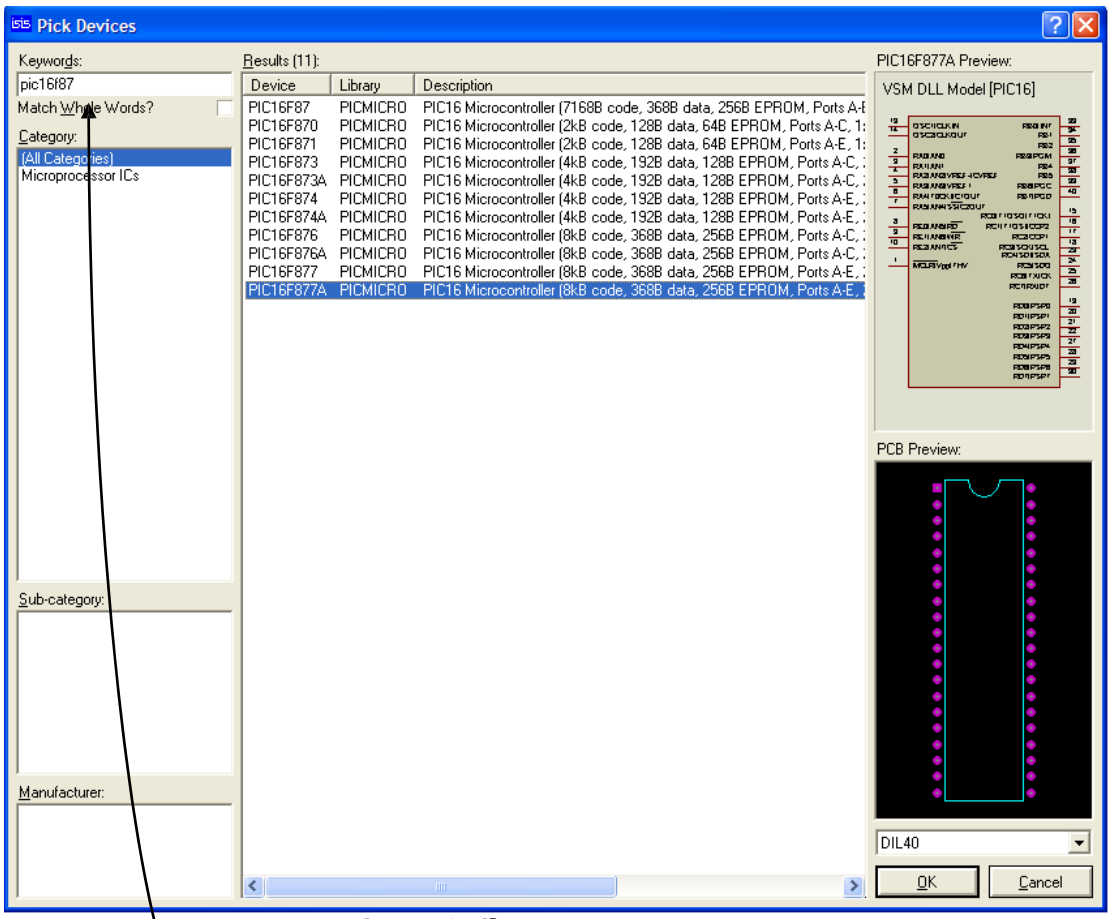


Figure 6 Library Browser

Type ' PIC16F877A ' in the Key words field and double click on the result to place the PIC16F877A into the Object Selector.

Type ' PIC16F877A ' in the Key words field and double click on the result to place the PIC16F877A in to the Object Selector. Do the same for the LEDs, Buttons, Crystal oscillator, capacitors, 7 SEG-COM-Cathode, Resistors.

Once you have selected all components into the design close the Library Browser and left click once on any component in the Object Selector
(This should highlight your selection and a preview of the component will appear in the Overview Window at the top right of the screen see Figure 7). Now left click on the Editing Window to place the component on the schematic - repeat the process to all components on the schematic.

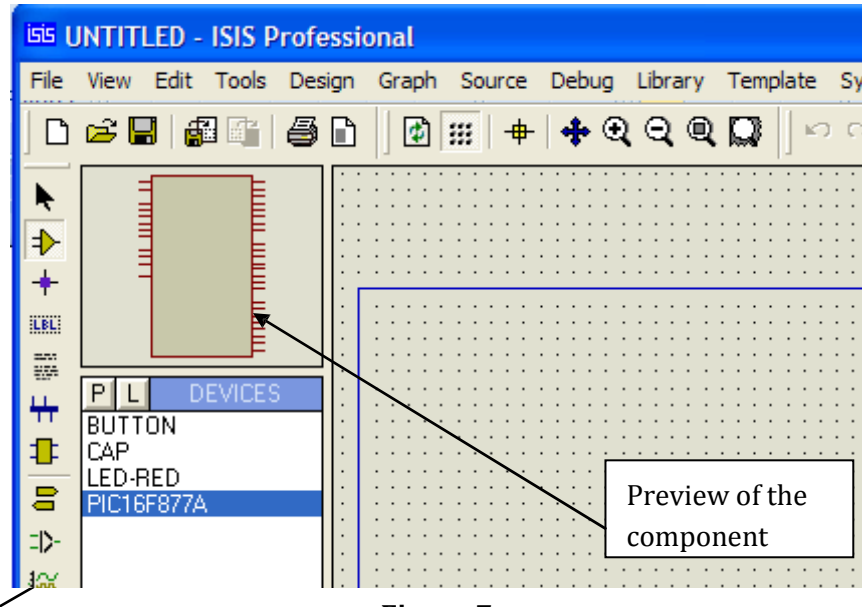


Figure 7

When you click left on any component in the Object Selector, a preview of the component will appear in the Overview Window

In order to place ground or 5 voltage right click on the Editing Window , select place then terminal then select ground (0 V) or power (5 V).

Connect the components to obtain the circuit you need.

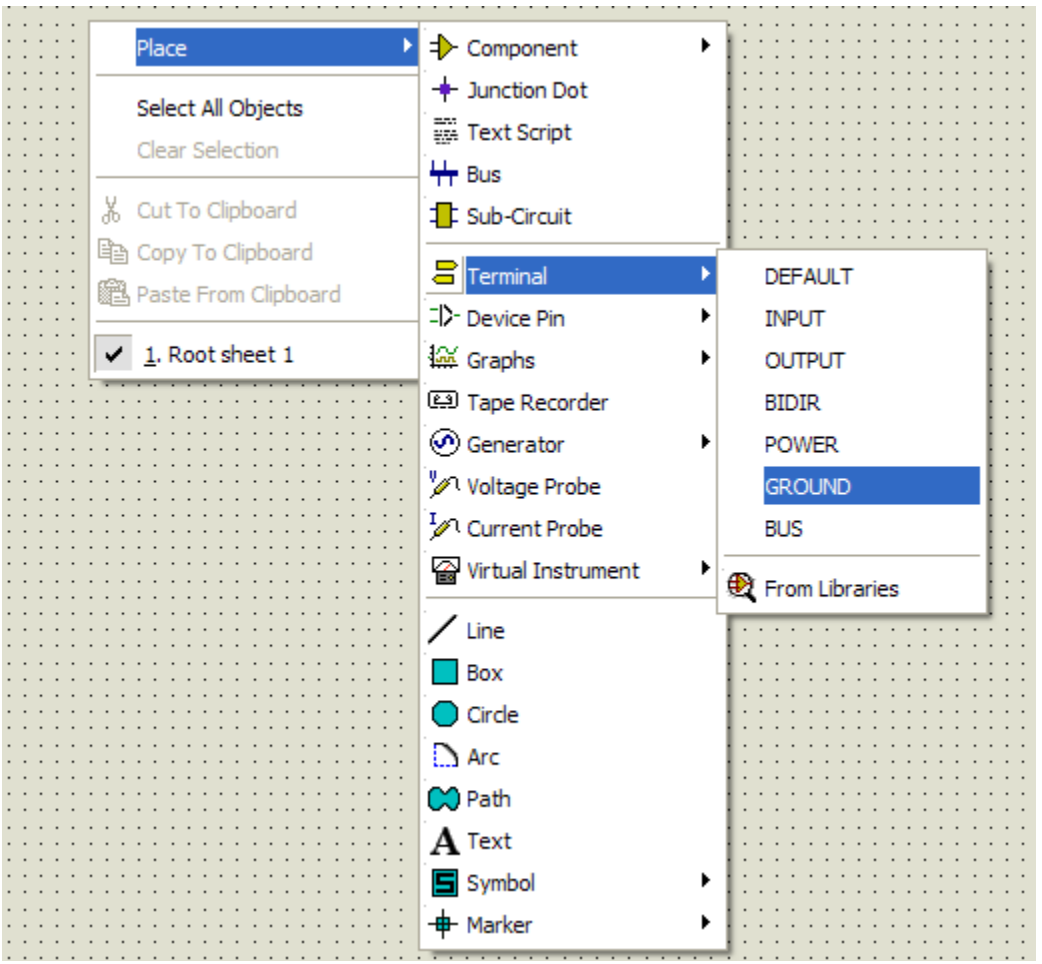


Figure 8

Attaching the HEX File

The next stage is to attach the HEX file to our design in order to successfully simulate the design. We do this through the following steps.

It is necessary to specify which file the processor is to run. In our example this will be filename.hex (the hex file produced from MPASM subsequent to assembling filename.asm).

To attach this file to the processor, right click on the schematic part for the PIC and then left click on the part. This will bring up the Edit Component dialogue form which contains a field for Program File. If it is not already specified as filename.hex either enter the path to the file manually or browse to the location of the file via the button to the right of the field. Once you have specified the hex file to be run press ok to exit the dialogue form.
We have now attached the source file to the design .

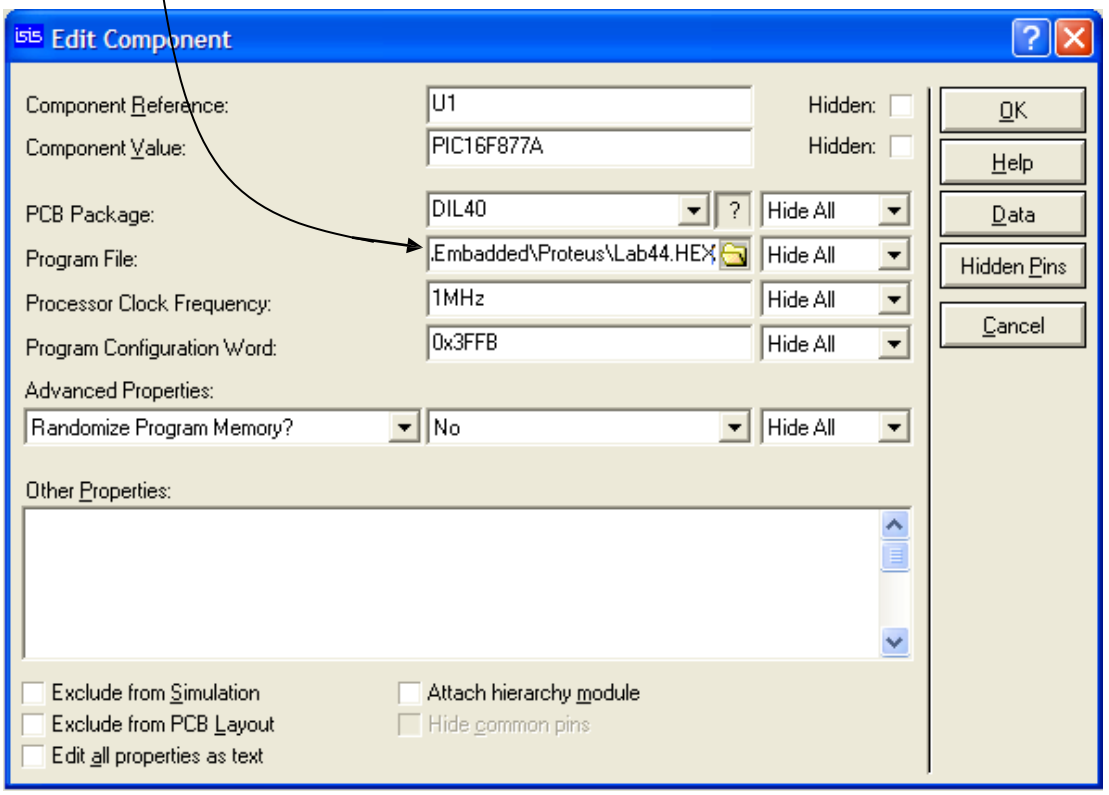


Figure 9

Debugging the Program (Simulating the Circuit)

In order to simulate the circuit point the mouse over the Play Button on the animation panel at the bottom right of the screen see Figure 10 and click left. The status bar should appear with the time that the animation has been active for.

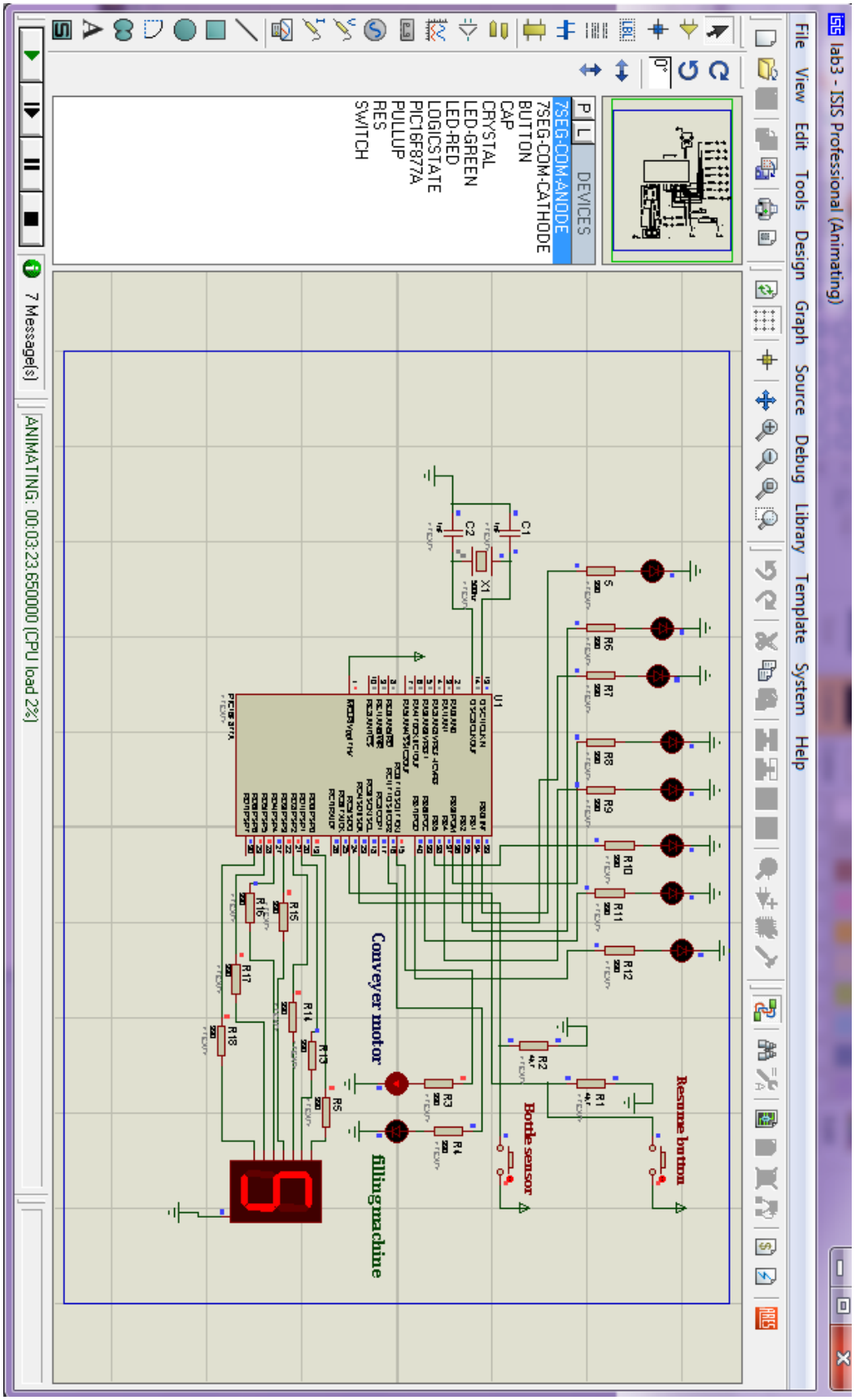


Figure 10: The Filling Machine Circuit