

MPLAB8 and Flow Charts

One of the challenges you face with any microcontroller is how to get the program onto the chip.

One way to program a PIC chip is to use an external programmer, such as PICStart-Plus



Figure 1: PICStart Plus from www.Digikey.com

Once you compile your program with MPLAB (coming next), you place your PIC chip into the programmer and select *program*. That easy. (This is how the boot-loader on your PIC chips was programmed if you're curious.)

The problem with external programmers is you have to remove your PIC chip from the board, program it, and put it back into your board. This is inconvenient, breaks pins, takes time, etc.

A boot-loader is a small program resident on your PIC chip which runs on reset. This program waits to see if you are trying to download a new program.

- If so, it takes data sent on the serial port and writes it to program memory.
- If not, it runs whatever program is in program ROM

The boot-loader on your PIC chip takes up the lower 0x300 words of program memory. On reset, it sends the message

3 2 1 0 >

to the serial port at 9600 baud.

- If it gets to zero before you hit the return button on the PC keyboard (ascii 13), it then executes whatever program is in memory, starting at address 0x800
- If you hit the return key, however (ascii 13), the boot loader clears program memory and waits to receive a new program on the serial port at 9600 baud.

Note that this means you need to start your programs at 0x800

MPLAB8 and Assembler Programming

To write a program in assembler in MPLAB8

Step 1. Create a new directory. I prefer using your Z: drive with a folder Z:\ECE376\ASM\Count

Step 2. Start MPLAB8

Step 3. Click on File New Project

Project Wizard if this is a new project

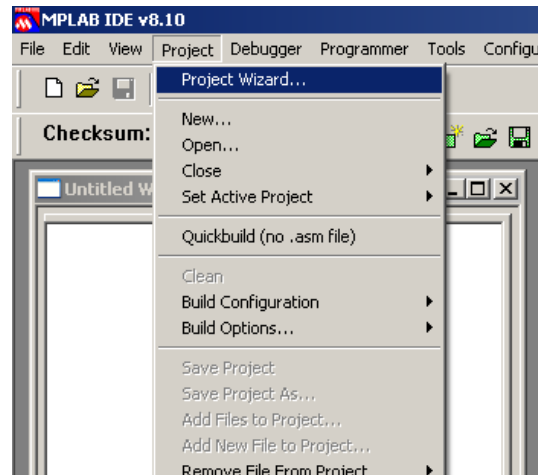


Figure 2: To start a new program, start with the Project Wizard in MPLAB8

This takes you through the process of starting a new project (i.e. a new program). Click OK

Device = PIC18F4620 (next)



Figure 3: Select a PIC18F4620 as your device

Program Language is MPASM

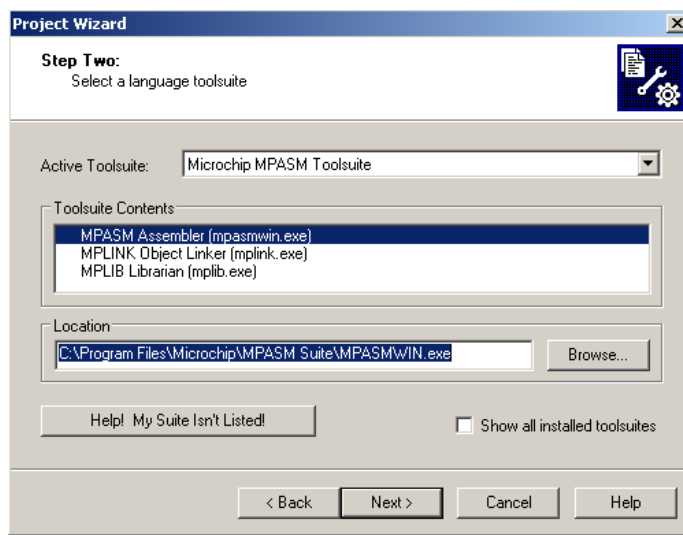


Figure 4: Select the programming language. For assembler programs, select MPASM

Directory for Files: Select the directory on your Z-drive (I don't have a Z-drive so I'm using my C drive.)

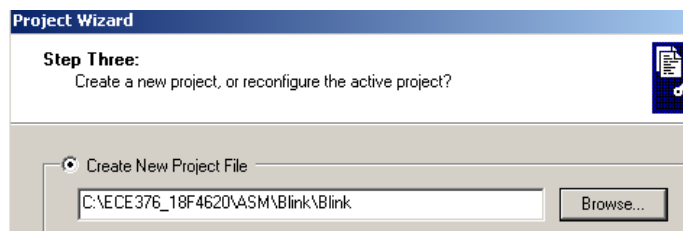


Figure 5: Select where you want your program and other files to be located

If you have an ASM file, select that file. If not, leave the file name blank and continue.

Click on View Project

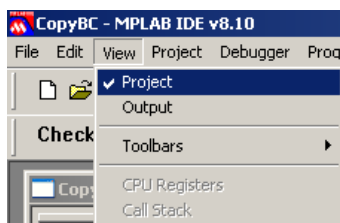


Figure 6: Once a project is created, click on View-Project to see the project

You should see the following:

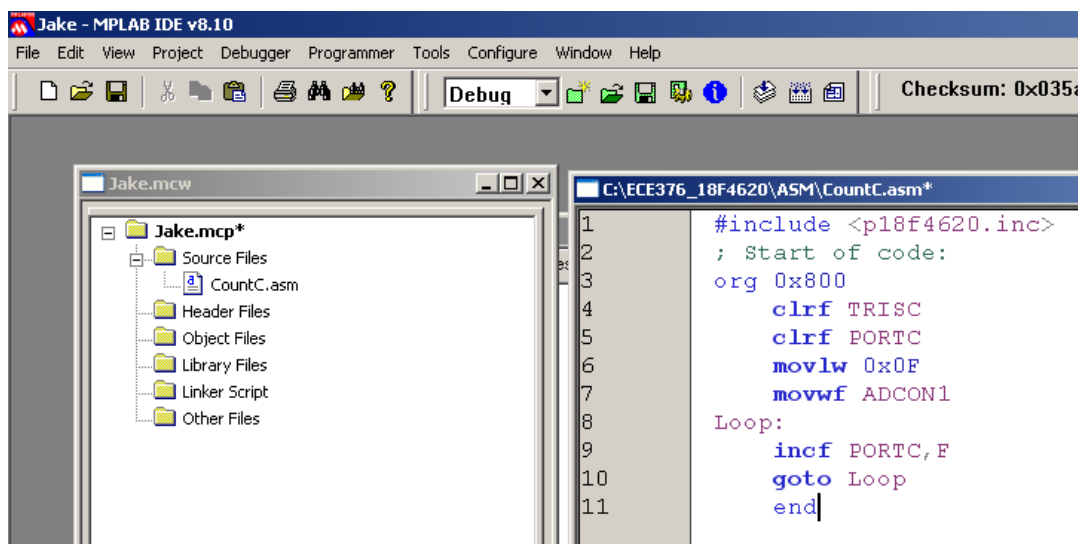


Figure 7: Under Source Files, right-click and select the file you want to compile

Change the default to decimal. Click on Project Build Options Project

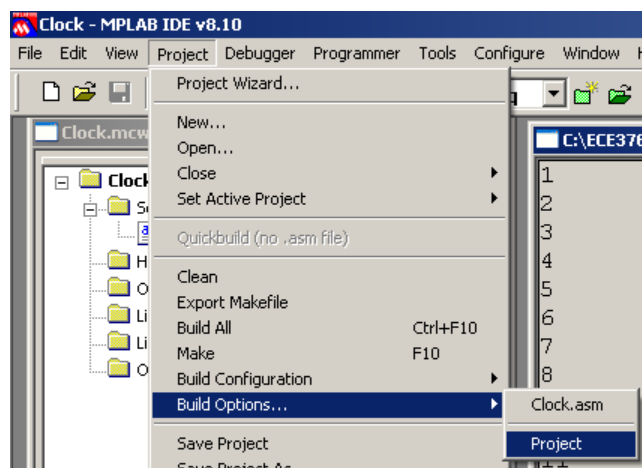


Figure 8: Change the default representation of numbers by going under Project - Build Options - Project

Click on MPASM and select Decimal. This results in numbers like 100 representing 100 base 10.

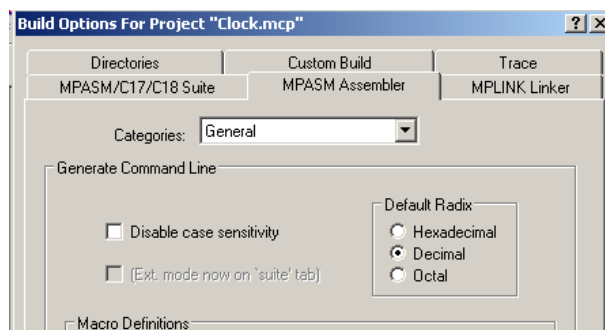


Figure 9: Now go under MPASM Assembler and change the default to Decimal

The source file is what you compile.

- If this is blank, right click on Source File and select the ASM file you wish to compile.
- If you don't have an ASM file yet, select File New edit a file, and save it as .ASM

To compile your code, click on Project Build All (or hit key F10)

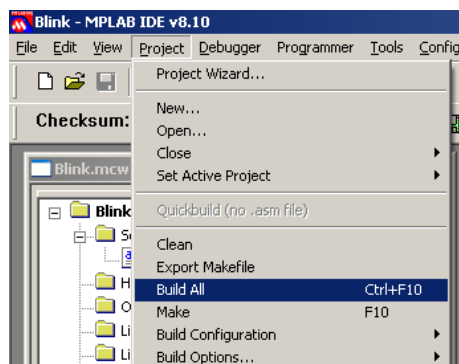


Figure 10: Once a project is created, you can compile it by clicking F10 or Project - Build All

If your program compiles correctly, you get the message 'Succeed'

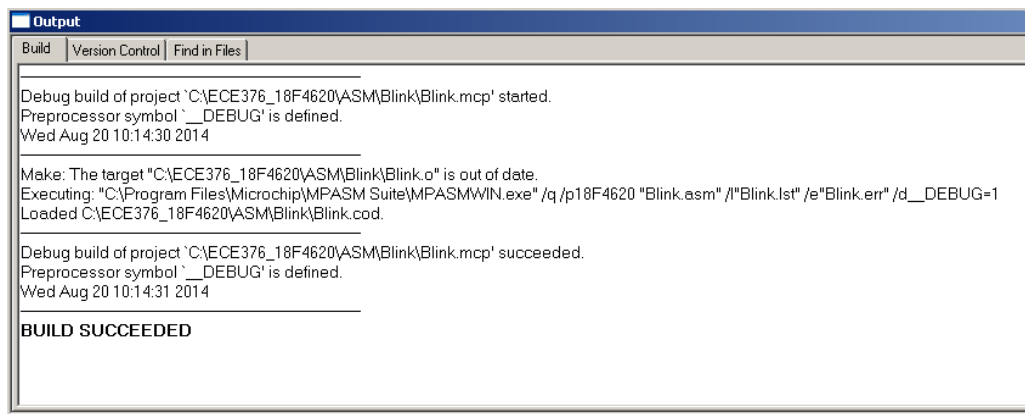


Figure 11: If all goes well, the message 'Build Succeeded' will appear

If there is an error in your code (such as a space in line 13 below), you will get an error message along with a notice which line has a problem

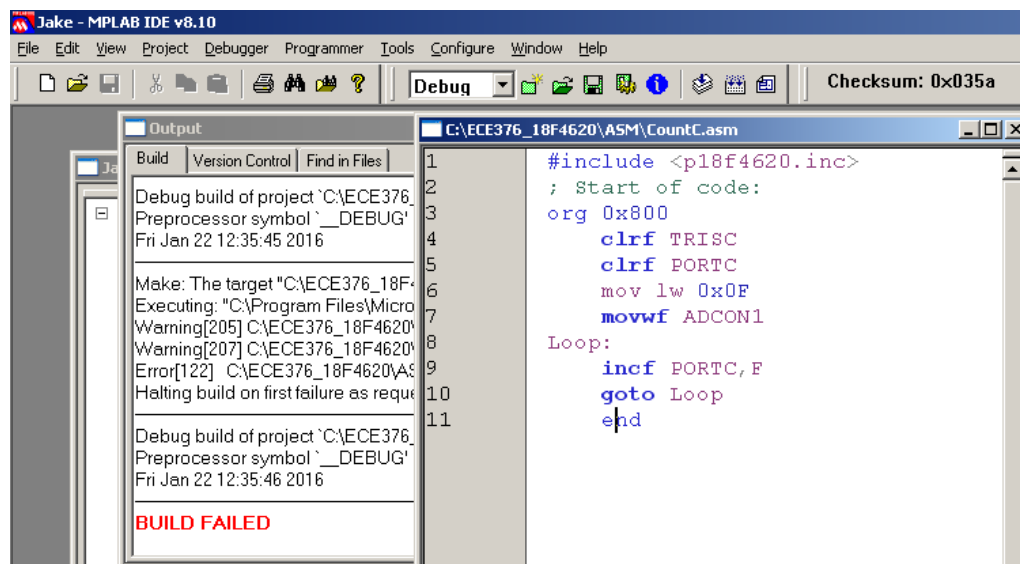


Figure 12: If there is an error in your code, Build Failed appears when you compile your program

Note that compiling doesn't mean your code is correct - it only means the compiler could understand it.

If you want to see what your program looks like, click on View Program Memory

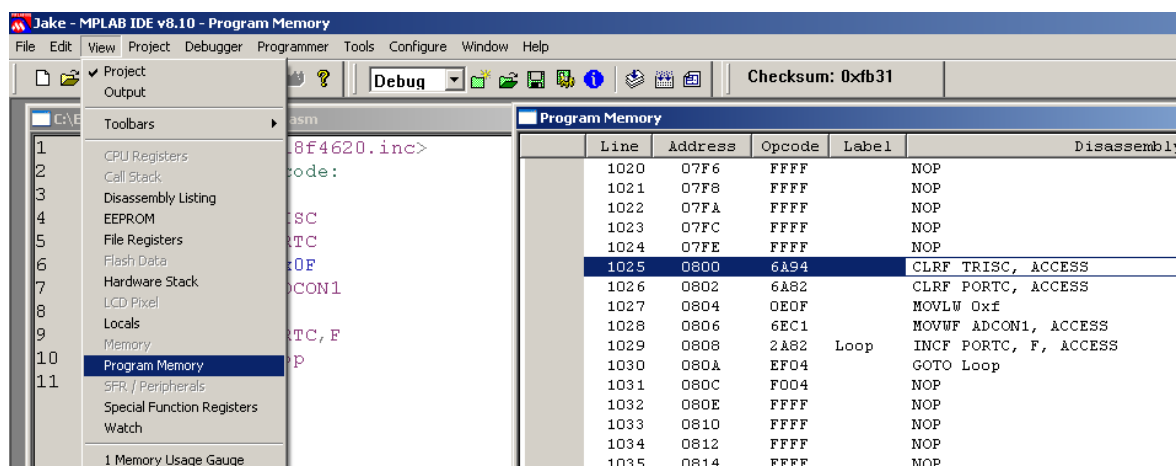


Figure 13: Once compiled, View - Program Memory shows you the resulting program in memory

Note that your program starts at address 0x800 (due to the ORG statement. This keeps it away from the boot-loader). Also note that this is a very small program: it takes 8 lines of code (out of 32,000). A PIC can do more.

The program is stored in the file .HEX. This is a text file that contains the program in machine language (the OP-Code above)

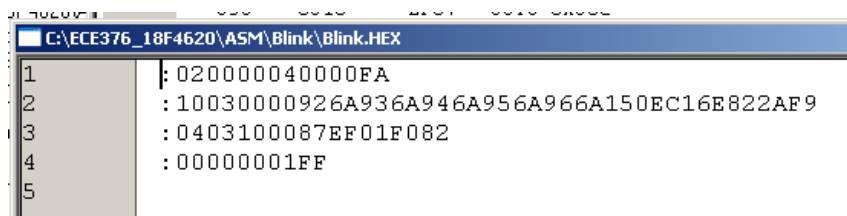


Figure 14: Once compiled, a dot-hex file is created. This contains the actual program in machine code

To download your code to your PIC board,

- Power up your PIC board (i.e. plug it in)
- Connect the serial cable to a PC
- Run a terminal program, such as Hyperterminal or PIC_Flash_Tool (you might have to download the PIC Flash Tool. It's on the ECE 376 Resources page)

<http://www.bisonacademy.com/ECE376/Resources.htm>

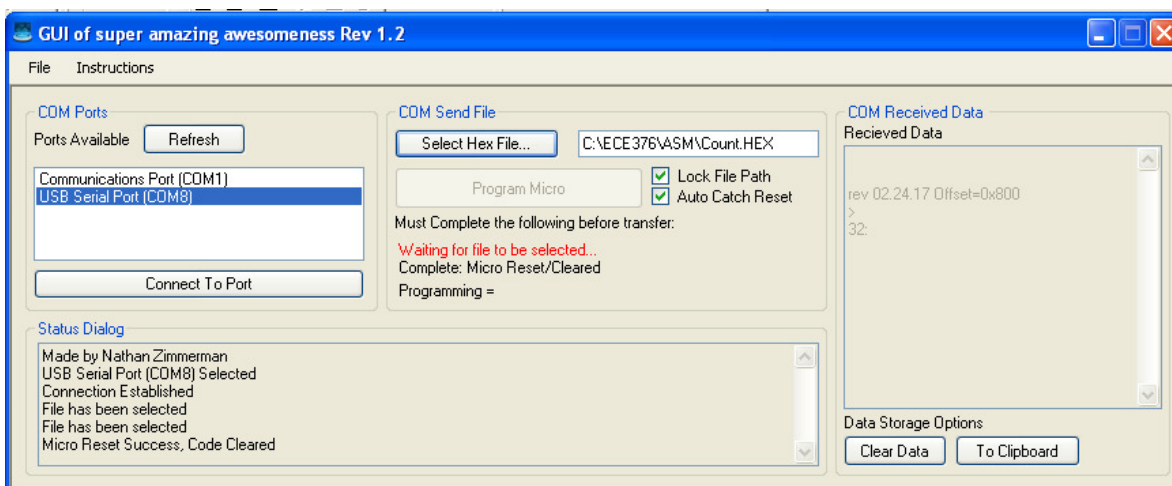


Figure 15: To download your hex file, run the program PIC-Flash-Tool.exe

- Select the USB Serial Port (COM number varies)
- Select the HEX file to download.
 - Note: PIC_Flash only recognizes capital .HEX. You might have to rename your file if the compiler used lower case letters
- Hit RESET on your PIC board. This should result in a 321 message on the COM Received Data window.
- When the count gets to 2, this program should send a carriage return - which clears out the old program and waits for a new one. At that point, Program Micro lights up.
- Press Program Micro. You will see the LED on RA4 blink a few times (each blink is one line in the .HEX file) then your program is running.

Flow Charts and Assembler Programs

Flow Charts:

A flow chart is a graphical way to display how a program works (i.e. the algorithm). The purpose of a flow chart is to make the program easier to understand. Likewise, when you make a flow chart, try to

- Keep it simple (less than 20 blocks), but
- Keep it informative (more than one block)

It also helps if you follow a few rules:

- Flow charts should start at the top of the page
- The program execution should move down towards the bottom of the page
- There should be a single exit point

The main symbols for flow charts are as follows:

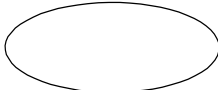
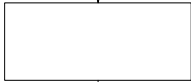
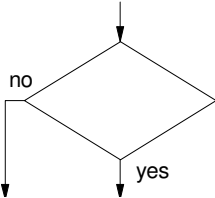
Symbol	Image	Meaning
Oval		Start / End of routine or program
Rectangle		Function or operation
Parallelogram		Input / Output
Diamond		Decision

Table 1: Select flow-chart symbols and meaning

Ideally, your program should match up with the flow chart. Sort of like how your English term paper should match up with the outline.

For complicated routines, it helps to write the flow chart first. For smaller programs, you can just write the program first then draw the corresponding flow chart. This again is sort of like how you write your English papers and the corresponding outline.

PIC I/O

To illustrate some programs, it will help to use some of the I/O pins on the PIC. These are pins which allow you to detect when a button is pressed (input) or turn an LED on and off (output). We'll talk more about this later - but for now, understand that the PIC has five I/O ports: PORTA..PORTE. These are physically connected to the pins on your PIC chip as follows:

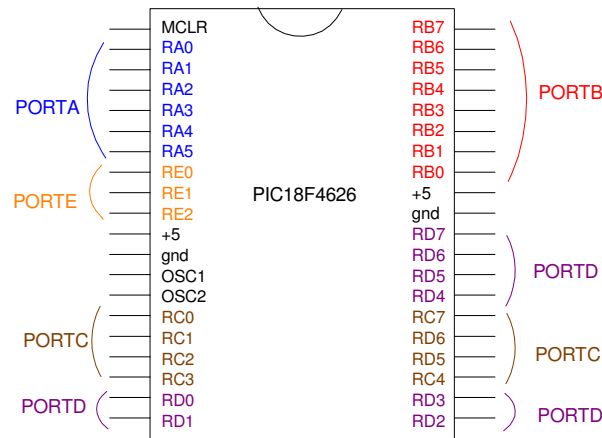


Figure 16: Pin assignments for a PIC18F4620 microcontroller used in class

The PIC18f4620 chip has 33 I/O lines split into five ports:

	PORTA	PORTB	PORTC	PORTD	PORTE
Pins	2..7	33..40	15..18, 24..26	19..22, 27..30	3
Binary Input	5	8	8	8	3
Binary Output	5	8	8	8	3
Analog Input	5	5	-	-	3

Table 2: Pin assignments and I/O options for ports A through E

Setting Up I/O Ports for Binary I/O

Three registers are associated with each port

- PORTx: Defines whether the pin is 0V (0) or 5V (1)
- TRISx: Defines whether the pin is input (1) or output (0)
- LATx: I don't understand what the latch does. The data sheets say "Read-modify-write operations on the LATC register read and write the latched output value for PORTC." To this, I say "huh?" So far, ignoring the LAT registers hasn't caused any problems for me. They're probably good for something though.

In addition, you need to initialize ADCON1 to 15

```
movlw    15          load the number 15 to W
movwf    ADCON1      write W to ADCON1
```

This sets all I/O pins to binary. Some can be analog inputs as well - we'll cover this later when we get to A/D converters.

TRISx are 8-bit registers. Each bit defines whether a given pin is input (1) or output (0). You can write to all 8 bits at once or set and clear each bit one at a time. For example, the command

```
movlw    0x0F          binary 0000 1111
movwf    TRISB
```

sets RB0..3 to input (1) and RB4..7 to output (0). The commands

```
bsf      TRISC, 1
bcf      TRISD, 2
```

sets PORTC pin 1 (making RC1 input) and clears PORTD pin 2 (making RD2 output)

PORTx defines the value on each pin:

- logic 0 is 0V
- logic 1 is 5V.

When a pin is input, the logic level is defined by the external voltage applied to the pin. Writing to an input has no affect.

When a pin is output, the logic level is defined by your program. Writing a 1 outputs 5V, writing a 0 outputs 0V.

note: Each I/O pin can source or sink up to 25mA. If you want to tie an I/O pin to 0V or 5V, you should use a 200+ Ohm resistor rather than a wire. (Your evaluation boards use 1k resistors). If you accidentally set that I/O pin to an output opposite of the connection, the 1k resistor limits the current to 5mA, saving the PIC. If you set the pin to input, the current should be zero (inputs have high impedance) and the 1k resistor has no effect.

Example 1: Write a program which counts how many times you press RB0. Display this count on PORTC.

Software: To count one time each button press, you need to

- Keep checking RB0 until it goes high, then
- Keep checking RB0 until it goes low, and
- Repeat

If all you do is wait for RB0 to go high, you'll count really fast while RB0 is high rather than just once.

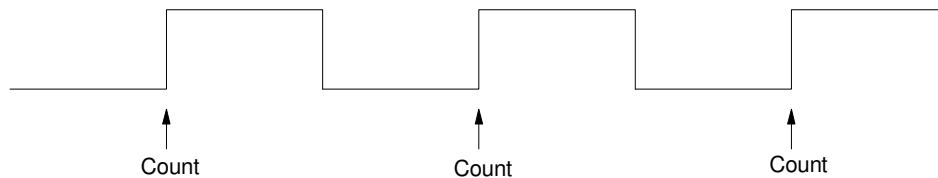


Figure 17: The assembler program should count every rising edge

```
#include <p18f4620.inc>

; --- COUNT_RB0.ASM ----
; This program counts how many times
; RB0 is pressed and displays the result
; on PORTC

; Program

    org 0x800

    clrf TRISA
    movlw 0xFF
    movwf TRISB
    clrf TRISC
    clrf TRISD
    clrf TRISE
    movlw 0x0F
    movwf ADCON1
    clrf PORTC

Loop1:
    btfsc PORTB,0
    goto Loop1
Loop2:
    btfss PORTB,0
    goto Loop2
    incf PORTC,F
    goto Loop1
end
```

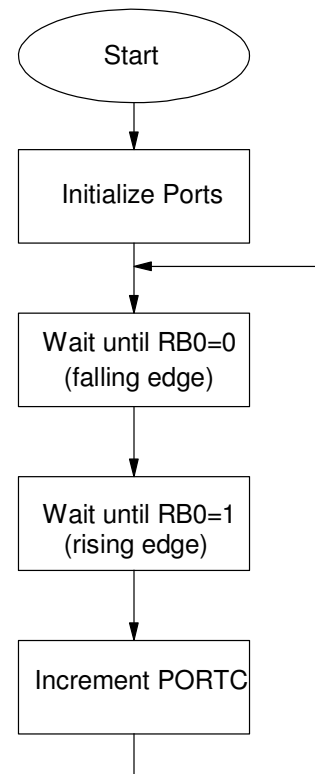


Figure 18: Flow chart for a counting program and its corresponding assembler code

Example 2: Design a circuit which displays a random number from 0..7 every time RB0 is pressed and released.

Solution: One way to do this is to use the above program but count really fast while RB0 = 1. When you release, the resulting value will look like a random number.

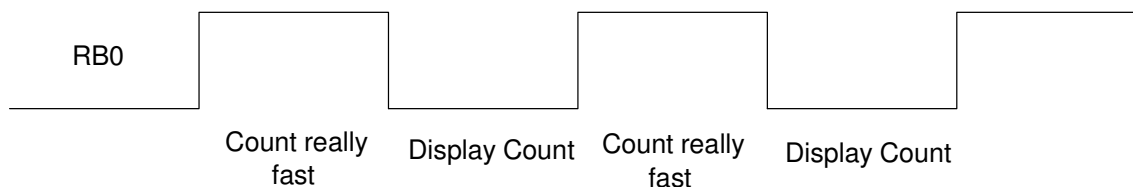


Figure 19: One way to generate a random number is to count really fast while a button is held down

```

; --- RANDOM.ASM ---
; This program generates a random number 0..7 every time RB0 is pressed
; and sends the result to PORTC

#include <p18f4620.inc>

; Variables
DIE EQU 0

; Program

org 0x800

clrf TRISA
movlw 0xFF
movwf TRISB
clrf TRISC
clrf TRISD
clrf TRISE
movlw 0x0F
movwf ADCON1

Main:
btfsc PORTB,0
incf DIE,W
andlw 0x07
movwf DIE
movwf PORTC
goto Main

end

```

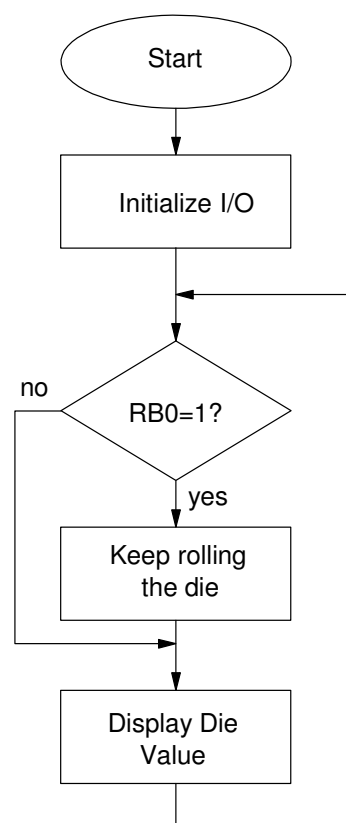


Figure 20: Assembler code and flow chart for a random number generator

Top Down Programming:

A better way to write this program uses subroutines and top-down programming. The idea is to identify the main functions you need to do to make the program work and make each of these a subroutine. You then fill in each subroutine one by one to get the program to work.

For example, the previous program could be

```

; --- RANDOM.ASM ---
; This program generates a random number 0..7 every time RB0 is pressed
; and sends the result to PORTC

#include <p18f4620.inc>

; Variables
DIE EQU 0 ; random number located at address 0

; --- Main Routine ---

    org 0x800
    call Init
Main:
    btfsc PORTB,0
    call Roll
    call Display
    goto Main

; --- Subroutines ---

Init:
    clrf TRISA
    movlw 0xFF
    movwf TRISB
    clrf TRISC
    clrf TRISD
    clrf TRISE
    movlw 0x0F
    movwf ADCON1
    return

Roll:
    incf DIE,W
    andlw 0x07
    movwf DIE
    return

Display:
    movf DIE,W
    movwf PORTC
    return

end

```

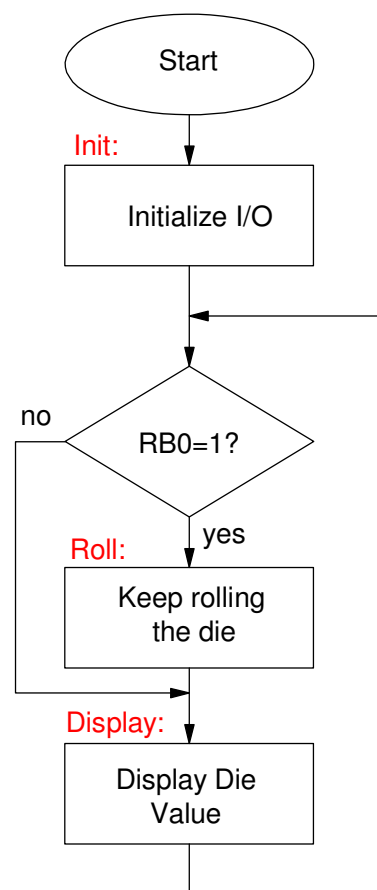


Figure 21: Assembler code and corresponding flow chart for rolling a 6-sided die

Note that, with top-down programming,

- The main routine is much easier to understand
- Modifications can be made by changing one subroutine

