Flow Charts and Assembler Programs

Flow Charts:

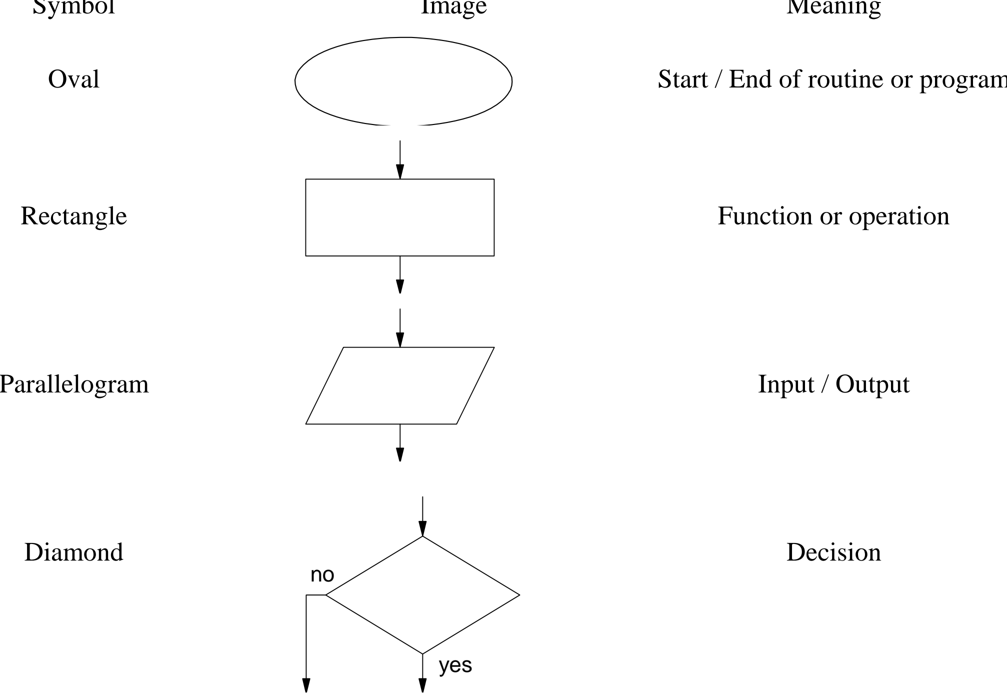
A flow chart is a graphical way to display how a program works (i.e. the algorithm). The purpose of a flow chart is to make the program easier to understand. Likewise, when you make a flow chart, try to

- 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:

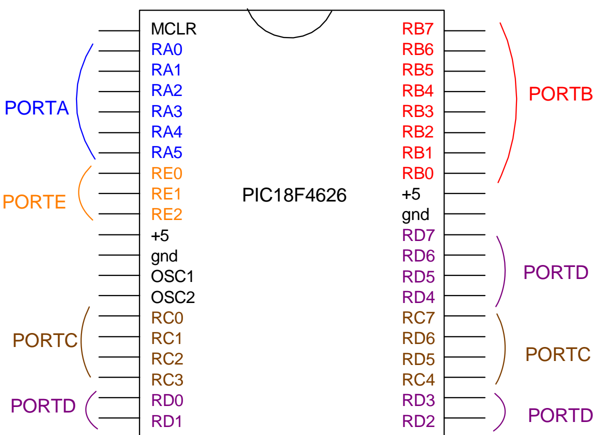


Ideally, your program should match up with the flow chart. Sort of like how your English term paper should match up with the outline.

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:



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

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.

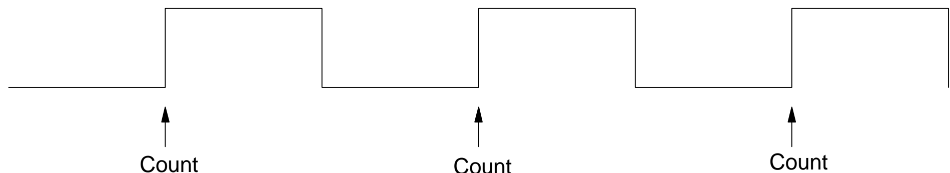
note: Each I/O pin can source or sink up to 25mA. If you want to tie an I/O pin to 0V or 5V, you should use a 200+ Ohm resistor rather than a wire. (Your evaluation boards use 1k resistors). If you accidently set that I/O pin to an output opposite of the connection, the 1k resistor limits the current 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.

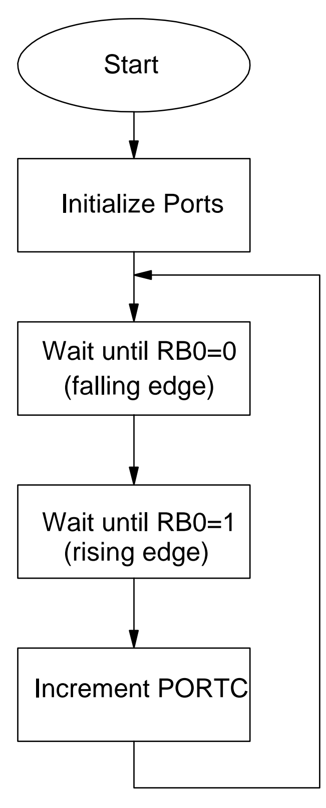
Software: To count one time each button press, you need to

- 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.




```
 #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         ;PORTA is output
    movlw  0xFF
    movwf  TRISB         ;PORTB is input
    clrf   TRISC         ;PORTC is output
    clrf   TRISD         ;PORTD is output
    clrf   TRISE         ;PORTE is output
    movlw  15
    movwf  ADCON1        ;everyone is binary
    clrf   PORTC         ; start from zero


Loop1:                  ; Start of main loop
        btfsc PORTB,0   ; Wait until RB0=0
        goto  Loop1
Loop2:
        btfss PORTB,0   ; Wait until RB0=1
        goto  Loop2

        incf  PORTC,F   ; Increment count

        goto  Loop1

        end
```



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

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.

RB0

Count really fast | Display Count | Count really fast | Display Count

```
; ---  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

; Program

        org 0x800

        clrf   TRISA    ;PORTA is output
        movlw 0xFF
        movwf TRISB     ;PORTB is input
        clrf   TRISC    ;PORTC is output
        clrf   TRISD    ;PORTD is output
        clrf   TRISE    ;PORTE is output
        movlw 15
        movwf ADCON1    ;everyone is binary

Main:
        btfsc PORTB,0  ; if RB0=1

        incf    DIE,W   ; add one to the die roll
        andlw   0x07    ; take the result mod 8
        movwf   DIE     ; and save it

        movwf   PORTC   ; display the die value

        goto    Main    ; and repeat
```

Start

Initialize I/O

RB0=1?

no

yes

Keep rolling the die

Display Die Value

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  ; if RB0=1
        call  Roll     ; keep rolling the die

        call  Display  ; display the die value

        goto  Main     ; and repeat

; ---  Subroutines ---

Init:
        clrf  TRISA    ;PORTA is output
        movlw 0xFF
        movwf TRISB    ;PORTB is input
        clrf  TRISC    ;PORTC is output
        clrf  TRISD    ;PORTD is output
        clrf  TRISE    ;PORTE is output
        movlw 15
        movwf ADCON1   ;everyone is binary
        return

Roll:
        incf   DIE,W   ; add one to the die roll
        andlw  0x07    ; take the result mod 8
        movwf  DIE     ; and save it
        return

Display:
        movf   DIE,W   ; display die on PORTC
        movwf  PORTC
        return

        end
```


Note that, with top-down programming,

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

Example 3: Design a circuit which displays a random number from 1..6 every time RB0 is pressed and relased.

Solution: Change the subroutine *Roll:* to count from 1 to 6.

- Increment DIE by one
- If DIE == 7, reset it to 1

```
Roll:
        incf   DIE,F   ; add one to the die roll
        movlw  7
        cpfseq DIE     ; check if DIE = 7
        return         ; if DIE < 7, return
        movlw  1       ; if DIE=7, make DIE=1
        movwf  DIE
        return
```