Chapter 2. PIC 16F877 Microcontroller - Overview

1. PIC 16F877 Architecture

PIC 16F877 is a 40-pin 8-Bit CMOS FLASH Microcontroller from Microchip. The core architecture is high-performance RISC CPU with only 35 single word[1] instructions. Since it follows the RISC architecture, all single cycle instructions take only one instruction cycle except for program branches which take two cycles. 16F877 comes with 3 operating speeds with 4, 8, or 20 MHz clock input. Since each instruction cycle takes four operating clock cycles, each instruction takes 0.2 μs when 20MHz oscillator is used.

It has two types of internal memories: program memory and data memory. Program memory is provided by 8K words (or 8K*14 bits) of FLASH Memory, and data memory has two sources. One type of data memory is a 368-byte RAM (random access memory) and the other is 256-byte EEPROM (Electrically erasable programmable ROM).

The core feature includes interrupt capability up to 14 sources, power saving SLEEP mode, and single 5V In-Circuit Serial Programming (ICSP) capability. The sink/source current, which indicates a driving power from I/O port, is high with 25mA. Power consumption is less than 2 mA in 5V operating condition.

The peripheral features include:

(a) 3 time blocks: Timer0 for 8-bit timer/counter; Timer1 for 16-bit timer/counter; and Timer2: 8-bit timer/counter with 8-bit period register, prescaler and postscaler.
(b) Two Capture, Compare, PWM modules for capturing, comparing 16-bit, and PWM generation with 10-bit resolution.
(c) 10-bit multi-channel (max 8)Analog-to-Digital converter module.
(d) Synchronous Serial Port (SSP) with SPI (Master Mode) and I^{2}C[2] (Master/Slave)
(e) Universal Synchronous Asynchronous Receiver Transmitter (USART/SCI) with 9-bit address detection
(f) Parallel Slave Port (PSP) 8-bits wide, with external RD, WR and CS controls
(g) I/O ports.

The key feature of 16F877 is summarized below:

FLASH Program Memory (14-bit word)	8K Words
Data Memory (RAM)	368 Bytes
Data Memory (EEPROM)	256 Bytes
Interrupts	14
I/O Ports	Ports A, B, C, D, E
Timers	3
Capture/Compare/PWM Modules	2
Serial Communications	MSSP, USART

[1] The 'word' here is not the usual term we use to indicate 2 bytes (or 16 bits). The 'word' in PIC could be 12, 14, or any number of bits. For 16F877, the word means a size of 14 bits.
[2] I^{2}C stands for Inter-IC bus.

Parallel Communications	PSP
10-bit Analog-to-Digital Module	8 channels
Instruction Set	35 Instructions

2. Pin and Package

There are three package types are available: DIP, PLCC, and QFP. This book assumes that we all use the DIP because of its best fit to breadboard or proto-board.

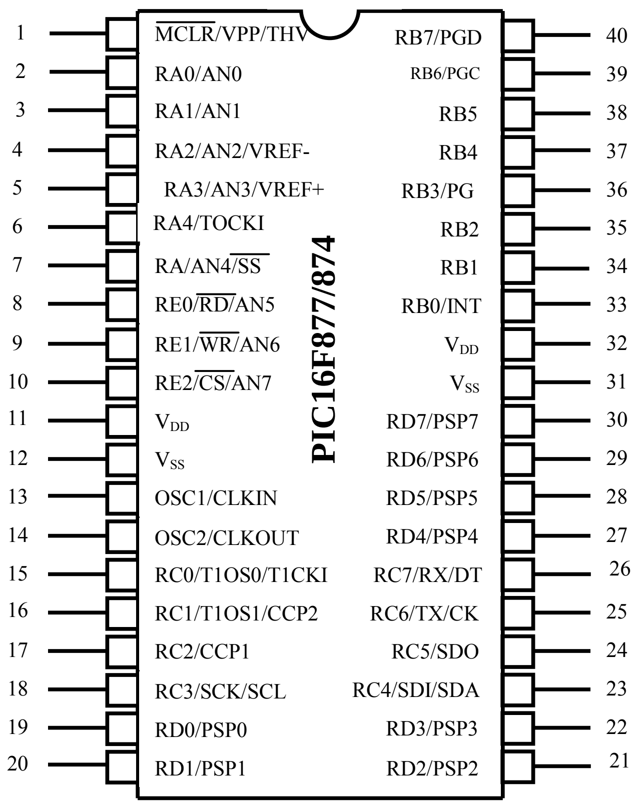


Fig 1. PIC 16F877 IC Package: DIP

Table 1. Pin Description

Pin #	Pin Name	Description		Pin #	Pin Name	Description
1	~MCLR			21	RD2	PORTD.2
2	RA0/AN0	PORTA.0 /Analog Channel 0		22	RD3	PORTD.3
3	RA1/AN1	PORTA.1 /Analog Channel 1		23	RC4/SDI/SDA	PORTC.4/SDI(for SPI)/SDA(for I^2C)
4	RA2/AN2	PORTA.2 /Analog Channel 2		24	RC5/SDO	PORTC.5 /SDO (for SPI)
5	RA3/AN3	PORTA.3 /Analog Channel 3		25	RC6/TX	PORTC.6 /TX (for Serial Com.)
6	RA4/T0CK1	PORTA.4 / External Clock for Timer 0		26	RC7/RX	PORTC.7 /RX (for Serial Com.)
7	RA5/AN4	PORTA.5 /Analog Channel 4		27	RD4	PORTD.4
8	RE0/AN5	PORTE.0 /Analog Channel 5		28	RD5	PORTD.5
9	RE1/AN6	PORTE.1 /Analog Channel 6		29	RD6	PORTD.6
10	RE2/AN7	PORTE.2 /Analog Channel 7		30	RD7	PORTD.7
11	Vdd	+3 ~ +5V		31	Vss	GND
12	Vss	GND		32	Vdd	+3 V ~ +5 V
13	OSC1/CLKIN	Oscillator Connection /Clock In		33	RB0/INT	PORTB.0/External Interrupt
14	OSC2/CLKOUT	Oscillator Connection / Clock Out		34	RB1	PORTB.1
15	RC0/T1CK1	PORTC.0 /External Clock for Timer 1		35	RB2/PGM	PORTB.2 /Programming Input
16	RC1/CCP2	PORTC.1 /CCP2		36	RB3	PORTB.3
17	RC2/CCP1	PORTC.2 /CCP1		37	RB4	PORTB.4
18	RC3/SCK/SCL	PORTC.3 /SCK(for SPI)/SCL(for I^2C)		38	RB5	PORTB.5
19	RD0	PORTD.0		39	RB6/PGC	PORTB.6 /Debugger/ICSP
20	RD1	PORTD.1		40	RB7/PGD	PORTB.7 /Debugger/ICSP

3. Block Diagram

The internal schematic of PIC16F877 is illustrated in the following block diagram. In the block diagram, we first see built-in function modules in boxes such as PORTA, PORTB for inputs and outputs, a 10-bit Analog-to-Digital (AD) module, 3 timer modules (Timer 0, Timer 1, and Timer 2), EEPROM, etc. We also see the modules are connected via internal data bus of 8-bit width. This is the reason why 16F877 is called an 8-bit microcontroller. Since the program memory (FLASH) is organized with 14-bit word, it's content is accessed (or fetched) via 14-bit program bus by the instruction register. Instruction registers are not in the file (RAM) area so is not accessible by program. Another internal register, program counter (PC) accesses the program memory location ("addressing") via 13 bit information. With 13 bits, the addressing range goes to $2^{13}=2^3*2^{10}$=8K word.

RAM file registers for data are connected via 8-bit data bus. The **W** register is a user accessible register. As you see in the block diagram, **W** directly accesses the ALU and the 8-bit data bus. As we can see, there should be **W** involved to move data between modules and inside the file RAM, except via indirect addressing mode. This means that writing to the output ports is done via the **W** register. The details of the indirect addressing are discussed in the PIC instruction.

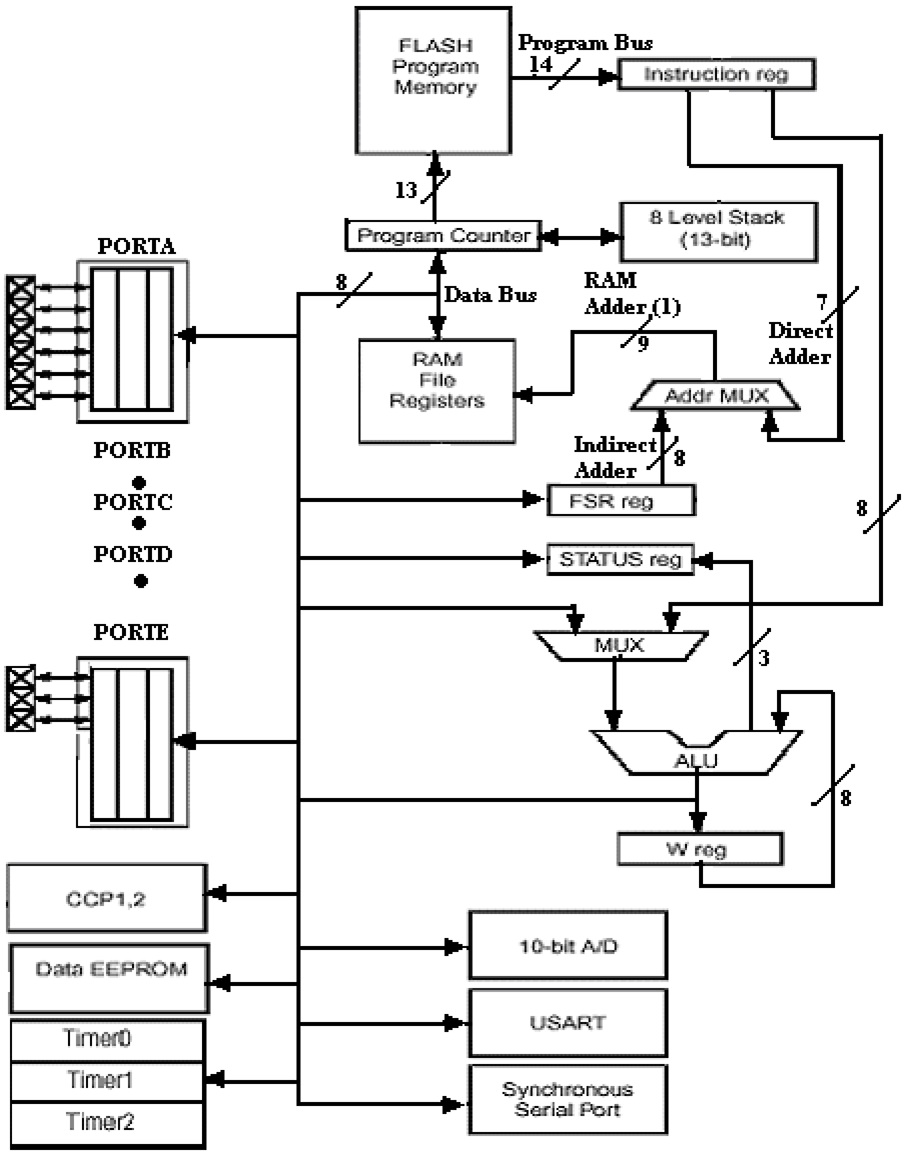


Fig. 2 PIC16F877 Block Diagram

4. Program Memory

The PIC16F877 has a 13-bit program counter (PC) capable of addressing an 8K word of program memory pace, and accessing a location above the physically implemented address will cause a wrap-around.

The reset vector is at address 0000 and the interrupt vector is at address 0004[3]. Details of this will be discussed through out the book as related subjects pop up.

[3] All addresses are in hexadecimal number unless stated otherwise. Hexadecimal numbers are usually indicated by a multiple-digit number followed by a letter **h**. In assembly language programming, however, we indicate a hexadecimal number with preceding 0x followed by a multi-digit number. For example, **1001h** and **0x1001** both indicate a 4 digit hexadecimal number 1001.

5. Data Memory (RAM)

The data memory in RAM is partitioned into multiple banks (or pages) which contain the General Purpose Registers (GPR) and the Special Function Registers (SFR). The SFRs are already internally defined by the system so users are not allowed to override these registers. These are the main tools to utilize the built-in functions. In other words, assigning a port (PORTA, PORTB, etc) or a pin of a port as input or output, for example, is done using one or more of the SFRs. The GPRs are actually empty spaces to be used to store data or variables. In other words, when you define a variable like, `x`, `y`, or `temp`, then you have to allocate a space (a byte size) for each in the GPRs in RAM area. The registers in the RAM area are called *file registers*.

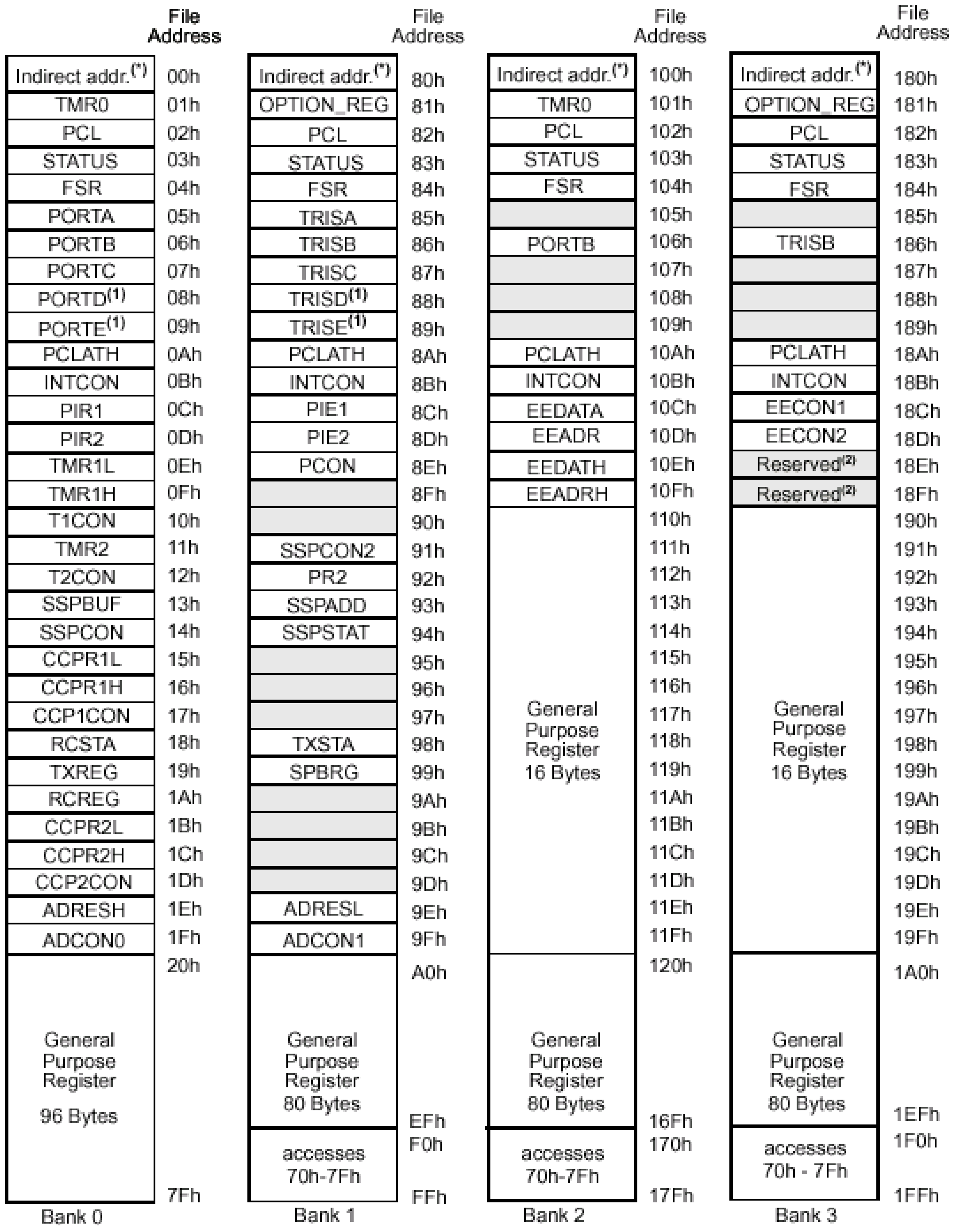
Each bank extends to 7Fh (128 bytes). The lower locations of each bank are reserved for the SFRs. Above the SFRs are GPRs, implemented as static RAM. All implemented banks contain SFRs. Some "high use" SFRs, such as STATUS register, PCL register, FSR register, and INTCON register from one bank may be mirrored in another bank for code reduction and quicker access. For example, the STATUS register presents at 0003h (bank 0), 0083h(bank 1), 0103h(bank 2), and 0183h(bank 3), simultaneously.

To access registers properly, the bank location of them must be selected. In other words, you have keep track which bank of registers the CPU currently accesses. If the current bank access is Bank0, and your next register is located in Bank1, you have to select the Bank1 before accessing the register. The PIC CPU is very strict on the bank location. If we use an analogy of word processing, when we want to select a word, we have to move the cursor around the word then we click the mouse to select: without moving the cursor to the word, you do select the word you intend to do, it would select any word near the current cursor position. The selection of a bank is done by setting two bits of the STATUS register.

Let's examine the file register map briefly. As mentioned above, the fourth register (address 03h, 83h, 103h, and 183h) at each bank is assigned to STATUS register (see the Register file map). The STATUS register tells you about the status arithmetic and logical status. It also has very important function of selecting a bank.

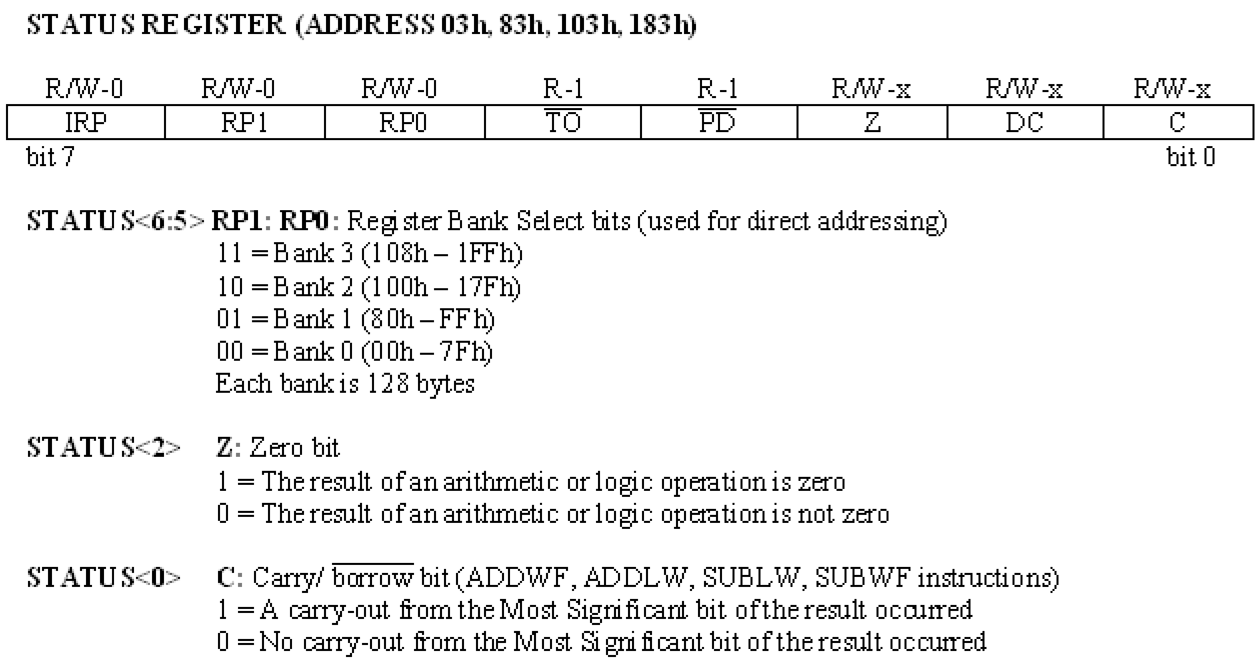
The detail of the STATUS register is shown below. Here, we highlight only bits of 6, 5, 2, and 0 of the register. Note that the bit number of the byte of the register starts from 0 ending at 7. As we see, bits 5 and 6 of the register, as we indicate them as STATUS<6:5>, are RP1 and RP0 bits which select banks. In other words, for example, if you access a register in bank 3, you have to set the two bits as `11`, or STATUS<6:5>=11b[4]. In this selection, we change each bit of a register. In such case, we may better use bit-oriented instruction (Details on instruction come later). STATUS<2> is for Z flag to see if an operation results in zero or not.

[4] a letter following a number indicates that the number is a binary number. In assembly language programming, we use letter B followed by a single quote, a binary number, then a single quote to indicate a binary number. For example, B'01101100' represents in a code a binary number 01101100b.



	File Address		File Address		File Address		File Address
Indirect addr.(*)	00h	Indirect addr.(*)	80h	Indirect addr.(*)	100h	Indirect addr.(*)	180h
TMR0	01h	OPTION_REG	81h	TMR0	101h	OPTION_REG	181h
PCL	02h	PCL	82h	PCL	102h	PCL	182h
STATUS	03h	STATUS	83h	STATUS	103h	STATUS	183h
FSR	04h	FSR	84h	FSR	104h	FSR	184h
PORTA	05h	TRISA	85h		105h		185h
PORTB	06h	TRISB	86h	PORTB	106h	TRISB	186h
PORTC	07h	TRISC	87h		107h		187h
PORTD(1)	08h	TRISD(1)	88h		108h		188h
PORTE(1)	09h	TRISE(1)	89h		109h		189h
PCLATH	0Ah	PCLATH	8Ah	PCLATH	10Ah	PCLATH	18Ah
INTCON	0Bh	INTCON	8Bh	INTCON	10Bh	INTCON	18Bh
PIR1	0Ch	PIE1	8Ch	EEDATA	10Ch	EECON1	18Ch
PIR2	0Dh	PIE2	8Dh	EEADR	10Dh	EECON2	18Dh
TMR1L	0Eh	PCON	8Eh	EEDATH	10Eh	Reserved(2)	18Eh
TMR1H	0Fh		8Fh	EEADRH	10Fh	Reserved(2)	18Fh
T1CON	10h		90h	General Purpose Register 16 Bytes	110h	General Purpose Register 16 Bytes	190h
TMR2	11h	SSPCON2	91h		111h		191h
T2CON	12h	PR2	92h		112h		192h
SSPBUF	13h	SSPADD	93h		113h		193h
SSPCON	14h	SSPSTAT	94h		114h		194h
CCPR1L	15h		95h		115h		195h
CCPR1H	16h		96h		116h		196h
CCP1CON	17h		97h		117h		197h
RCSTA	18h	TXSTA	98h		118h		198h
TXREG	19h	SPBRG	99h		119h		199h
RCREG	1Ah		9Ah		11Ah		19Ah
CCPR2L	1Bh		9Bh		11Bh		19Bh
CCPR2H	1Ch		9Ch		11Ch		19Ch
CCP2CON	1Dh		9Dh		11Dh		19Dh
ADRESH	1Eh	ADRESL	9Eh		11Eh		19Eh
ADCON0	1Fh	ADCON1	9Fh		11Fh		19Fh
General Purpose Register 96 Bytes	20h	General Purpose Register 80 Bytes	A0h	General Purpose Register 80 Bytes	120h	General Purpose Register 80 Bytes	1A0h
			EFh		16Fh		1EFh
		accesses 70h-7Fh	F0h	accesses 70h-7Fh	170h	accesses 70h - 7Fh	1F0h
	7Fh		FFh		17Fh		1FFh
Bank 0		Bank 1		Bank 2		Bank 3	

Fig. 3 Register file map for SFRs and GPRs.



STATUS REGISTER (ADDRESS 03h, 83h, 103h, 183h)

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	C
bit 7							bit 0

STATUS<6:5> RP1: RP0: Register Bank Select bits (used for direct addressing)
11 = Bank 3 (108h – 1FFh)
10 = Bank 2 (100h – 17Fh)
01 = Bank 1 (80h – FFh)
00 = Bank 0 (00h – 7Fh)
Each bank is 128 bytes

STATUS<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

STATUS<0> **C**: Carry/ $\overline{\text{borrow}}$ bit (ADDWF, ADDLW, SUBLW, SUBWF instructions)
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

Fig 4. Details of the STATUS register

Even though we have not formally discussed about instruction sets, it's always helpful to be exposed some instructions. So here is one example code to select bank for file register access. Every code in the book follows the Microchip MPLAB assembly programming notations and regulations (Chapter 3 explains about the MPLAB).

This example is to send an output signal (the term 'output signal' means an either High (+5V) or Low (0 V)) to a pin of a port. A port has 8 pins and each pin can work either as an input or output. Each pin's assignment as input or output is determined by a bit assignment of a corresponding direction assignment file register.

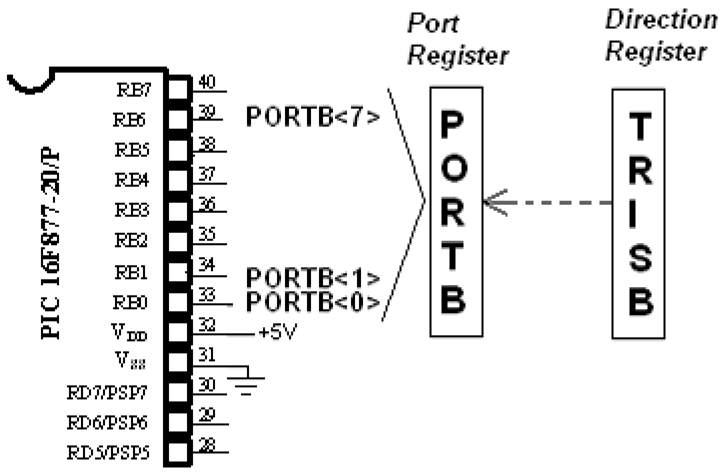


Fig 5. Port register and corresponding direction register

In practice, if you select a pin of PORTB as your output, then the corresponding direction assignment file register is TRISB register. TRISA is for PORTA, TRISC for PORTC, TRISD for PORTD, etc. If you want to use all 8 pins of PORTC as outputs, then you have to reset (or make 0) all 8 bits of TRISC. When you want to use the pin 1 of PORTC (notes as PORTC<1>) as input and all others as outputs, then you set TRISC<1> and reset all other bits of TRISC. Details of I/O port and settings come in next section. The example uses the pin0 of PORTB (noted as PORTB<0>) as an output. We see that PROTB is in Bank 0 and its corresponding direction assignment TRISB is in Bank 1.

```
;Example for Bank Selection
;
            list P = 16F877         ;Target processor = 16F877
;STATUS<6:5> will select banks
;  00   bank0
;  01   bank1
;  10   bank2
;  11   bank3
;
;PORTB is in Bank 0
;TRISB is in Bank 1
;
;Declaration of the SFRs –refer the file register map
STATUS      EQU   0x03        ;STATUS register is located in 03h (all 4 banks)
PORTB       EQU   0x06        ;PORTB in 06h (bank0)
TRISB       EQU   0x86        ;TRISB in 86h  (bank1)
; When started, bank 0 is automatically selected. So we are at bank 0 now
; In other words STATUS<6:5>=00
; Now we want to access and write a byte information to TRISB
; TRISB is in bank 1
; Select bank1
      bsf   STATUS, 0x05      ;set the 5th bit of STATUS
                              ;now STATS<6:5>=01
      movlw 0x00              ;load 00 to W register
                              ;(W register is more like an accumulator)
      movwf TRISB             ;store the value of W to TRISB register
                              ;Now all 8 pins of PORTB are used as outputs
; Now let's access PORTB which is located in bank 0
; bank 0 selection
      bcf   STATUS, 0x05      ;now STATUS<6:5>=00
      bsf   PORTB, 0x00       ;meaning that set the bit 0 of PORTB
                              ;which means, send +5V output through the
                              ;pin 1 of PORTB.
                              ;If you connect an LED to PORTB<0>, the LED
                              ;will turn on.
; end of example
      end
```

In the example code above, the prefix `0x` indicate the number follows is a hexadecimal number. Also, a semicolon (;) starts a comment line of section. The program should end with `end`.

6. Input/Output Ports

Since we already have our example code involving PORTB and TRISB, one of the Input/Output (I/O) ports, it's the best time to explore the I/O ports of 16F877. The I/O Ports are the simplest of peripherals and this is part of the reasons why the microcontroller was named by PIC, which stands for *Peripheral Interface Controller*. They allow the PIC to monitor and control external devices like sensors and relays. For most of the I/O Ports, the direction of I/O pins (input or output) is controlled by the data direction register called TRIS register. A '1' for TRISB<n> makes the pin PORTB<n> an input, while a '0' makes an output. An easy way to remember of this input/output setting is that '1' looks like an alphabet I (Input) and '0' an O (Output). From the previous example code, since all 8 bits of TRISB are reset, all the pins of PORTB are selected as output:

```
movlw 0x00
movwf TRISB       ;TRISB<7:0>=00000000
                  ;This makes all 8 pins of PORTB output pins
```

When we want to make only PORTB<1> as output and all other pins as input, we would do the following code revision.

```
movlw 0xFD        ;which is equivalent to (movlw  b'11111101')
movwf TRISB       ;TRISB<7:0>=11111101
                  ;PORTB<1> as output, all other pins as input
```

As you saw in the example code and above example, to send a value to any file register including the port registers like PORTB, PORTC, etc, you load a select number to **W** register, then move the content of **W** register to a file register.

Figure below shows a typical I/O Port and read and writing operations. The writing to an I/O pin is to send data into a data latch. By the control of the TRIS, the data in available on the pin.

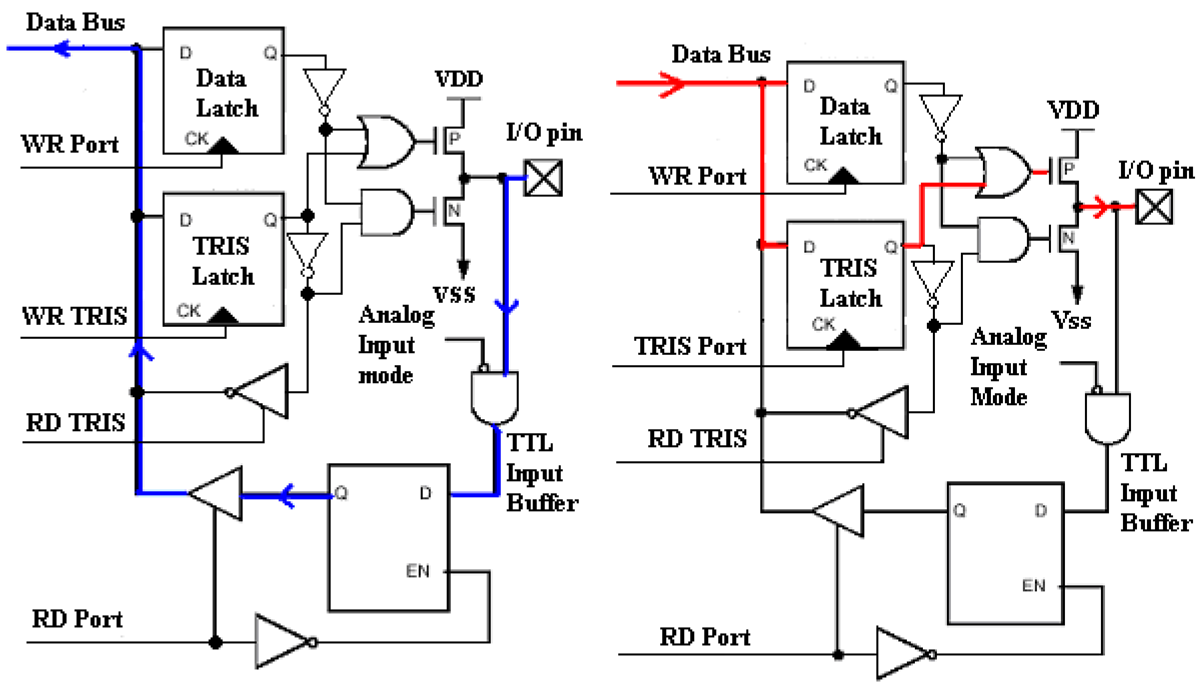


Fig 6. Input (left) and Output (right) operation of a port register.

Then how do we actually write to the pin? In PIC, writing to a port is writing to a port register. This means you can send out all 8-bit information to a port, for example PORTB. Then, how do we write to a port? Writing to a port is to send data in **W** register to the port. In line with the first example code, let's write 11110000 to the pins of PORTB.

```
bsf   STATUS, 0x05      ;we are now in bank 1
movlw 0x00              ;load 00 to W register
movwf TRISB             ;PORTB pins are set as output
bcf   STATUS, 0x05      ;we are now in bank 0
movlw 0xF0              ;W=11110000
movwf PORTB             ;PORTB=11110000
                        ;B0=0, B1=0, B2=0, B3=0, B4=1,
                        ;B5=1, B6=1, B7=1
```

When this code is running inside PIC, you can measure the voltage levels of the pins of PORTB to check the logical information's physical transformation at the pins. A measurement of B0 would give you 0 volt and a measurement of B6 would indicate +5 volt (or the supply voltage to the Vdd pin of the PIC chip). So when you connect an LED to B6, you would see a lit LED.

The above example code is to write a byte information. But we can send only 1 bit information too. This is done by using a bit-oriented instruction instead of a byte-oriented instruction. Let's turn on only B6 of the PORTB.

```
bsf   STATUS, 0x05      ;we are now in bank 1
movlw 0x00              ;load 00 to W register
movwf TRISB             ;PORTB pins are set as output
bcf   STATUS, 0x05      ;we are now in bank 0
bsf   PORTB, 0x06       ;Set the B6 of PORTB
                        ;All the pins are remained with the
                        ;previous values.
```

Care should be exercised in port writing. The writing operation to a pin is, since writing is to write into a latch, actually so-called Read-Modify-Write. In other words, when you want send out '0' to a pin of a port by saying

```
bcf  PORTB, 0x00
```

you may think that the CPU will just clear the bit so that the voltage at the pin is 0 volt. What actually happens is the whole port (PORTB register: 8 bits) is first read (READ), then the bit 0 (single bit) is cleared (MODIFY), then finally new modified value (8 bits including the affected bit) is written (WRITE) back to the port of PORTB.

Actually, any instruction that depends on a value currently in the register is going to be a Read-Modify-Write instruction. This includes ADDWF, SUBWF, BCF, BSF, INCF, XORWF, etc. Instructions that do not depend on the current register value, like MOVWF, CLRF, and so on are not R-M-W instructions.

Let's consider an LED case. Suppose we make PORTB all outputs and drive the pins low. On each of the port pins is an LED connected to ground, such that a high output lights it. Let's suppose that the processor is running very fast, say, at 20 MHz. Now if we go down the port setting each pin in order;

```
bsf   PORTB,0x00
bsf   PORTB,0x01
bsf   PORTB,0x02
.
.
.
```

we may see that only the last pin is set, and only the last LED actually turns on. Probably, all LEDs are dim, kind of half-on.

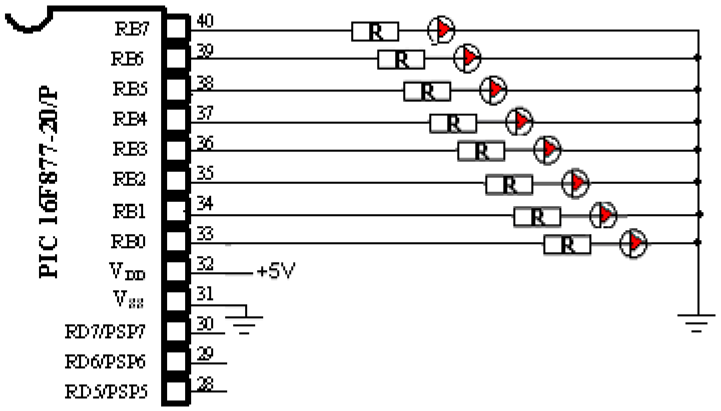


Fig 7. Example of LEDs on PORTB

This is because the LED takes a while to turn on. As each pin was set, the pin before it was not turned yet and so was read as a zero. This zero is written back out to the port latch (R-M-W) which clears the bit you just tried to set the instruction before. So we may have to have a few ms delays between the operations of LED.

Another important consideration about the output at the pin is not the voltage level but the amount current can flow by the voltage. It is clear that there would not be any current when the pin voltage is 0 volt. However, it is not always clear how much current can a source with a +5 volt provide to a connected load like LED. According to the specification of PIC, each pin can drive maximum 25 mA. In other words, if one device needs more than 25mA from the pin, your code may work but your pin cannot activate the device. Fortunately, however, most of devices like LED, relay, motor driver, motor control chip, and others do not require high current driving inputs. So we do not have to worry about the driving power of PIC, but we surely have to check when we connect any devise at the output pin of the PIC of this driving limit of 25mA.

Let's further discuss about LED. Most LEDs have their characteristics specified at a current of 20 mA. So now you see the maximum current driving ability of PIC is not just randomly determined. But we do not light an LED with full brightness; it would shorten the life time of the LED. So we usually add a resister in series to reduce the amount of current flowing through the LED. When current flows through an LED, there is a voltage drop across the LED. The voltage drop is not the same for all types of LEDs, but it is reasonably right to assume that the voltage drop is about 1.7 – 2.0 volt. Therefore, when an LED with a series resistor is connected to an output pin, then the resistor picks up about 3 volt. Now we all know that the resistor value will determine the current through the resistor and the LED. So if you want to have half the maximum brightness, you would want to have only 10 mA flowing. To make that, your resistor should be 3/0.01 = 300 ohm. In this selection, the resistor consumes about 30 mW therefore a resistor wattage rating of 1/8 W is more than enough.

When we read data from a pin, we read it directly from the pin, without the latch involvement. Reading from input pins is the same as reading from the port to **W** register in assembly language programming.

```
bsf   STATUS, 0x05      ;we are now in bank 1
movlw 0xFF              ;load FF to W register
movwf TRISB             ;PORTB pins are set as input
bcf   STATUS, 0x05      ;we are now in bank 0
movf  PORTB,0           ;W now holds PORTB<7:0>
                        ;the trailing 0 is for the destination
                        ;of the operation (chapter 3 discusses about
                        ;the direction of operation in detail.)
                        ;In short, 0 indicates the result is stored in W
                        ; 1 stores in a file register (but 1 is a
                        ; default value, so you do not see 1 in the
                        ; code.)
```

The above byte-oriented file register reading can be replaced to a bit-oriented reading (and monitoring) if only one input pin is the interest.

```
bsf   STATUS, 0x05      ;we are now in bank 1
movlw 0xFF              ;load FF to W register
movwf TRISB             ;PORTB pins are set as input
bcf   STATUS, 0x05      ;we are now in bank 0
btfss PORTB,0x06        ;test the B6 bit of file register PORTB
                        ;Skip the next line of instruction
                        ;if B6 is set.
goto  A                 ;if B6=1 this line is skipped.
goto  B                 ;
```

It is enough for an example code with I/O description. And it's about time to discuss more on the instruction sets of PIC, and the coding and program development environment. Chapter 3 gives details about the instruction sets of PIC 16F877.