EMBEDDED

SYSTEMS

PROGRAMMING

WITH   THE

PIC16F877

Second Edition

By Timothy D. Green

Table of Contents

Preface

This book is intended for use by Junior-level undergraduates, Senior-level undergraduates, and Graduate students in electrical engineering as well as practicing electrical engineers and hobbyists and seeks to provide a gentle introduction to embedded systems programming with the Microchip PIC16F877 microcontroller. After introducing the PIC16F877 and its programming, this book covers the fundamental techniques and advanced level techniques of embedded systems programming in a general sense. The general sense ESP techniques can be applied to *any* microcontroller. There is also an introduction to the fundamentals of digital signal processing (DSP) using the PIC16F877.

I would like to thank Dr. Dan Simon of the Cleveland State University Electrical Engineering Department for his kind and valuable help and suggestions in the preparations for this book. I would also like to thank John R. Owerko and James R. Jackson, both of A.R.F Products, Inc., for their expertise in the security systems market. I owe them both a great debt for my knowledge of security systems and for expanding my knowledge of the techniques of embedded systems programming in general.

Special thanks also go to Sister Renee Oliver who proofread the manuscript and offered many helpful suggestions. Thanks go to my friends Damian Poirier, Jim Strieter, Greg Glazer, Zarif Bastawros, Brian McGeever, Ted Seman, and Jim Chesebrough who offered many helpful suggestions.

Any errors that remain in the text are mine and I will correct them in the next edition.

Timothy D. Green
November 2005
Cleveland, Ohio

List of Figures

Abbreviations and Acronyms

ABS   = Absolute Value
ACCUM   = Accumulator
ADC   = Analog-to-Digital Converter
ADPCM   = Adaptive Differential Pulse Code Modulation
ALU   = Arithmetic Logic Unit

Arccos   = Arc-Cosine
ASCII   = American Standard Code for Information Interchange
ATM   = Automatic Teller Machine
BOR   = Brown Out Reset
BPF   = Band Pass Filter

CK   = Clock
Cos   = Cosine
CPU   = Central Processing Unit
D   = Data
DAC   = Digital-to-Analog Converter

dB   = Decibels
DC   = Direct Current
DDS   = Direct Digital Synthesis
DIP   = Dual Inline Package
DPSK   = Differential Phase Shift Keying

DSP   = Digital Signal Processing
DTMF   = Dual Tone Multi-Frequency
EEPROM   = Electrically Erasable Programmable Read Only Memory
EMC   = Electro-Magnetic Compatibility
EMI   = Electro-Magnetic Interference

EPROM   = Erasable Programmable Read Only Memory
ESP   = Embedded Systems Programming
Fmax   = Maximum Frequency
Fosc   = Oscillator Frequency (of the PIC)
Freq   = Frequency

FSK	=	Frequency Shift Keying
HPF	=	High Pass Filter
Hz	=	Hertz
IIC or I2C	=	Inter-Integrated Circuit
INT	=	Interrupt
I/O	=	Input/Output
ISR	=	Interrupt Service Routine
kHz	=	Kilohertz
LED	=	Light Emitting Diode
LPF	=	Low Pass Filter
Max	=	Maximum
MHz	=	Megahertz
Min	=	Minimum
ms	=	Milliseconds
MSSP	=	Master Synchronous Serial Port
OSC	=	Oscillator
PC	=	Personal Computer or Program Counter
PIC	=	Peripheral Interface Controller
PISO	=	Parallel-In, Serial-Out (Shift Register)
PLL	=	Phase-Locked Loop
POR	=	Power-On Reset
PROM	=	Programmable Read Only Memory
PSP	=	Parallel Slave Port
PWM	=	Pulse Width Modulation
Q	=	Flip-Flop, Counter, or Shift Register Output State (Data Out)
RAM	=	Random Access Memory (A Read/Write Memory)
RC	=	Resistor/Capacitor (Time Constant or Circuit)
RF	=	Radio Frequency
RFI	=	Radio Frequency Interference
ROM	=	Read Only Memory
Sin or sin	=	Sine
SIPO	=	Serial-In, Parallel-Out (Shift Register)
SPI	=	Serial Peripheral Interface
sqrt	=	Square Root
SR	=	Sampling Rate

uC	=	Microcontroller
uP	=	Microprocessor
USART	=	Universal Synchronous/Asynchronous Receiver/Transmitter
WDT	=	Watch-Dog Timer
XTAL	=	Quartz Crystal (Sets Oscillator Frequncy)

Trademarks

IBM and IBM-PC are registered trademarks of International Business Machines, Inc.

PIC, PIC16F87X, PIC16F877, MPLAB, MPASM, In-Circuit Debugger, In-Circuit Serial Programmer are registered trademarks of Microchip Technology, Inc.

CTI and CTI Speech Compressor are registered trademarks of Compressed Time, Inc.

Touch Tone, Unix, C, and C++ are registered trademarks of AT&T, Inc.

Linux is a registered trademark of the Free Software Foundation.

I2C, IIC, and Inter-Integrated Circuit are registered trademarks of Philips Corp.

The PIC Instruction Set, Assembly Keywords, and Mnemonics are Copyrighted by Microchip Technology, Inc.

Maxim and MAX690CPA are registered trademarks of Maxim Corporation.

Borland, Turbo, Turbo C++ are registered trademarks of Inprise, Inc.

Chapter 1:   Introduction to ESP and the PIC

An embedded system is a product which uses a computer to run it but the product, itself, is not a computer. This is a very broad and very general definition. Embedded systems programming, therefore, consists of building the software control system of a computer-based product. ESP encompasses much more than traditional programming techniques since it actually controls hardware in advance of real time. ESP systems often have limitations on memory, speed, and peripheral hardware. The goals of ESP programmers are to get the "maximum function and features in the minimum of space and in minimum time".

Embedded systems are everywhere! Name almost any appliance in your home or office and it may have a microprocessor or a microcomputer to run it. A watch, microwave oven, telephone, answering machine, washer, dryer, calculator, toy, robot, test equipment, medical equipment, traffic light, automobile computer, VCR, CD player, DVD player, TV, radio, and printer all have computers in them to run them.

These examples of embedded systems are simple but the concept of embedded systems applies to much larger systems as well. Overall, there are four levels of size, option, and complexity in embedded systems. These levels are:

1) High Level
2) Medium Level
3) Low level with hardware
4) Low level without hardware

A good example of a high level embedded system is an air-traffic control system. It would use a main-frame computer with many terminals and many users on a time-sharing basis. It would connect to several smaller computers, run the radar, receive telemetry, get weather information, have extensive communications sub-systems, and coordinate all of these function in an orderly, systematic way. It is necessarily a high-reliability system and may, therefore, have extensive backup systems. It would have a custom-built operating system that would be completely dedicated to controlling air-traffic.

An example of a medium level embedded system is a typical automatic teller machine (ATM) at any bank or bank terminal. It may use a more advanced microprocessor with many peripheral functions. Consider that it contains a video terminal, a keyboard, a card-reader, a printer, the money-dispensing unit, a modem, and many input/output ports. The ATM probably doesn't use a custom operating system but would use something off the shelf, like Unix or Linux. The controlling software is probably written in a high-level language like C or C++.

The appliances and other things given on page 11 are all examples of the low-level-with-hardware embedded systems. They do not use microprocessors but do use microcontrollers, which are complete computers on a single chip. Microcontrollers have a CPU, RAM, ROM, and, typically, several peripheral hardware modules which are built-in and are under software control. The PIC16F877 is such a microcontroller. Any of the example products and applications on page 11 could be controlled by the PIC. They could be programmed in C or C++ but care would be needed so as not to use too much RAM or ROM inadvertently. The process or program also must not need very high speed operation – it should not be timing-critical. More control, stability, memory management, and speed can be gained by programming in assembly languages. The programming at the low-level will interact with the hardware in much finer detail than in the medium-level or the high-level systems.

The low-level-without-hardware embedded systems are almost identical to the low-level-with-hardware systems and can run exactly the same products, devices, and applications. The differences which are present in the low-level-without-hardware systems are that the microcontroller and the system have an absolute minimum of hardware peripheral functions. At this level, the software must mimic the desired hardware peripheral functions. This puts a much greater challenge on the ESP programmer. (Assembly language is a MUST.)

There are several characteristics in ESP that separate it from traditional programming techniques. They are as follows:

1) ESP is all about process control and control systems. ESP is what runs a given product.
2) ESP systems must run in "real-time". The program must keep pace or stay ahead of the real world and its timing. For example, a telephone answering machine may use a complex algorithm to compress, expand, encode, and decode speech signals. The ESP program must be able to run these processes as speech is coming in or going out. There must be no delays. A traditional program would not be sensitive to the requirements of speed that are needed here.
3) ESP software must run with infinity-loops. If they didn't, the products could not run at all! In contrast, infinity-loops are the cardinal sin of traditional programming.
4) ESP software often uses "event-driven" techniques, especially at the low-levels. These techniques are highly structured and save operating time. Traditional programming may also use "event-driven" techniques but it is not critical.
5) Low-level ESP software systems must sometimes mimic the hardware that the product needs. There is no parallel to this in traditional programming.
6) Embedded systems usually have far less memory than traditional programming environments. This eliminates heavy nesting of subroutines and recursive subroutine calls.

7) The arithmetic/logic unit of a microcontroller is much smaller than ones in a traditional setting, and, consequently, ESP is not as mathematically oriented as a traditional program.

Embedded systems programming at the low levels is necessarily a multi-disciplinary field. The programmers and designers must consider hardware issues, manufacturing issues, electromagnetic interference/electromagnetic compatibility problems, and noise limitations.

Low-level code doesn't just consist of algorithms but the code, itself, is, at times, a great function of the code geometry. Getting optimum performance of low-level embedded code depends very much on how the instructions are placed in program memory.

The C/C++ language *may* be used in low-level embedded systems programming but not where "fine controls" or "high speeds" are required. The blanket statement that a "C compiler can produce code that is nearly as good as an assembly language program" is OK for traditional programs, high-level ESP, and medium-level ESP. It is *not true* for low-level ESP! Low-level ESP is special and has very exacting demands on its code. (This fact will be explained in detail in Chapter 6.)

The overall scope of this book is to show the reader how to program the PIC, use its peripheral functions, and provide the fundamentals of general ESP techniques.

The plan of this book is as follows:

Chapter 2 introduces microcontrollers in general and details the basic structure of the PIC16F877.

Chapter 3 introduces the most fundamental elements of programming the PIC in assembly language using a simple circuit and program. Several instructions are introduced here. (Chapter 3 is the "Hello World" Chapter.)

Chapter 4 covers the first half of the PIC instruction set by considering the design of a very simple home security system. This method illustrates not only what the instructions are but, also, how to use them in a practical way.

Chapter 5 covers the rest of the PIC instruction set in the same way as in Chapter 4 with a more advanced security system design which includes a keypad and a digit display interface.

Chapter 6 covers the fundamentals of general ESP and shows the reader the style, technique and art of ESP. These techniques and ideas apply to all processors and are the main-stay of all of low-end embedded systems programming. Examples of coding and programs are given in the PIC assembly language.

Chapter 7 covers more advanced ESP techniques, such as sine-wave generation, DTMF signaling, data compression, pulse-width modulation, and testing techniques.

Chapter 8 covers the PIC peripherals with complete examples of how to use them. These include counters, timers, interrupts, the analog-to-digital converter, the pulse-width modulators, measurement hardware, and event generation hardware.

Chapter 9 covers the PIC peripherals for serial data communications. These are the USART, a shift-register interface, and the "Inter-Integrated Circuit" serial interface.

Chapter 10 is an introduction to the fundamentals of Digital Signal Processing (DSP) in an intuitive way and with detailed examples. Filter designs and programs are given in a cookbook fashion. Some advanced and exotic DSP applications and techniques are given.

Appendix A is a detailed view of the PIC instruction set.

Appendix B is a set of useful C++ programs which help the reader design projects for the PIC.

Appendix C is a list of special-function registers and their bit-settings.

Appendix D is a register-file map.

Appendix E shows the PIC16F877 external pins and their functions.

Appendix F shows the sequence of instructions to save registers and restore them when doing a processor interrupt.

Chapter 2: Microcontrollers and the PIC16F877

2.0   Chapter Summary

Section 2.1 covers the types of memories used in the general sense and their organization with respect to the data and addressing. Section 2.2 discusses the memories used in the PIC. Section 2.3 introduces the most fundamental elements of programming at the assembly language level.

2.1   Memory and Memory Organization

A microcontroller is a complete computer system on a single chip. It is more than just a microprocessor: It also contains a Read-Only Memory (ROM), a Read-Write Memory (RAM), some input/output ports, and some peripherals, such as, counters/timers, analog-to-digital converters, digital-to-analog converters, and serial communication ports.

The internal view of a typical microprocessor is shown in Figure 2-1 and is composed of three things: an arithmetic/logic unit (ALU) which performs calculations on data; a set of registers which hold the user's data and the system's data; and a control unit which orchestrates everything and interprets and executes the user's instructions. As far as the microprocessor is concerned, it assumes that there are sets of data memories and program memories (RAM and ROM) in the system. The only thing the microprocessor has to do is run a cycle of getting new instructions and executing them from the memories.

Both the RAM and the ROM are organized as indexed sets of data words, where each "index" is the "address" of its corresponding data. Both the data and its address codes are numbers represented in binary or hexadecimal.

The RAM is a read-write memory which can rapidly read and write the data. It is a volatile memory which means that it loses its memory when power is removed (turned off). The ROM is for program memory and is "read-only" except in modern variants, such as Electrically Erasable Programmable Read Only Memory (EEPROM) and Flash Memory, which allow data words to be written as well as read. The writing of an EEPROM is not the same as a RAM since the data-writing time of the EEPROM is about ten thousand times as long as the data-writing time of the RAM. The ROM and its variants are non-volatile memories that preserve their memories when the power is removed (turned off).

Figure 2-1
uP Internal View
Block Diagram
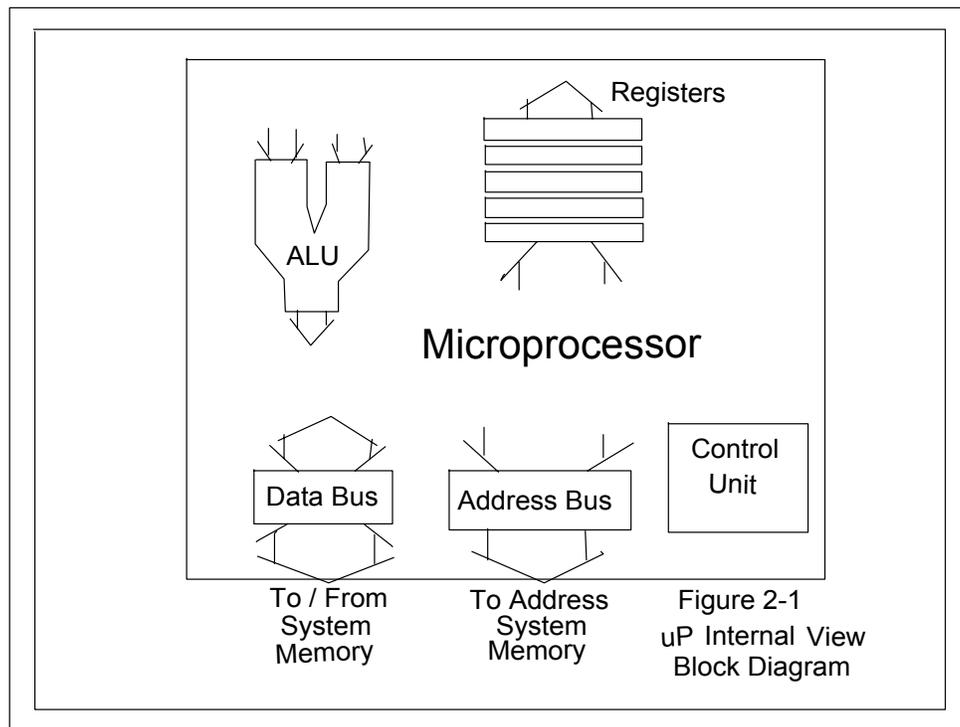
2.2 The PIC16F877

The PIC16F877`s internal block diagram is shown in Figure 2-2. The PIC contains an ALU, which does arithmetic and logic operations, the RAM, which is also called the "register-file", the program EEPROM (Flash Memory), the data EEPROM, and the "W" register. The "W" register is not a part of the register-file but is a stand-alone, working register (also called an "accumulator").

The ALU, the RAM, the "W" register, and the data EEPROM each manipulate and hold 8-bit-wide data, which ranges in value from zero to 255 (or, in hexadecimal, from 0x00 to 0xFF).

The program EEPROM (Flash Memory) works with 14-bit-wide words and contains each of the user's instructions.

It is not uncommon for microcontrollers to have different sizes of data memory and program memory (in the PIC: 8-bits for data and 14-bits for program words). More than that, the key is that the data and program memories occupy separate spaces. This allows access to each at the same time.

RAM

| | | | | | Program EEPROM 8192 Words (14-Bits Long) |

ALU

128 Bytes | 128 Bytes | 128 Bytes | 128 Bytes

W Register

0   1   2   3
4- Banks

Timing & Control

In / Out Ports & Other Peripherals

Data EEPROM 256 Bytes
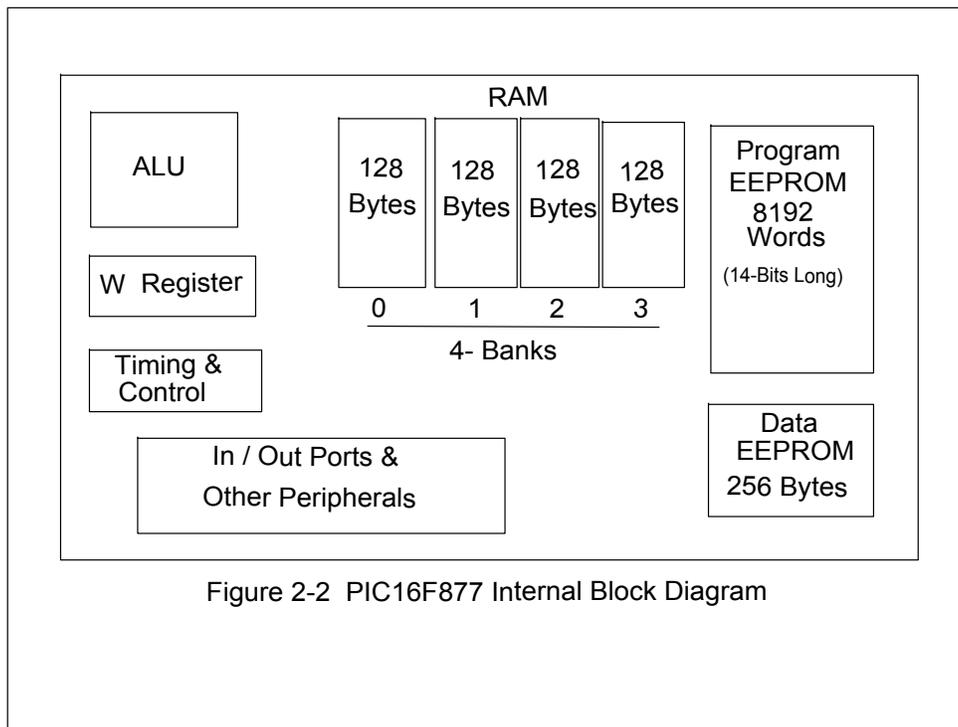
Figure 2-2 PIC16F877 Internal Block Diagram

The PIC's RAM addresses range from zero to 511 but the user can only access a RAM byte in a set of four "banks" of 128 bytes each and only one bank at a time. Not all of this RAM is available to the user as read-write memory, however. Many addresses are dedicated to special functions within the processor but they "look-like" RAM and are accessed the same way.

The PIC's program EEPROM (Flash Memory) has addresses that range from zero to 8191 (0x1FFF). The user's program occupies this memory space.

2.3   Programming The PIC

All types of computer programs can be broken-down into four main sets of actions:

1) Top-Down Execution
2) Conditional Branching
3) Loops
4) Subroutine Calls

Programming the PIC in assembly language is no exception but it is more difficult to work with than high-level languages, like BASIC and C++.

Assembly language uses a one-to-one correspondence of mnemonic words with the binary machine codes that the processor uses to code the instructions. The user writes the program using the mnemonic words called the "source" program and gives this to the program on the PC called the "assembler" which converts it into the machine code of the PIC in the form of a list of hexadecimal numbers. This set of numbers is called the "object" program. The user then writes the object program into the PIC in the downloading process of programming the PIC. When this is done, the PIC is ready to run its new program.

Understanding how to code a program in assembly language is contingent upon understanding how the PIC works at the machine level.

The PIC executes instructions from program memory in sequential addresses, starting from address zero, when the PIC is reset upon power-up. The address of the current instruction being executed is given in a special register called, the "program-counter" (PC). The PIC's control unit automatically increments the program-counter (PC), gets the next instruction, decodes that instruction, and then executes it. If this is done on sequential addresses, this is called, "top-down" execution. There are also ways to do non-sequential-address executions. This is done with special instructions which load new addresses into the program-counter. This is how conditional-branching, loops, and subroutines are done at the machine language level.

Each line of source program code in assembly language has up to four parts: A label, an op-code, an operand, and a comment. This is shown as follows:

LABEL: OPCODE OPERAND(S)   ; COMMENT

The label is an arbitrary name the user picks to mark a program address, a RAM address, or a constant value. If the label has its first character (a letter) that starts in column one of the text, the colon is optional. Otherwise, the colon separates the label from the "Op-Code". The "Op-Code" is short for, "Operation-Code", and is the mnemonic name for the instruction to be executed. The operand or operands are the data or the address that the instruction uses when it is to be executed. This is where labels come into play such as when an instruction needs a new address. Comments are optional and must begin with a semi-colon. Comments are for documenting the source program so that it will be easy to read and understand. For example, the following lines are valid source program code lines:

```
LOOP:     BSF       FLAGS,2          ; Set Alarm Flag Bit
MAIN:     CALL      SORT_SUB         ; Sort the Data

          DECFSZ   TEMP,F
```

; This is an example of a line which consists of only a comment

MAIN   CALL   SORT_SUB ; Sort the Data (No Colon in Label)

Chapter 3:   Simple PIC Hardware
& Software ("Hello World")

3.0   Chapter Summary

Section 3.1 introduces a simple PIC system and analyzes its controlling program. It looks at each instruction, examines its structure and coding, and sets a foundation for proper usage of assembly language. Section 3.2 summarizes the instructions and concepts covered in Section 3.1.

3.1   A Simple Example System

An external view of the PIC with its pins is shown in Figure 3-1. This is the most basic view of the PIC's functions. Most pins have a second use, or, even a third use: to run the PIC's peripherals, such as the ADC, the timers, and the serial ports, to name a few. These other functions will be shown later as they are needed.

The basic function of these pins is for digital inputs and digital outputs. The individual bits on each of the input/output ports (A-through-E) can each be selected as "input" or "output" by special configuration registers in the RAM. The software must set these bits before the ports can be used. This will be shown in more detail shortly.

An example circuit which uses the input/output (I/O) ports "B" and "C" is shown in Figure 3-2. This is a simple, bare minimum, PIC example circuit that serves to introduce some simple software and instructions. Port "B" has a set of 8 DIP switches with resistor pull-ups on it to allow data from these switches to be read into the PIC. Port "C" drives a set of 8 LEDs through resistors to allow the PIC to send out and display its data. The power supply must drive both sets of power pins as shown in Figure 3-2. The clock input, OSC1, is driven by an external oscillator module as shown. Also, a 10K Ohm pull-up resistor is used on Pin 1 (/MCLR) to keep the PIC out of its "Reset" state.

The following program can be used to run the circuit of Figure 3-2:

```
LIST  P = 16F877
INCLUDE "P16F877.INC"

ORG         0x0000      ; Program starts at address zero.
NOP
BANKSEL   PORTC       ; Select Bank Zero
MOVLW     B'00000000' ; Reset PORTC
MOVWF     PORTC
```

```
        BANKSEL    TRISC       ; Select Bank One
        MOVLW      B'00000000' ; Make PORTC All Outputs
        MOVWF      TRISC
        MOVLW      B'11111111' ; Make PORTB All Inputs
        MOVWF      TRISB
        BANKSEL    PORTC       ; Select Bank Zero
MAIN:
        MOVF       PORTB,W     ; Read DIP Switches into W-Register
        MOVWF      PORTC       ; Write W-Register to LEDs
        GOTO       MAIN        ; Loop To Address "MAIN"
        END
```
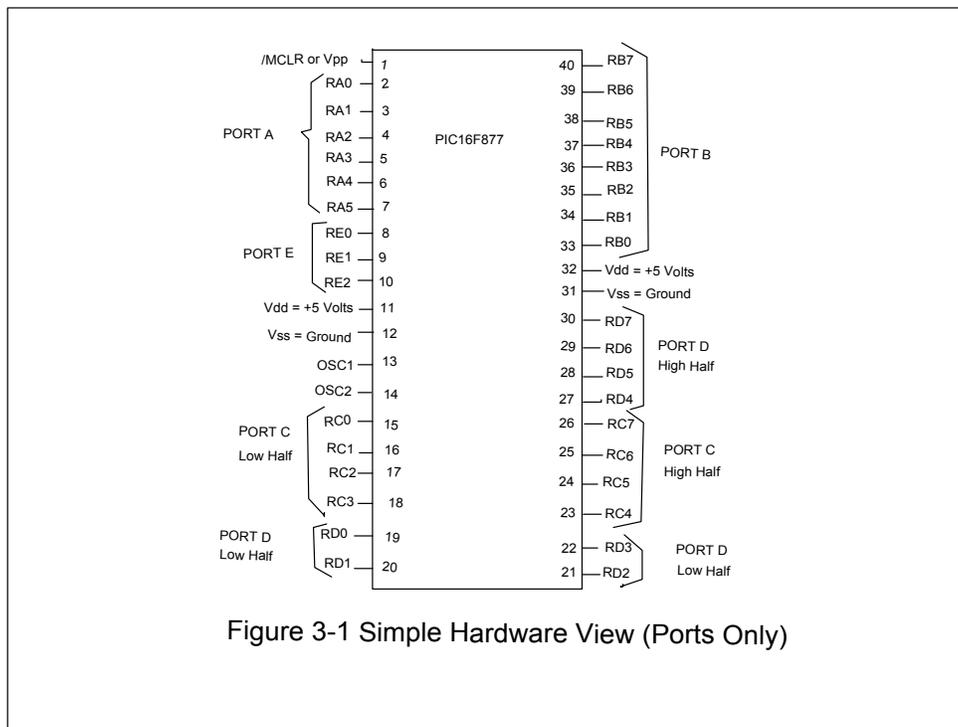

Figure 3-1 Simple Hardware View (Ports Only)

After setting up Port "C" as an "output" and Port "B" as an "input", this program reads the value of the switches on Port "B" and sends it back out to the LEDs on Port "C" in a continuous loop. This may seem like a very trivial program but it is still useful as a test and a simple demonstration of the PIC.

This program is an example of an assembly language program. The MPLAB assembler on the IBM-PC takes this ASCII coded text and converts it into machine code for the PIC to use.
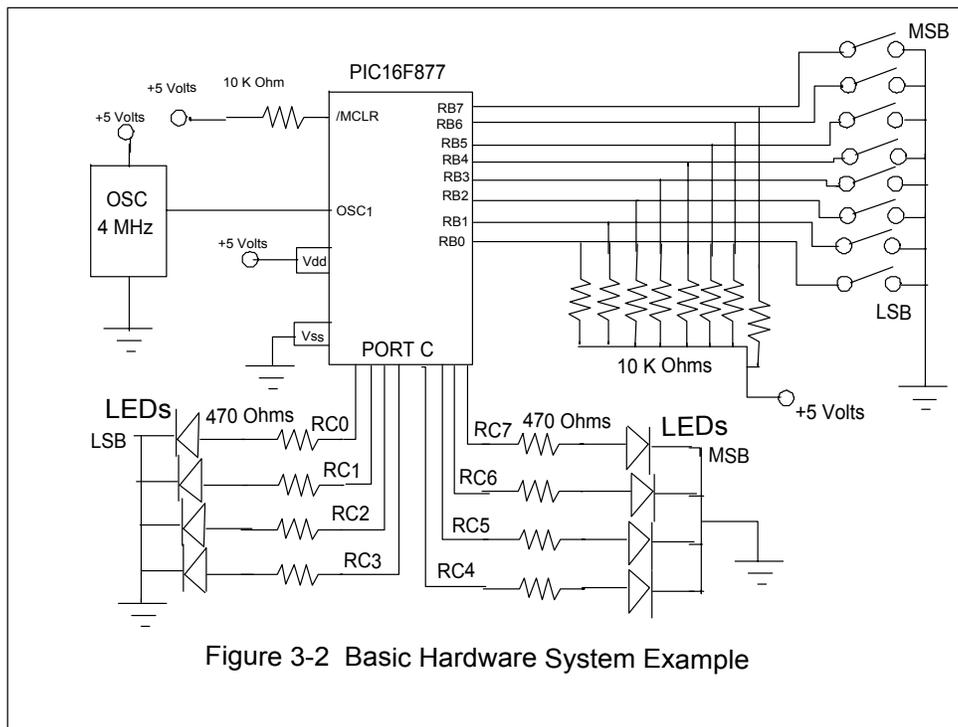
Figure 3-2 Basic Hardware System Example

The major part of this program uses the I/O ports and their configuration registers and we can now look at the RAM or register-file bytes in extensive detail. Appendix D shows the addresses and names of the registers and RAM bytes used by the PIC.

The RAM or Register-File is divided into four banks of 128 bytes each. Only one bank can be used at a time. Not all of the bytes in a bank can be used as user memory because some bytes have special purposes, such as I/O ports. However, all of these bytes work like RAM: They can be read from, and written to, just like a memory byte.

Each register-file byte has a unique address within its bank which ranges from zero to 127 (0x00 to 0x7F). The register, PORTC, for example, is located in Bank zero and has the address 0x07. Notice that some register-file bytes in Appendix D are repeated across each of the banks at the same corresponding addresses. For example, STATUS, PCL, FSR, PCLATH, and INTCON all occupy the same line addresses in each of the four banks. This is so that they can be accessed and have the same values contained in them independent of the bank which is currently selected. For the other registers which are not repeated, the proper bank must be selected before they can be used. For example, storing data in the register byte at address 0x30 in Bank Zero, changing the current bank to Bank One, and then trying to read the right data in the register byte at address 0x30 will not give the same data that was stored even though the addresses were both 0x30. Bank Zero must be selected again as the current bank to get

the right data. However, the RAM addresses from 0x70 through 0x7F can be used in any bank to reference the same data *without* switching banks. This is a very valuable feature!

The usage of the register files will be made more clear as we analyze the program and consider how the instructions are coded and used.

The program starts with the statements, "LIST" and "INCLUDE". These are not instructions in that they are not translated into machine code. They are "assembler directives" which tell the assembler (MPLAB) where to get information about the PIC chip being used (PIC16F877). All of your programs must start with these statements, in this order, as shown.

The statement, "ORG 0x0000", is an assembler directive which tells the assembler at what address the following instructions will start. Here they will start at address zero where "0x0000" indicates "hexadecimal zero".

The next statement is the "NOP" instruction. It is a true instruction that gets translated into machine code. "NOP" stands for "No Operation" (it does nothing). It is coded as 14 zero bits, or, 0x0000 in hexadecimal. Remember that all of the instructions are each 14 bits long.

The next statement is "BANKSEL   PORTC". This is not a true instruction, per se, but it does get translated into two machine instructions which select the current register file bank to be used. Here, "PORTC" is interpreted as "Bank Zero". Later "BANKSEL   TRISC" will be interpreted as "Bank One". In general, the operand of BANKSEL will select the lowest bank number where the operand name is found in Appendix D.

The next statement is the "MOVLW" instruction which means "move the literal value that follows into the W register". Remember that the W register is not a part of the RAM but is an "accumulator" or "working register" within the PIC. The instruction:

        MOVLW   B'00000000'

Says, "move the binary value of all zeros to W". The general machine coding of the "MOVLW" instruction is:

        11 0000 kkkk kkkk

where the "k"s are the single binary bits of the literal data (the data byte). The instruction:

        MOVLW   B'00000000'

Would be coded as:

11 0000 0000 0000   or   0x3000.

Later in the program is the instruction

MOVLW   B'11111111'

Which is coded as:

11 0000 1111 1111   or   0x30FF.

The next instruction is, "MOVWF   PORTC", which means, "move the value in the W register to the register-file byte at the address given (address given here is PORTC)". That is, "Move W to RAM". In general, this is coded as:

00 0000 1fff ffff

where the "f"s are the binary bits of the address for the desired register-file byte. Since Port C has the address 0x07 the coding of "MOVWF   PORTC" would be:

00 0000 1000 0111   or 0x0087.

It would be perfectly legal in the assembly language program to say, "MOVWF 7" to mean, "MOVWF   PORTC" and it would be translated in exactly the same way. However, it would be very confusing to anyone who tried to read the program and, for this reason, it is better to let the assembler keep track of the register-file names and let the assembler translate them to the proper addresses. In general, it is best to let the assembler translate all of the label names (RAM or ROM) in the program to their values. Never use translated numbers directly in the ASCII text of the program. Always use label names. This is a very good programming practice.

At the bottom of the program there is the label, "MAIN", and the instruction, "MOVF PORTB,W". The label "MAIN" is there to mark the address of the "MOVF" instruction so that we can come back to that address. The "MOVF" instruction stands for, "move the register-file byte value to the W register". Specifically, it moves the value in Port B to the W register. (It is also possible to say, "MOVF PORTB,F", which would mean, "move the value in Port B to Port B". This sounds absurd but there is a reason that this is allowed.)

The next new instruction is the, "GOTO   MAIN", which means, "put the address-value at MAIN into the program-counter to transfer to that address". This causes a "loop" to occur in the program. The instruction at the address "MAIN" is the "MOVF".

The last statement is the, "END", which is an assembler directive that means, "end the program".

Now that the program directives and instructions are understood, we can come to a better understanding of how the program works.

The first step in the program is to send zeros to Port C in preparation for setting up Port C to be an output port. This may seem backwards (sending data to a port before it is declared to be an output port) but there is a good reason for it: When the declaration is made as "output", whatever data that is in the port output buffer is immediately sent out. If there is garbage in the port output buffer, that garbage will be the first output from the port. This is why zeros were sent to the port first.

The next steps are to declare Port C and Port B to be as "output" and "input", respectively, by using the TRISC and TRISB registers. If a "1" is sent to a TRISC bit, the corresponding Port C bit will be an "input". If a "0" is sent, the Port C bit will be an "output". This rule applies to all of the ports (A, B, C, D, and E) and their corresponding TRIS registers. (Ports "A" and "E" need additional configurations, however.)

In the "MAIN" loop, the data from the DIP switches on Port B is moved to the W register as an intermediary and is then moved to Port C to display the data on the LEDs. Then the "GOTO" closes the loop to form an infinite loop.

The thing for you to do now is to enter this program in an ASCII text editor and run a simulation of it in the MPLAB. See if you can enter and run this program.

3.2   Summary of Instructions and Concepts

1) When you are using MPLAB, your programs must start with:
        LIST   P=16F877
        INCLUDE "P16F877.INC"
2) The register-file map in Appendix D shows where each special function register-file is and which RAM banks restrict its use.
3) RAM addresses from 0x70 through 0x7F may be freely used by the user and are accessible as the "same" data in any bank.
4) The "ORG" assembler directive tells the program where the current program memory address is. Use, "ORG   0xhex-number".
5) The "NOP" instruction means "No Operation". It does take time for the PIC to run it, however.
6) The "BANKSEL" directive selects the RAM bank for the current use. Consult Appendix D to get the right RAM bank.
7) The "MOVLW" instruction means "Move the Literal value that follows into the W register".
8) The "MOVWF" instruction means "Move the W register's data into RAM". The data in the W register remains unchanged after doing this instruction.

9) The "MOVF" instruction means "Move data from the RAM into the W register".

10) The "GOTO" instruction "Loads the Program Counter with the address (label) that follows" to "Jump" to a new, non-sequential address in program memory.

11) Labels are used to reference addresses in program memory so that "GOTO" instructions can use them.

12) The "END" directive is the last line of any program and is used to signal the end of the assembly process.

13) Always use label-names to reference RAM or program memory.

14) Send zeros to a port before you declare the port as "output" so that garbage will not initially corrupt the port's output.

15) Use the appropriate "TRISx" register to declare a port as "input" or "output".

Chapter 4:   The PIC Instruction Set (Part I)

4.0   Chapter Summary

Section 4.1 covers the PIC instruction set by building a simple security system. It covers the RAM and ROM in more detail, looks at instruction timing, and introduces the use of subroutines. Section 4.2 summarizes the instructions and concepts.

4.1   The PIC16F877 Instruction Set

The PIC16F877 only has 35 instructions and it would be easy to list them all at once. It would, then, be harder to explain them and give examples of how they are used. A much more natural way is to pick a specific application for the PIC then illustrate the instruction set, show how the instructions are used, and show how a program uses sequences of instructions to accomplish the desired tasks.

Let's consider the design of a home security system or a burglar alarm using the PIC. A simple security system circuit is shown in Figure 4-1. The main alarm sensor that is used in this system is the magnetic reed switch. Figure 4-1 shows five of these in series, tied to ground, and connected with a pull-up to the PIC input port pin. Suppose that Port B is configured as an input port and that this line connects to RB0 (bit zero). The magnetic reed switch has two parts: The switch, itself, which is mounted to a door frame or a window frame, and a magnet, which is mounted to the door or the window. When the door or the window is closed, the magnet holds the reed switch "ON" or "shorted". When the door or the window is moved, the magnetic field is broken and the reed switch turns "OFF" or "open". This is the mechanism that trips the alarm.

The software segment that would sense an alarm is:

```
ALARM_TEST:
        BTFSS       PORTB,0
        GOTO        ALARM_TEST
```

Where "BTFSS" means, "Bit-Test-File-Skip-if-Set". That is, if Port B, bit zero (RB0) is sensed as a "one" ("set"), the next instruction will be skipped. When each reed switch is closed, the Port B, bit zero line will read as "zero" since it is tied to ground. When any reed switch is opened, the Port B, bit zero line will be pulled high ("one") and this will be the alarm condition.

Figure 4-1 A Simple Security System
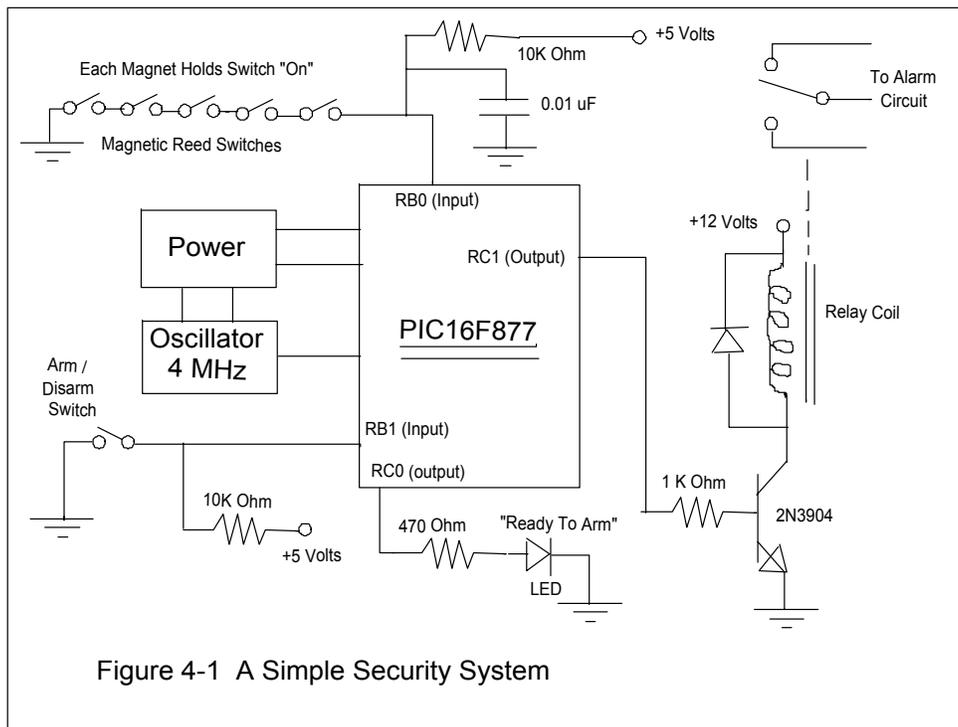
The general format for the "BTFSS" instruction is:

BTFSS   Register-File,Bit-Number

Where the bit-number selects one of eight (8) bits as 0-through-7. There is also a "BTFSC" instruction that is identical to "BTFSS" except that it skips when the tested bit is "Clear" ("zero").

When the alarm condition is sensed, the alarm will be activated by setting Port C, bit one (RC1) to a "one" ("1") as:

BSF   PORTC,1

Which means, "Bit-Set-File". This will turn the transistor "ON", turn the relay "ON", and the relay contacts will switch on the alarm. The alarm can be reset with:

BCF   PORTC,1

Which means, "Bit-Clear-File". Both "BSF" and "BCF" specify one of eight (8) bits as 0-through-7 for any RAM byte or register-file.

The alarm circuit has two other hardware elements that need to be controlled. One is an "Arm/Disarm" switch input on Port B, bit one (RB1) and the other is a "Ready-To-Arm" LED on Port C, bit zero (RC0).

The software that would control this system is as follows:

```
        LIST  P=16F877
        INCLUDE "P16F877.INC"

REED:       EQU  0        ; Reed Switch is PORTB, Bit Zero
ARM:        EQU  1        ; Arm/DisArm Switch is PORTB, Bit One
LED:        EQU  0        ; "Ready-To-Arm" LED is PORTC, Bit Zero
ALARM:      EQU  1        ; Alarm is PORTC, Bit One

        ORG         0X0000
        NOP
        BANKSEL     PORTC       ; Send Zeros to Output Port
        MOVLW       0X00        ; Before Set-Up
        MOVWF       PORTC
        BANKSEL     TRISB
        MOVLW       0XFF        ; Port B is to be Input
        MOVWF       TRISB
        MOVLW       0X00
        MOVWF       TRISC       ; Port C is to be Output
        BANKSEL     PORTB

MAIN:
        BTFSS       PORTB,REED; Test Reed Switches, Skip if "Open"
        GOTO        READY
RESET:
        BCF         PORTC,LED ; Reset "Ready-To-Arm" LED
        BCF         PORTC,ALARM ; Reset Alarm
        GOTO        MAIN

READY:
        BSF         PORTC,LED ; Set "Ready-To-Arm" LED
        BTFSS       PORTB,ARM; Test "Arm/DisArm" Switch, Skip if Set
        GOTO        MAIN

ALARM_SENSE:
        BTFSS       PORTB,REED; Test Reed Switches, Skip if "Set"
                                ; (Alarm)

        GOTO        ALARM_SENSE

        BTFSS       PORTB,ARM; Test "Arm/DisArm" Switch, Skip if Set
```

```
        GOTO           MAIN

SET_ALARM:
        BSF            PORTC,ALARM ; Activate Alarm
        BTFSC          PORTB,ARM; Test "Arm/DisArm" Switch….
        GOTO           SET_ALARM; ---- Still Set, Stay in Alarm
        GOTO           RESET      ; ---- Not Set, Reset Alarm
        END
```


This program uses a new assembler directive called, "EQU", or "Equate" to define constants, such as bit positions and data, as label names. Its usage is:

        Label: EQU   "data"

It is good programming practice to use EQUs to make programs more readable.

This alarm system program is fine if the "Arm/DisArm" switch and the "Ready-To-Arm" LED are located on the outside of the house. If they are located on the inside of the house, this program has a serious problem: It is impossible to leave the house after the system is armed since the alarm will be tripped when the door is opened!

This problem is usually solved by adding a delay of thirty (30) seconds before activating the alarm so that the user can leave the house and not trip the alarm. An external timer could be attached to the PIC to provide this delay but the better solution is to let the PIC generate its own delays in software.

The PIC runs with a clock frequency of 4 MHz and this controls instruction execution speed. The instruction execution speed is one-fourth of the clock frequency. Each instruction executes in one or two instruction cycles, or, in one or two microseconds. Each instruction that "skips", such as, "BTFSS", executes in two instruction cycles when it "skips" and one instruction cycle when it doesn't. The "GOTO" instruction always executes in two instruction cycles.

Let's see how to build a 30-second software delay. The first step is to build a one-millisecond delay as follows:

```
        MOVLW      D'250'          ; Load W with Decimal 250
        MOVWF      TIME            ; Initialize RAM "TIME"
LOOP_ONE_MS:
        NOP                        ; (1) Cycle
        DECFSZ     TIME,F          ; (1) Cycle
        GOTO       LOOP_ONE_MS   ; (2) Cycles
```

This program segment uses the "DECFSZ" instruction, which means, "Decrement-File-Skip-if-Zero". When the Byte's data is decremented, it may be stored back in the RAM

or it may be stored in the W register. Specify the RAM as the destination with the "F" or specify the W register with the "W". This instruction decrements the RAM byte ("TIME", in this case) and skips the next instruction if the result was zero. The initial value of "TIME" is 250 and the loop has a length of four instruction cycles. As the loop counts down from 250 each count adds four microseconds of delay which, when multiplied by 250, gives a total time of about one millisecond.

A loop can then be formed around this loop that counts down from 250 to give a delay of 250 milliseconds. Then another loop can be placed around it that counts down from 120 to give a total delay of 30 seconds!

(There is also an "INCFSZ" instruction which increments a register-file and skips the next instruction if the result is zero. Either "INCFSZ file,F" or "INCFSZ file,W".)

The total nested-loop structure for a 30-second delay is:

```
              MOVLW       D'120'        ; Count 120 of 250 millisecond delays
              MOVWF       TIME2
LOOP_30_SEC:
              MOVLW       D'250'        ; Count 250 of one millisecond delays
              MOVWF       TIME1
LOOP_250_MS:
              MOVLW       D'250'        ; Count of 250 Loops of four cycles
              MOVWF       TIME
LOOP_ONE_MS:
              NOP
              DECFSZ      TIME,F
              GOTO        LOOP_ONE_MS

              DECFSZ      TIME1,F
              GOTO        LOOP_250_MS

              DECFSZ      TIME2,F
              GOTO        LOOP_30_SEC
```

This code must be put into two places in the alarm software: Once just before the "ALARM_SENSE" loop and once just after it. This will allow the user 30 seconds to enter and leave before the alarm goes off.

This is a lot of code to make two copies of and insert into a relatively simple software loop. The resulting code would be much more complicated, much more difficult to follow, and the probability of making a mistake would be much higher. Are there any easier alternatives?

There are. The solution is to use subroutines with the "CALL" and "RETURN" instructions. The "CALL" instruction works just like a "GOTO" instruction except that

the PIC automatically saves the address of the next instruction after the "CALL" in a special memory. This action is automatic and invisible to the user. When you are finished with the subroutine, you use the "RETURN" instruction to tell the PIC to go back to the part of the program where you left off. That is, to get the address out of the special memory and "GOTO" it automatically. The "CALL" and "RETURN" instructions each take two instruction cycles to execute.

For example, the one-millisecond delay can be accomplished with the following subroutine:

```
DELAY_ONE_MS:
        MOVLW       D'250'          ; Count 250 Loops
        MOVWF       TIME
LOOP_ONE_MS:
        NOP                         ; Four Cycle Loop
        DECFSZ      TIME,F
        GOTO        LOOP_ONE_MS
        RETURN
```

This would give a delay of one millisecond each time it was called, as follows:

```
        CALL        DELAY_ONE_MS
```

We could also make a 250-millisecond delay subroutine as follows:

```
DELAY_250_MS:
        MOVLW       D'250'          ; Count 250 Milliseconds
        MOVWF       TIME1
LOOP_250_MS:
        CALL        DELAY_ONE_MS
        DECFSZ      TIME1,F
        GOTO        LOOP_250_MS
        RETURN
```

Notice that this subroutine calls the "one-millisecond" subroutine 250 times. There is no law that says you can't have one subroutine that calls another subroutine – you can – it's called "nesting" subroutines. The PIC allows you to nest subroutines up to only eight (8) levels, however.

The 30-second delay can then be formed as the following subroutine:

```
DELAY_30_SEC:
        MOVLW       D'120'          ; Count 120 of 250-Millisecods
        MOVWF       TIME2
LOOP_30_SEC:
```

32

```
CALL        DELAY_250_MS
DECFSZ      TIME2,F
GOTO        LOOP_30_SEC
RETURN
```

4.2   Summary of Instructions and Concepts

1) Subroutines may be nested up to eight (8) levels deep.
2) Each instruction takes either one or two instruction-cycles to run.
3) Each instruction-cycle time runs in the period of the PIC's oscillator frequency, Fosc, divided by four (4). If Fosc = 4 MHz, each instruction-cycle will run in one microsecond.
4) The "EQU" directive is used to make user-defined labels for RAM addresses, constants, and bit-positions.
5) The "GOTO", "CALL", and "RETURN" instructions each take two instruction-cycles to run. Also, anytime an instruction "skips" it takes two instruction-cycles to run.
6) New Instructions Covered:

```
BTFSS   file,bit
BTFSC   file,bit
BSF     file,bit
BCF      file,bit

DECFSZ    file,destination
INCFSZ    file,destination
CALL       address
RETURN
```

Chapter 5:   The PIC Instruction Set (Part II)

5.0   Chapter Summary

Section 5.1 discusses the need for a more advanced security system. Section 5.2 introduces the keypad and display interfaces. Section 5.3 introduces the processor status flags. Section 5.4 details the keypad software. Section 5.5 details the LED digit display software. Section 5.6 explains indirect RAM addressing. Section 5.7 covers more advanced general features of the PIC processor. Section 5.8 shows how the keypad and the display software work in harmony with each other. Section 5.9 is a last look at the security system. Section 5.10 summarizes the instructions and concepts of Chapter 5.

5.1   Introduction

The basic PIC instructions were introduced in Chapter 4 with an example of a simple security system. In a similar way, the remaining instructions will be introduced with an example of a more complex security system.

A more complex system is needed for a large house or an industrial user. If there are many rooms, doors, and windows, a single "Ready-To-Arm" LED is not enough to give the location of the "insecure" site when the user desires to set the alarm. Also, a single "Arm/DisArm" switch presents no difficulty to a thief who knows where the switch is and wants to disable the alarm. The security system should also offer the user a, "Home/Away" setting so that a remnant of the system can still work while the user is at home (i.e., when he or she is asleep or wants to use the bathroom without triggering the alarm).

These problems could be solved by having many "Ready-To-Arm" LEDs and many "Arm/DisArm" switches but this is not practical when flexible and compact software is available. The solution is to use a twelve-key keypad and a set of seven-segment LED digits to get and display more complex information to run the security system. For example, the LED digits could display a message like, "Zone 3 Not Ready To Arm", or, "Office 407 Not Ready". These messages could be static, or constantly displayed, or they could repeatedly scroll across the display over a few seconds. Code numbers could be entered on the keypad to set-up and run the system. An "Arm/DisArm" switch is not a problem for a would-be thief but entering a four-digit security-code that could be changed daily is much more formidable!

Before we can introduce more instructions and show the software that could do these things, we need to show how the keypad and the LED digits are interfaced to the PIC.

5.2 Keypad And Display Interface

A twelve-key keypad usually consists of a matrix of twelve switches, as shown in Figure 5-1. Three lines connect the columns while four lines connect the rows. When a key is pressed, one column and one row are connected. The PIC must use its software to scan each column-line and look for a row-line that is connected, if, and when, a switch is pressed. The PIC keypad interface circuit is shown in Figure 5-2.
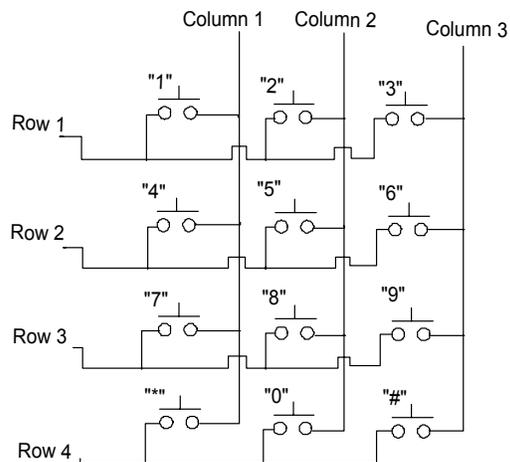
Figure 5-1 Twelve-Key Matrix Keypad

Figure 5-2 PIC Matrix Keypad Interface Circuit
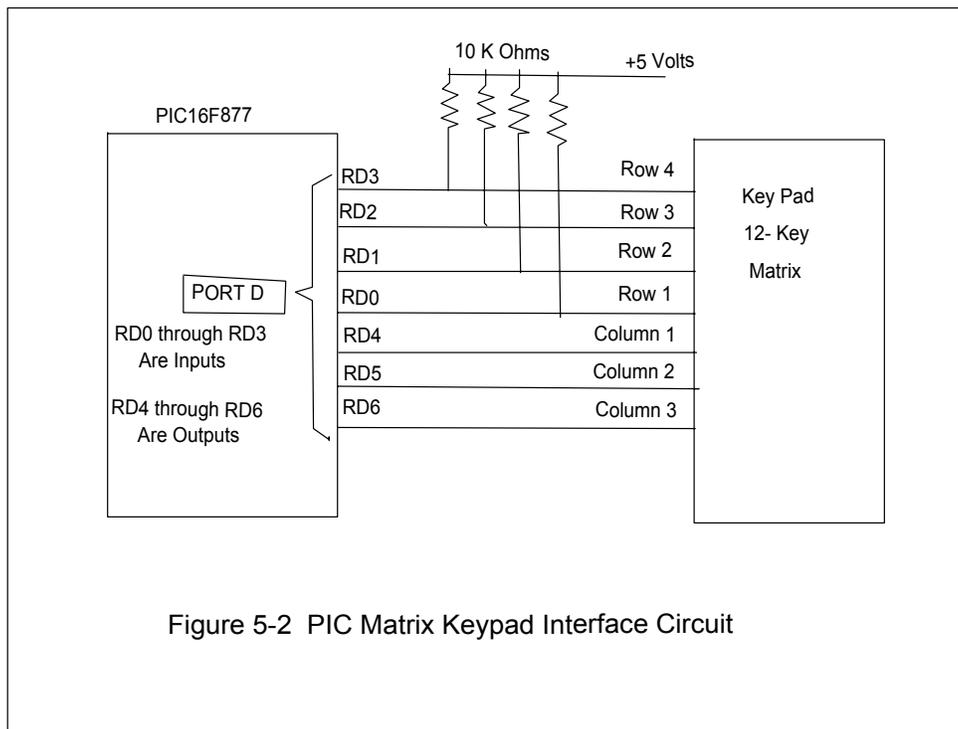
A typical seven-segment LED digit is shown in Figure 5-3. The digit consists of eight (8) LEDs: Seven for the segments and one for the decimal point. The LEDs are connected together in a "common-anode" or a "common-cathode" form. If only one digit is to be used, the common-cathode is tied to ground or the common-anode is tied to five volts, and the PIC would drive each segment LED through a 470 Ohm resistor, as shown in Figure 5-4. If more than one digit is to be used, the digits are usually multiplexed, as shown in Figure 5-5. Multiplexing digits is a way of saving PIC output port lines by time-sharing them. Let's see how this works.

Figure 5-3 Seven Segment LED Digit Display
In Common Cathode and Common Anode Forms
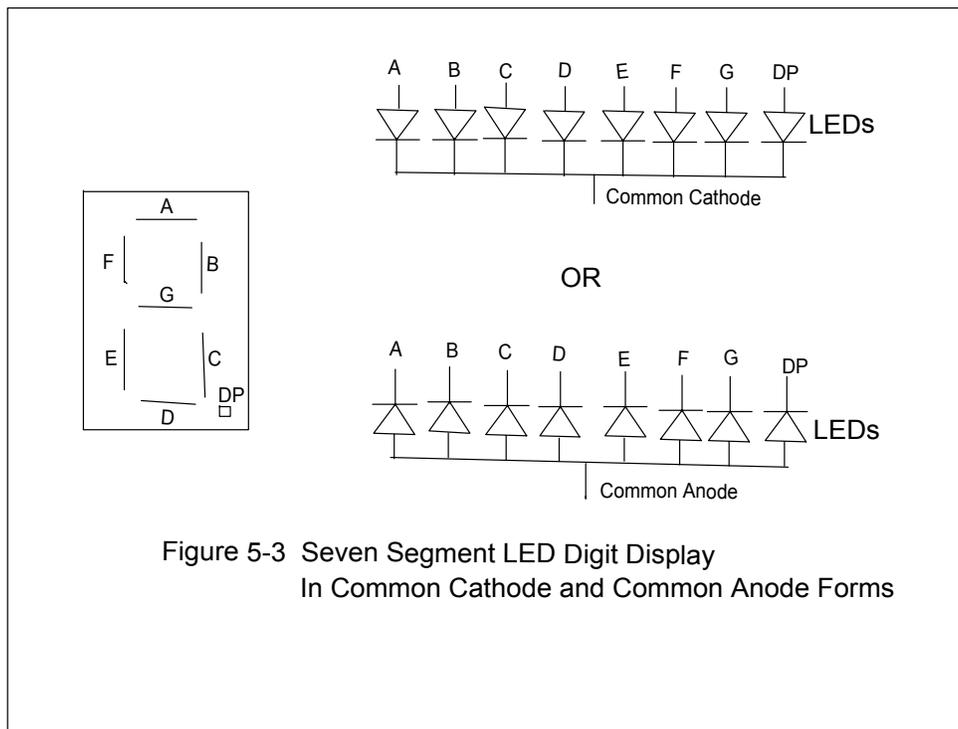
For digit multiplexing, the corresponding segments of each digit are wired together in parallel, as shown in Figure 5-5. Each common-anode or common-cathode of a single digit is driven by a transistor. The idea is to turn on one transistor at a time and supply that digit's segments for a few milliseconds at a time and then turn it off. Then repeat this for each transistor and digit's segments for the whole display, over and over. The overall effect on the human eye is that all of the digits appear to be "on" at the same time! Each digit gets the right information and there is no blurring or garbage.
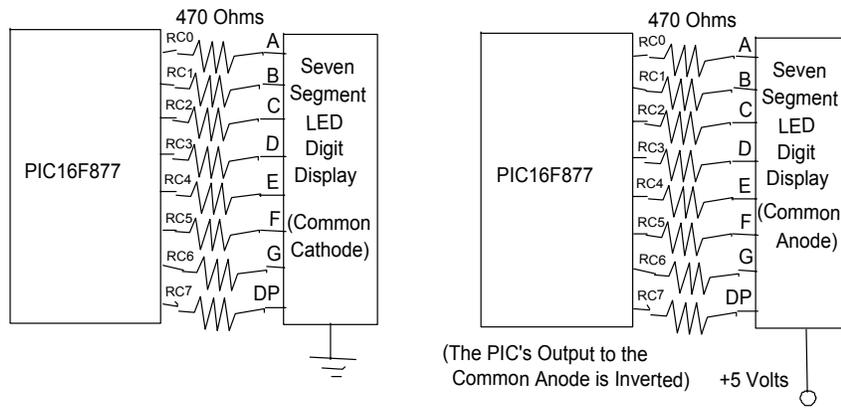
Figure 5-4 Single LED Digit Drives for
Common Cathode and Common Anode Forms
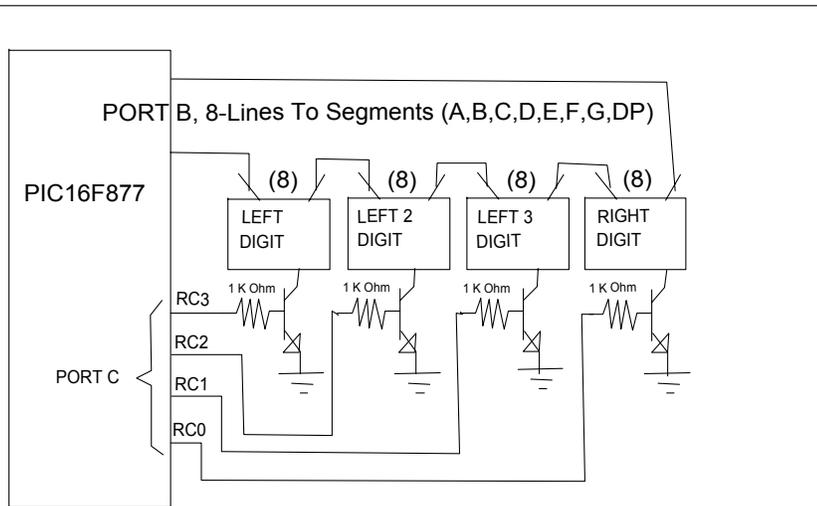
Figure 5-5a Multiplexed LED Digit Drives for Common Cathode Form

Figure 5-5b Multiplexed LED Digit Drives for Common Anode Form
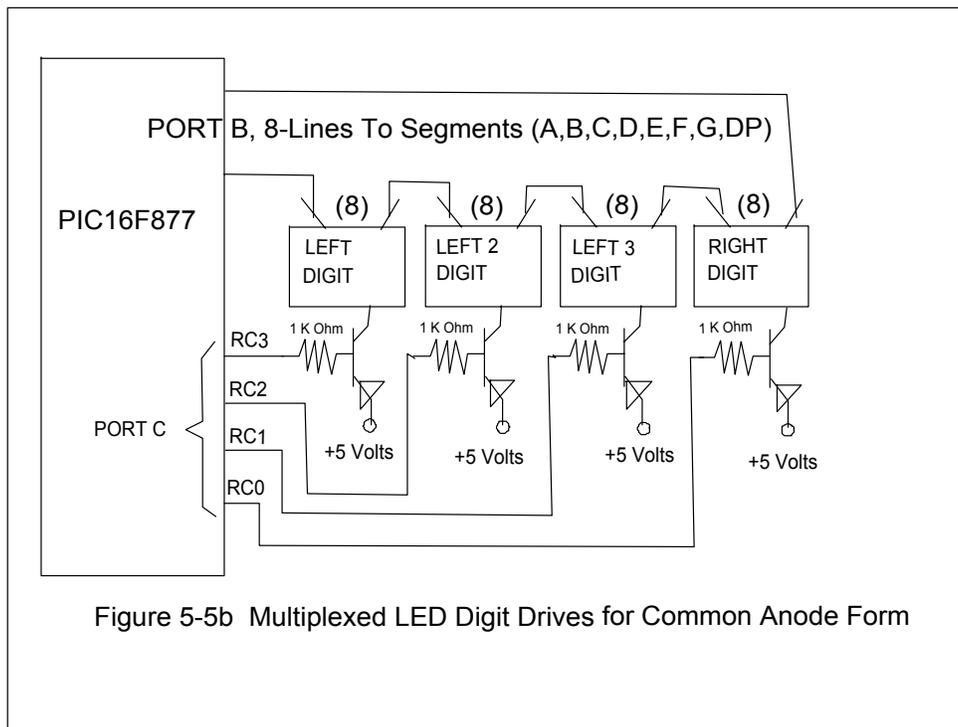
5.3   The STATUS Register and Flag Bits

We are almost ready to begin a full discussion of how the keypad and the LED digits are run in software. First, is should be noted that the instructions that are needed to do this affect flag bits in the status register. Three flag bits are affected automatically as a result of doing each of these instructions.

The STATUS register is shown in Appendix C. The bits RP1 and RP0 are set by the user to select one of the four banks of the RAM registers. This can be done with the BCF and BSF instructions or with the BANKSEL directive. The BANKSEL directive produces the right combinations of BCF and BSF instructions for the STATUS register to select the appropriate RAM bank.

The three flag bits that interest us most are the "Z", the "DC", and the "C" bits. The "Z" bit is set if the instruction produces a result of zero. The "C" bit is the "carry" bit and shows the carry-out of the seventh bit of the result (The result bits range from zero-through-seven). The "DC" bit shows the "carry-out" of the third bit of the result (this is used for binary-coded-decimal arithmetic). A complete list of which instructions affect the flag bits, and how, can be found in Appendix A. The "MOVF f,d" instruction is the only instruction that we have seen so far (Chapter 3) that affects a flag bit.

Specifically, it only affects the "Z" bit, showing if the value or data that is moved is zero (Z = 1) or not (Z = 0).

We are now ready to write the software to control the keypad and the LED digits.

5.4 The Keypad Software

Look again at Figure 5-2. The row lines are pulled-up to five volts (Logic "1") and are connected through the key-switches to the column lines. The idea is to set the column lines low (Logic "0"), one-at-a-time, and search for a low (Logic "0") on the row lines. If a row line is low, a key has been pressed, and the scanning software subroutine returns a number code corresponding to the key. If no key is pressed, the routine returns a code of zero.

The scanning subroutine is as follows:

```
KEY_SCAN:
        BCF         PORTD,4     ; Column 1 = LOW
        BSF         PORTD,5     ; Others = HIGH
        BSF         PORTD,6

        BTFSS       PORTD,0     ; Row 1
        RETLW       1           ; Key = "1"
        BTFSS       PORTD,1     ; Row 2
        RETLW       4           ; Key = "4"
        BTFSS       PORTD,2     ; Row 3
        RETLW       7           ; Key = "7"
        BTFSS       PORTD,3     ; Row 4
        RETLW       10          ; Key = "*"

        BSF         PORTD,4
        BCF         PORTD,5     ; Column 2 = LOW

        BTFSS       PORTD,0     ; Row 1
        RETLW       2           ; Key = "2"
        BTFSS       PORTD,1     ; Row 2
        RETLW       5           ; Key = "5"
        BTFSS       PORTD,2     ; Row 3
        RETLW       8           ; Key = "8"
        BTFSS       PORTD,3     ; Row 4
        RETLW       11          ; Key = "0"

        BSF         PORTD,5
        BCF         PORTD,6     ; Column 3 = LOW
```

```
BTFSS      PORTD,0    ; Row 1
RETLW      3          ; Key = "3"
BTFSS      PORTD,1    ; Row 2
RETLW      6          ; Key = "6"
BTFSS      PORTD,2    ; Row 3
RETLW      9          ; Key = "9"
BTFSS      PORTD,3    ; Row 4
RETLW      12         ; Key = "#"
RETLW      0          ; No Key Pressed
```

The new instruction used in this subroutine is "RETLW k", which means, "Return from subroutine with the W register containing the eight-bit value, k". Thus, when KEY_SCAN returns, the key-code will be in the W register. The "RETLW" instruction runs in two instruction cycles.

The KEY_SCAN routine can be improved by eliminating the repeated code to scan the rows and replacing them with subroutine calls to a "ROW_SCAN" routine. Notice that each of the row scans in KEY_SCAN return values as:

(1,4,7,10) (2,5,8,11) and (3,6,9,12).

These are additions of zero, one, and two on the base values of (1,4,7,10). Assume that the ROW_SCAN routine will also return a zero if no keys are pressed. The improved KEY_SCAN routine is as follows:

```
KEY_SCAN:
       BCF      PORTD,4    ; Column 1 = LOW
       BSF      PORTD,5    ; Others = HIGH
       BSF      PORTD,6

       CALL     ROW_SCAN ; (W) = Code
       ADDLW    0          ; Add Zero to Set Zero Flag
       BTFSS    STATUS,Z
       RETURN              ; (W) = 1,4,7,10

       BSF      PORTD,4
       BCF      PORTD,5    ; Column 2 = LOW

       CALL     ROW_SCAN ; (W) = Code
       ADDLW    0          ; Add Zero to Set Zero Flag
       BTFSC    STATUS,Z
       GOTO     KEY_SCAN2

       ADDLW    1          ; Adjust (W)
       RETURN              ; (W) = 2,5,8,11
```

```
KEY_SCAN2:
        BSF          PORTD,5
        BCF          PORTD,6      ; Column 3 = LOW

        CALL         ROW_SCAN ; (W) = Code
        ADDLW        0            ; Add Zero to Set Zero Flag
        BTFSS        STATUS,Z
        ADDLW        2            ; Adjust (W)
        RETURN                    ; (W) = 3,6,9,12  OR  (W) = 0


ROW_SCAN:
        BTFSS        PORTD,0
        RETLW        1            ; Key "1"
        BTFSS        PORTD,1
        RETLW        4            ; Key "4"
        BTFSS        PORTD,2
        RETLW        7            ; Key "7"
        BTFSS        PORTD,3
        RETLW        10           ; Key "*"
        RETLW        0            ; No Key Found
```

The new instruction used in the new KEY_SCAN routine is, "ADDLW k",
which means, "Add the W register and the eight-bit value, k, and put the result in the W
register". The "ADDLW" instruction affects the "Z", "DC", and "C" status flag bits.

Addition can also be done between a RAM byte (register-file) and the W register
with the "ADDWF f,d" instruction. This does, "W = (file) + W" or "(file) = (file) + W"
and it also affects the "Z", "DC", and "C" status flag bits.

Using the "ADDLW" instruction it is possible to make more improvements in the
KEY_SCAN and ROW_SCAN routines. In the original KEY_SCAN each set of row
value codes had a difference of three (i.e., From "1", "4" = "1" + 3, "7" = "4" + 3, …).
What if the ROW_SCAN routine were called with the first row code in W and each time
the key test failed, we add three to W? Also, the "Z" flag can be set within the
ROW_SCAN routine so we don't have to check it in the KEY_SCAN routine. Can you
write an improved version of the KEY_SCAN and ROW_SCAN routines using these
techniques? Can you think of even more improvements?

5.5   The LED Display Software

Let's now look at the software controls for the four-digit, multiplexed, seven-segment display shown in Figure 5-5a. Port B of the PIC controls the segments while Port C, bits (0,1,2,3), control the digit-drives.

The display subroutine software is as follows:

```
DISPLAY:
        CLRF        PORTC        ; Turn Off All Segments and Digits
        CLRF        PORTB
        MOVF        LEFT_DIGIT,W      ; Get Left Digit's Segment Codes
        MOVWF       PORTB        ; Set-Up Segments for Activation
        BSF         PORTC,3               ; Activate Digit-Drive
        CALL        DELAY_5_MILLISECONDS

        CLRF        PORTC        ; Turn Off All Segments and Digits
        CLRF        PORTB
        MOVF        LEFT_DIGIT2,W     ; Get Left2's Segments
        MOVWF       PORTB
        BSF         PORTC,2
        CALL        DELAY_5_MILLISECONDS

        CLRF        PORTC        ; Turn Off All Segments and Digits
        CLRF        PORTB
        MOVF        LEFT_DIGIT3,W     ; Get Left3's Segments
        MOVWF       PORTB
        BSF         PORTC,1
        CALL        DELAY_5_MILLISECONDS

        CLRF        PORTC        ;Turn Off All Segments and Digits
        CLRF        PORTB
        MOVF        RIGHT_DIGIT,W     ; Get Right-Digit's Segments
        MOVWF       PORTB
        BSF         PORTC,0
        CALL        DELAY_5_MILLISECONDS
        CLRF        PORTC
        CLRF        PORTB
        RETURN
```

One new instruction used is, "CLRF f", which means, "Clear Register-File f". There is also, "CLRW", which means, "Clear W". Both of these set the "Z" (Zero) flag so that Z = 1. The "C" and "DC" flags are not affected.

The "DISPLAY" routine must not have very much delay between calls, otherwise the digits will appear dim or flicker.

The use of the "DISPLAY" routine presupposes that the proper display codes are already in the digit RAM spaces (LEFT_DIGIT, …, RIGHT_DIGIT). What are these display codes and how do we get them? Each display segment shown in Figure 5-3 corresponds to a bit position on Port B as:

Segment	Port B Bit
"a"	0
"b"	1
"c"	2
"d"	3
"e"	4
"f"	5
"g"	6
"d.p."	7

If we were to make a table of base-ten digits to display and their display codes, it would look like:

Digit	Display Code
0	0x3F
1	0x06
2	0x5B
3	0x4F
4	0x66
5	0x6D
6	0x7D
7	0x07
8	0x7F
9	0x6F

This table would be used, directly, by the following subroutine:

```
GET_DISPLAY_CODE:
        ADDWF     PCL,F          ; Add (W) to PCL, With Result in PCL
        RETLW     0x3F           ; "0"
        RETLW     0x06           ; "1"
        RETLW     0x5B           ; "2"
        RETLW     0x4F           ; "3"
        RETLW     0x66           ; "4"
        RETLW     0x6D           ; "5"
        RETLW     0x7D           ; "6"
        RETLW     0x07           ; "7"
        RETLW     0x7F           ; "8"
```

```
        RETLW      0x6F            ; "9"
```

When this routine is called, the user puts the digit to find in the W register. The first instruction, "ADDWF PCL,F", adds the digit in W to the low-half of the program-counter, PCL, and puts the sum in PCL. In effect, this does a "GOTO"-like instruction and goes to the corresponding RETLW instruction where W is filled with the proper display code upon returning.

        For example, suppose that W = 1 when "GET_DISPLAY_CODE" is called. When control is transferred to the "ADDWF PCL,F" instruction, the whole program-counter is incremented, automatically, to get to the next instruction, and then W is added to the low-half of the program-counter to point to the proper RETLW instruction. This is the, "RETLW   0x06", which has the display code for "1", and this code is in the W register when the subroutine returns.

        Remember that the program counter is the register where the PIC keeps track of the address of the current instruction being executed. It is actually composed of two registers: A "high" half and a "low" half. This is because the PIC can address 8 K of program memory and it needs 13 bits to do this. These must be split into two 8-bit bytes. When the program counter is incremented, the low half is incremented first, and, if a carry is generated, it is automatically added to the high half. The low half is the register "PCL". The high half is not directly accessible but it can be set-up through the register "PCLATH" (We will see more of this later.)

        Any table look-up routine, like "GET_DISPLAY_CODE", can be used with a number of entries up to or equal to 255. Some care must be taken, however, for *any* table, large or small. The "ADDWF PCL,F" instruction *only* adds to the *low-half* of the program-counter. There is *no* carry propagation to the high-half. If, for example, the "ADDWF PCL,F" instruction is located at the address 0x01FC and the table extends for ten entries, a value in W equal to four (4) will transfer control to the address 0x0100 and whatever is in that location will be treated as if it were an instruction (it may be) and the program will continue there! You *must* make sure, by using the "ORG" directive, to set an address that allows a large enough space to fit the table.

        If the CALL to a table look-up routine is *not* within the same 256-byte address block as the table itself, the table's high-half address must be loaded into the PCLATH register before the CALL is made. For example,

```
        ORG         0x0100
        ------
        ------
        MOVLW       HIGH  GET_DISPLAY_CODE
        MOVWF       PCLATH
        MOVF        DIGIT,W
        CALL        GET_DISPLAY_CODE
```

```
          -----
          -----
          ORG        0x0380
GET_DISPLAY_CODE:
          ADDWF      PCL,F
          RETLW      0x3F
          -----
          -----
```

This code gets the high-address of the GET_DISPLAY_CODE routine into the PCLATH register before the CALL. The CALL is in the 0x01 block while the routine is in the 0x03 block.

5.6   Improved DISPLAY and Indirect Addressing

In actual practice, the "DISPLAY" routine would be done differently to improve its usage of delay timings. The five-millisecond delay routine was called four times – once for each digit. This eats up too much time. It would be better to call the "DISPLAY" routine once every five milliseconds and let the program keep track of which digit is to be displayed. This requires an addressing technique called, "indirect addressing".

Indirect addressing allows you to use a RAM location to index the address of *another* RAM location. The index RAM location is called, "FSR", and the RAM address to be indexed is put into the "FSR" register-file. With the address of the RAM to use, in place, in the "FSR" register, the user works with a RAM location called, "INDF". See Figure 5-6 for more information and explanation of the concept of indirect addressing.

Suppose we are in Bank Zero and we want to index 16 Bytes in a row in RAM starting from Address = 0x30 to Address = 0x3F. This will be a RAM Data Array.

1) Move the Address Value = 0x30 into the FSR register.
2) Work with *any* instruction that references a RAM Byte, where "INDF" is used in place of the desired RAM Byte. For example:

    INCFSZ   INDF,F   will increment the Byte at Address = 0x30
    And skip if zero.

    MOVF   INDF,W   will move the data in Address = 0x30 into W.

    BSF   INDF,2   will set bit two of the Byte at Address = 0x30.

3) If we increment the "FSR" register (by doing INCF   FSR,F) and do the examples above, we are *now* working with the Data at Address = 0x31.

Suppose we want to Clear this Array. Do:

```
                MOVLW      0x30            ; Set-Up the Start of the Array
                MOVWF      FSR             ;--- Start in FSR
                MOVLW      D'16'           ; Count Out 16 Bytes
                MOVWF      COUNTER
LOOP:
                CLRF       INDF            ; Clear the Byte at FSR's Address
                INCF       FSR,F           ; Increment FSR's Address
                DECFSZ     COUNTER,F
                GOTO       LOOP
```

Figure 5-6   Illustration of Indirect RAM Addressing

For this to work in the example program, the RAM locations for the digits (the segment codes) must be in increasing sequential order as: LEFT_DIGIT, LEFT_DIGIT2, LEFT_DIGIT3, and RIGHT_DIGIT.

Let's see how this works in an improved version of the "DISPLAY routine as follows:

```
DISPLAY2:              ; Call this subroutine once every 5 milliseconds.
        CLRF       PORTB        ; Blank The Display
        CLRF       PORTC

        MOVF       DIGIT_INDEX,W    ; Get The Current Address
        MOVWF      FSR              ;    For Display
        MOVF       INDF,W           ; Get The Digit's Segment Codes
        MOVWF      PORTB            ; Display the Segments
        MOVF       DIGIT_POSITION,W    ; Get The Digit to Drive
        MOVWF      PORTC               ; Activate the Drive

        INCF       DIGIT_INDEX,F    ; For Next Time, Increment
                                    ; Address
        BCF        STATUS,C         ;    Shift to Next Digit to Drive
        RRF        DIGIT_POSITION,F
        BTFSS      STATUS,C         ; Check Limits
        RETURN

INITIALIZE_DISPLAY:
        MOVLW      LEFT_DIGIT       ; Put the ADDRESS of LEFT_
                                    ; DIGIT into W (Not the DATA)

        MOVWF      DIGIT_INDEX      ; Set-up with LEFT_DIGIT
        MOVLW      0x08             ; Set Digit Drive for Left Digit
        MOVWF      DIGIT_POSITION
        RETURN
```

At the start, the segments and digits were blanked (cleared) to prevent flickering when a new digit is set-up. Then we get the value in "DIGIT_INDEX", which is the RAM *address* of the digit's segment codes to be displayed, and put this into the "FSR" register. Using this RAM *address* in "FSR", we get the *data* at this address (the segment codes) using the "MOVF   INDF,W" instruction and send it to Port B. Next, we get the "DIGIT_POSITION" and send it to Port C to drive the digit transistor.

This completes the part of the "DISPLAY2" routine which uses indirect addressing. The rest of the routine must set up the next digit to be displayed when the routine is called the next time. This introduces two new instructions. The first is,

"INCF   f,d", which increments a register-file (RAM byte) and can store the result in the RAM or in the W register. There is also a, "DECF   f,d", instruction which decrements a register-file. Both the "INCF" and the "DECF" have the same operation format and both affect the "Z" (Zero) flag. The "C" and "DC" flags are not affected.

　　　The next new instruction is the, "RRF f,d", which does a bit-wise rotate-right on a register-file. There is also a, "RLF f,d", instruction which does a bit-wise rotate-left. The actions of these instructions are best seen in Figure 5-7. The only flag bit affected in both of these instructions is the "C" (Carry) flag.

Figure 5-7 Diagram of RLF and RRF Instructions
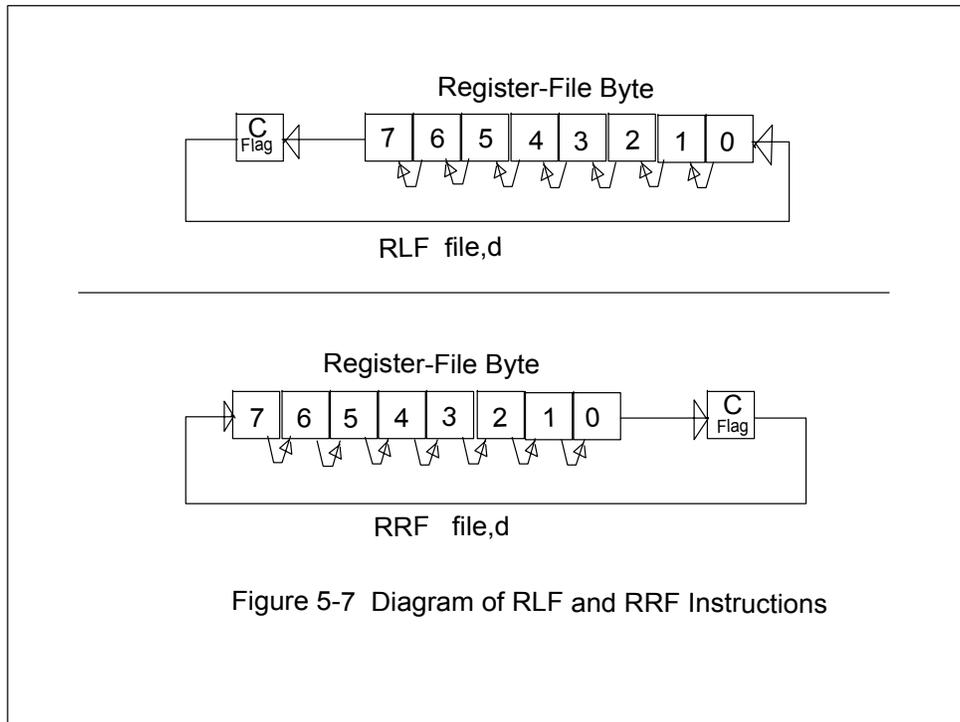
　　　To set-up the next digit for display, the "DIGIT_INDEX" is incremented to point to the next sequential RAM address. If that was "LEFT_DIGIT", its increment would be "LEFT_DIGIT2". The "DIGIT_POSITION" is then rotated right, after clearing the carry bit, to set the next digit-drive bit. The sequence of digit-drive bits on Port C looks like:

Bit 3	Bit 2	Bit 1	Bit 0	
1	0	0	0	---- LEFT_DIGIT
0	1	0	0	---- LEFT_DIGIT2
0	0	1	0	---- LEFT_DIGIT3
0	0	0	1	---- RIGHT_DIGIT

If the current digit-drive bit is as (0 0 0 1) and this gets rotated right, the "C" (Carry) flag will be set (=1). This indicates that the data must be re-initialized and set-up for the LEFT_DIGIT. The instruction, "MOVLW   LEFT_DIGIT", moves the RAM *address* into the W register and this is placed in the "DIGIT_INDEX" RAM byte.

5.7   Odds & Ends

One remaining point in using indirect addressing of RAM is the STATUS register bit, "IRP" (Bit 7), which selects which banks of RAM to use. Direct addressing of RAM used the STATUS bits, "RP1" and "RP0", to select among four banks of seven-bit-addressable RAM locations. Since indirect addressing uses 8-bit addresses in the "FSR" register, only one extra bit, "IRP", must be set-up to select pairs of RAM banks. It is as follows:

IRP = 0   for Bank 0 and Bank 1
IRP = 1   for Bank 2 and Bank 3.

In the security system, the KEY_SCAN routine is good for when only one key-press is needed. If the system requires multiple key-presses, such as when entering a multi-digit number, the software must check for the key to be released before looking for the next key to be pressed. This will avoid a "string" or "run" of a single key being interpreted as "several keys".

Once a key is pressed, released, and recognized as "valid", it must be decoded into a number or digit. This may require another look-up table. Suppose that a two-digit, decimal number is to be entered. The most-significant digit must be multiplied by ten and added to the least-significant digit to form the complete binary number. Or, suppose, that a three-digit, decimal number (less than 256) is to be displayed on the LED digits. How would a binary number be converted to a set of decimal digits?

To multiply a four-bit number by ten you could use a look-up table or you could multiply it by four, add it to itself to get a "multiply by five", and then multiply the result by two, to get a "multiply by ten". This is as follows:

```
MOVWF      TEMP1       ; Save W
MOVWF      TEMP2       ; Save W
BCF        STATUS,C    ; Prepare to Rotate
RLF        TEMP2,F     ; Multiply By two
```

```
RLF         TEMP2,W    ; Multiply By Two, Again (4*W)
ADDWF       TEMP1,F    ; Add W to (4*W) to Get  (5*W)
BCF         STATUS,C   ; Prepare to Rotate
RLF         TEMP1,W    ; Multiply By Two, To Get (10*W) in W
```

Converting from a binary byte to a three-digit, decimal number can be done with division or by repeated subtraction. For example, use a counting-loop and count the number of times that ten or one hundred can be subtracted from the number without going over. There are two subtraction instructions in the PIC. The first is, "SUBWF f,d", which means, "subtract W from the register-file (RAM)" or:

$$W = (file) - W$$
$$Or \ (file) = (file) - W.$$

The second is, "SUBLW k", which means, "Subtract W from the eight-bit value, k, and put the result in W". That is:

$$W = k - W.$$

In both of these, the STATUS flag bits "Z" (Zero), "C" (Carry), and "DC" (Digit Carry) are all affected.

In addition to the arithmetic instructions there are six logic instructions as follows:

ANDWF   f,d   Logical "AND" with register-file
ANDLW   k   Logical "AND" with eight-bit data

IORWF   f,d   Logical "OR" ("Inclusive OR") with register-file
IORLW   k   Logical "OR" ("Inclusive OR") with eight-bit data

XORWF   f,d   Logical "XOR" ("Exclusive OR") with register-file
XORLW   k   Logical "XOR" ("Exclusive OR") with eight-bit data.

Each of these affect only the "Z" (Zero) flag.

The logical "AND" function is used to force zeros (zero-bits) into data bytes. That is, it is used to mask off unwanted data bits. For example, suppose you wanted the lowest three bits of the W register and you wished to mask off the rest of the bits (make them zero). You could say, "ANDLW   0x07", to retain only the three lowest bits. If W contained 0xC6, doing the "AND" with 0x07 would give W = 0x06.

The logical "OR" function is used to force ones (one-bits) into data bytes. For example, suppose you wanted to put ones in the upper four bits of the W register. You could say, "IORLW   0xF0". If W contained 0xC3, doing this "OR" would give W = 0xF3.

The logical "XOR" function is useful for toggling bits "on" and "off". This could be used for making LEDs blink or flash. It is also useful for selectively complementing bits in memory or in the W register. For example, if you had W = 0x01 and you "XOR"ed it with 0x03, the result would be W = 0x02. If you "XOR"ed it again with 0x03, you would get W = 0x01. Two successive XORs cancel each other out.

Another logic instruction is, "COMF   f,d", which forms the one's complement of a register-file (RAM). The "COMF" instruction only affects the "Z" (Zero) STATUS flag. To form the one's complement of the W register, you could use, "XORLW   0xFF".

The last instruction to study in this chapter is, "SWAPF   f,d", which does a swapping of the upper and lower four-bits of a register-file (RAM). For example, if the location "TEMP" contained 0xA9, doing a "SWAPF   TEMP,F" would put 0x9A in "TEMP". "SWAPF" does not affect any flags.

The PIC has three more instructions which we will cover in Chapter 8. These instructions will not be covered, now, since they are special and more background is needed to understand them.

Adding or subtracting multi-byte numbers is more difficult in the PIC since there are no "add" or "subtract" instructions which include the "C" (Carry) STATUS flag at the *start* of the add/subtract. Other processors have an "add-with-carry" and a "subtract-with-borrow" instructions to make multi-byte arithmetic easier.

Suppose there are two, two-byte numbers to be added to get a two-byte result (sum). Let these RAM locations be defined as follows:

(IN_1_HIGH   IN_1_LOW)   (IN_2_HIGH   IN_2_LOW)

(SUM_HIGH   SUM_LOW)   (CARRY_HOLD).

The program to add these two-byte numbers is as follows:

```
        MOVF        IN_1_LOW,W        ; Load first number (low)
        ADDWF       IN_2_LOW,W        ; Add second number (low)
        MOVWF       SUM_LOW           ; Store Sum (low)
        CLRF        CARRY_HOLD        ; Reset Carry-Hold Byte
        BTFSC       STATUS,C          ; Test If Carry Set
        BSF         CARRY_HOLD,0      ; Set Carry-Hold if so
        MOVF        IN_1_HIGH,W       ; Load first number (high)
        ADDWF       IN_2_HIGH,W       ; Add second number (high)
        ADDWF       CARRY_HOLD,W      ; Add Carry to High Sum
        MOVWF       SUM_HIGH          ; Store Sum (high)
```

A double-byte subtraction is similar, but if the "C" (Carry) is set, a "one" is subtracted from the high-byte of the difference.

Another "odds & ends" point that needs to be covered is memory paging in the program memory. The "CALL" and "GOTO" instructions only have eleven (11) address bits, which gives a range of destinations only in a 2048 (2K) block of program memory. The PIC16F877 has a full program memory space of 8192 (8K). How can we get beyond the 2K limit?

The answer is to directly set or clear the upper-most bits in the high-half of the program-counter, "PCLATH", using the "BCF" and "BSF" instructions, prior to doing a "CALL" or "GOTO" instruction. These bits do *not* take effect *immediately upon* doing the "BCF" or "BSF" but are *delayed* or *postponed* until after the PIC gets the "CALL" or "GOTO" instruction to go to a higher memory beyond the 2K limit. In the case of a "CALL" instruction, the corresponding "RETURN" does not need to adjust the "PCLATH" bits since the full return-address is saved (Just do a "RETURN").

For example, consider the following program segment:

```
        ORG         0x0100
        BSF         PCLATH,4    ; Set Most Significant ADDR Bit
        BSF         PCLATH,3    ; Set 2nd Most Significant ADDR Bit
        CALL        HIGH_SUB

        ORG         0x1827
HIGH_SUB:

        RETURN
```

This would call the "HIGH_SUB" subroutine in the upper 2K block of the 8K program memory.

Just as there is the "BANKSEL" directive, there is also the "PAGESEL" directive. The "PAGESEL" directive simplifies the set-up of the PCLATH register prior to doing a GOTO or a CALL. Its syntax is, "PAGESEL program-address". For example, the above CALL to "HIGH_SUB" could be done more easily as:

```
        PAGESEL    HIGH_SUB
        CALL       HIGH_SUB
```

5.8   Using KEY_SCAN and DISPLAY Together

Let's see how KEY_SCAN and DISPLAY can be used together so that the user's keystrokes will appear in the LED digit display.

First, assume that the KEY_SCAN routine has been modified to de-bounce the keys and allow only one key at a time to be recognized. This will be shown in principle with the techniques of Chapter 6. Also, assume that a RAM location called "SKIP_CODE" has been filled with either 0x01 or 0xFF when the KEY_SCAN routine returns. As before, assume that the KEY_SCAN routine returns with the key-code number in the W register.

Let the following instructions be executed every five milliseconds and in this order:

```
CALL        KEY_SCAN
DECFSZ      SKIP_CODE,F
CALL        PROCESS_THE_KEY
CALL        DISPLAY
```

If the "SKIP_CODE" RAM byte contained 0xFF, then the "DECFSZ" would *not* skip and the "PROCESS_THE_KEY" routine would be called. The KEY_SCAN routine must do this only when there is a valid key-press. Otherwise, a value of 0x01 must be used in "SKIP_CODE" to skip over the "PROCESS_THE_KEY" routine when no key is pressed.

Since the DISPLAY routine displays only whatever is in the display-RAM bytes, the job of the "PROCESS_THE_KEY" routine would save the raw key-code, shift the display-RAM bytes to the left by one digit, convert the raw key-code to a display-code, and then fill in the new display-code on the right-hand digit.

This would be done as follows:

```
PROCESS_THE_KEY:
        MOVWF       SAVE_KEY   ; Save the raw key-code
        BSF         USER_FLAGS,KEY_IS_READY  ; Tell Program "Key"

        MOVF        LEFT_DIGIT2,W     ; Shift Over One Digit to the Left
        MOVWF       LEFT_DIGIT
        MOVF        LEFT_DIGIT3,W
        MOVWF       LEFT_DIGIT2
        MOVF        RIGHT_DIGIT,W
        MOVWF       LEFT_DIGIT3
```

```
        MOVF        SAVE_KEY,W
        SUBLW       D'10'               ; Check if = "*"
        BTFSS       STATUS,Z
        GOTO        NOT_STAR

        CLRF        RIGHT_DIGIT         ; Put Blanks in Right-Digit on "*"
        RETURN

NOT_STAR:
        MOVF        SAVE_KEY,W
        SUBLW       D'12'               ; Check if = "#"
        BTFSS       STATUS,Z
        GOTO        NOT_POUND

        CLRF        RIGHT_DIGIT         ; Put Blanks in Right-Digit on "#"
        RETURN

NOT_POUND:
        MOVF        SAVE_KEY,W
        SUBLW       D'11'               ; Check if = "0"
        BTFSS       STATUS,Z
        GOTO        NOT_ZERO

        CLRW
        CALL        GET_DISPLAY_CODE
        MOVWF       RIGHT_DIGIT
        RETURN

NOT_ZERO:
        MOVF        SAVE_KEY,W
        CALL        GET_DISPLAY_CODE
        MOVWF       RIGHT_DIGIT
        RETURN
```

(This program will do the trick to set-up the right display codes but it is not the best way to do it. Can you think of a better way?).

5.9   A Last Look at the Advanced Security System

The main body of the security system code would look like this:

```
        LIST  P=16F877
        INCLUDE "P16F877.INC"

        ORG         0x0000
        CALL        INITIALIZE

MAIN_LOOP:
        CALL        DELAY_FIVE_MILLISECONDS
        CALL        KEY_SCAN
        DECFSZ      SKIP_CODE,F
        CALL        PROCESS_THE_KEY
        CALL        DISPLAY
        CALL        SCROLL_MENU
        CALL        MENU_MODE
        GOTO        MAIN_LOOP

MENU_MODE:
        MOVF        MODE_CODE,W
        ANDLW       0x07                ; Restrict to eight values
        ADDWF       PCL,F               ; Start of "JUMP" Table
        GOTO        RESET
        GOTO        IDLE
        GOTO        SET_ZONES
        GOTO        ARM_HOME_AWAY
        GOTO        SET_ENTRY_CODES
        GOTO        CHECK_FOR_ACTIVE_ALARMS
        GOTO        SET_ALARM_CALL_POLICE
        GOTO        READY_TO_ARM
        ------- Body of Code -----------------------
        END
```

Most of the main-loop code runs the keypad and the display. The major modes of operation of the security system are controlled by the "MENU_MODE" routine. The configuration of a table look-up that contains only "GOTO" instructions is very common in assembly language programs. It is called a "Jump-Table". The RAM byte "MODE_CODE" selects what part of the program is active at any one time. Since there are only eight (8) codes that are used, as (0,1,2,…,7), the "ANDLW 0x07" instruction will restrict any code to this range. This is done for safety reasons so as not to over-index the table look-up.

Each section of code in the various "modes" would have its own messages and prompts for the user's inputs. There must be very little delay in each of the modes and they must all return to the main-loop. The techniques in Chapter 6 will show you how to do this.

This method of design keeps the body of the code short and simple. It is easy to understand both conceptually and practically.

5.10   Summary of Instructions and Concepts

1) The STATUS register contains the RP0 and RP1 bits which select the RAM banks in the direct addressing mode and the IRP bit which selects the RAM banks in the indirect addressing mode. It also contains the "Z", "DC", and "C" flags, which are set-up as the results of arithmetic/logic instructions and the "MOVF" instruction.
2) The BANKSEL directive manipulates the RP0 and RP1 bits of the STATUS register.
3) The "ADDWF   PCL,F" instruction is a "GOTO"-like instruction and is the basis of all table look-ups, including "Jump-Tables". The table-data must fit squarely within a 256-word block of program memory.
4) Indirect addressing of RAM allows for the manipulation of data arrays. The RAM address is placed in the "FSR" register and the instruction which is to use indirect addressing must reference the memory pointed-to by the "FSR" by using a reference to the "INDF" register.
5) The "CALL" and "GOTO" instructions are limited to a 2 K block of program memory, in themselves. Bits "4" and "3" of the "PCLATH" register must be manipulated before doing a "CALL" or "GOTO" whose target address is beyond the current 2 K block. These bits do not take immediate effect when they are set-up but are postponed until after the "CALL" or "GOTO" is executed. The "PAGESEL" directive simplifies this process.
6) A "RETURN" or a "RETLW" instruction does not need to manipulate the "PCLATH" register bits prior to returning from the subroutine.
7) An "Add" or "Subtract" instruction will generate a carry-bit ("C") but there is no way to include the carry-bit at the start of the "Add/Subtract" instruction.
8) If a CALL is made to an "ADDWF   PCL,F" table look-up subroutine which is not in the same 256-word block of memory as the "CALL", the high-half of the subroutine's address must be loaded into the PCLATH register before the "CALL" is made. (See pages 45 and 46.)
9) The following instructions were introduced in Chapter 5:
    a) RETLW   data
    b) ADDLW   data
    c) ADDWF   file,destination
    d) CLRF   file

e) CLRW
f) INCF   file,destination
g) DECF   file,destination
h) RRF   file,destination
i) RLF   file,destination
j) SUBLW   data
k) SUBWF   file,destination
l) ANDWF   file,destination
m) ANDLW   data
n) IORLW   data
o) IORWF   file,destination
p) XORLW   data
q) XORWF   file,destination
r) COMF   file,destination
s) SWAPF   file,destination

10) Three remaining instructions have not yet been introduced.

    This concludes Chapter 5. A complete list of all of the PIC16F877's instructions and how to use them is contained in Appendix A.

Chapter 6:   Fundamental ESP Techniques

6.0   Chapter Summary

Section 6.2 covers software readability. Section 6.3 covers software maintainability. Section 6.4 discusses the most general fundamentals of embedded systems software. Section 6.5 discusses the background routine. Section 6.6 covers the theory of the watch-dog timer. Section 6.7 discusses event driven software. Section 6.8 covers the theory of processor interrupts. Section 6.9 discusses slow inputs and outputs. Section 6.10 covers software time measurement techniques. Section 6.11 discusses hashing techniques. Section 6.12 covers the software methods of waveform encoding. Section 6.13 covers waveform decoding techniques. Section 6.14 discusses the fundamental tradeoffs of time, program memory, and RAM and how these affect program size and execution speed. Section 6.15 discusses "ROM states" which are useful for high speed operation. Section 6.16 discusses the limitations of C/C++ in low end embedded systems programming.

6.1   Introduction

Programming for embedded applications involves more than just programming in the traditional sense. Embedded systems programming embodies total systems design. The programming part requires an active understanding of the electronics and mechanics of the system.

Even so, programming for embedded systems has a style, technique, art, and science of its own. It is always true that each individual engineer and programmer has a style of programming that is uniquely his or hers, but there are sets of common practices in the industry which unify the goals of embedded systems programming in general. This chapter will give guidelines and the common practices for embedded systems programming in a general sense.

6.2   Software Readability

It is generally true that all programs need sound documentation but this is especially true for embedded systems programming since there is such a strong connection to the hardware. Often the documentation should explain the hardware connections that parallel the software constructs. Such things as gates, memories, LEDs, switches, motors, and other devices have wiring conventions of their own which become important when they are interfaced to the PIC. The documentation should clearly state how these devices are to be used and which parts of the PIC will control them.

Other aspects of documentation are similar to traditional programming in that it should clearly define the software operations that are used and fully explain the programmer's intent in using combinations of instructions to accomplish a given task.

Software readability should give another programmer a solid idea of what the program is about, without ego getting in the way. This is another aspect of writing with the intent of being understood. This is also important when you, as the programmer, leave the program alone for a few years and then come back to it when you need it. Your documentation should give you a clear picture of your program even after you are away from it for years. You should state all of the "hidden" secrets in the documentation without trying to carry them in your head. The documentation should be complete in every way.

6.3   Software Maintainability

Maintainability is the art of building complex software systems out of simple components that are easy to modify and make the software easily adaptable to new, but similar, applications. Products and systems rarely "stand alone" after they are created and are dynamic. New markets and new applications will come into being and the software systems must meet these future uses without having to be completely redesigned. Maintainability is the art of designing for the future as well as the present.

6.4   Software Fundamentals

Embedded software should be modular and should have a simple "main-loop". The main-loop should be "small" or at most two pages long. It should call subroutines as the functional, modular blocks to perform each major task. Each task should be free of overhead and its subroutine should do its tasks in as simple a way as possible. The security system software example at the end of Chapter 5 is a good example of a simple main-loop and a simple and maintainable software structure.

Subroutines should also be "small" if possible. They should have multiple entry-points and serve multiple purposes. This is contrary to high-level programming practices which emphasize "one entry-point, one function" subroutines. It may be necessary, due to memory constraints, to squeeze as many functions as possible into the smallest space.

Subroutines that work as modules must be careful when the subroutines they call take too long to run. There must be no or, very little, time delays or bottlenecks that would interfere with the timely operation of the system. Generally, it is wise to avoid recursive subroutine calls.

Another good programming practice is "information-hiding". Subroutines should be classed into layers of low-level, medium-level, and high-level functions. This does not mean that they should be nested this way. It means that information should be

processed in separate layers. For example, in a security system, a low-level function would read the raw data, such as, switch closures and voltages, set LEDs and other bit-level outputs and put the results in RAM. A medium-level function would set-up alarm conditions from the low-level data and put its results in RAM. The high-level functions would, then, perform the "command-and-control" tasks such as setting and reporting an alarm. Since each level puts its intermediate data into RAM, there is no need to nest these routines. Information and processes can be "hidden" from the upper levels and make the overall program more compartmentalized and make it more maintainable.

6.5   The Background Routine

One large subroutine should be called from the main-loop to do housekeeping tasks. This is called the "background" routine. It consists of common tasks which can remain invisible to the rest of the program, such as keeping a time-of-day clock and calendar, refreshing port settings, servicing slow inputs and outputs, sampling a keypad, running an LED digit display, and, in the case of a robot, running stepper motors.

6.6   The Watch-Dog Timer

Microprocessor and microcontroller systems are not as stable as other digital hardware due to the fact that they are more complex and they work with flexible software. It is easy for such a system to get "locked-up" in infinite software loops due to a noise spike or unanticipated data. As a programmer, you try to account for all of the possible input data that can come into your system and all of the possible results of the calculations on that data, but in reality, you can't. An unexpected situation may arise and put the processor into an infinite software loop.

The watch-dog timer is typically a hardware device external to the processor that has the power to reset the processor over-and over if the watch-dog timer itself is not kept in reset by the processor's software. Its job is to break the software out of any infinite software loops if they should occur. The software must be active all of the time and it must not have any substantial delays. It must reset the watch-dog timer on a regular basis and, if it doesn't, the watch-dog will keep resetting the processor over-and-over.

The PIC16F877 has a built-in watch-dog timer which is built into the chip, but it operates or can operate independently from the other PIC hardware. It can be enabled and disabled from software with an adjustable time-out time, and the PIC has a special instruction, "CLRWDT", which resets the watch-dog timer. The specific details of its use will be covered in Chapter 8 (PIC Peripherals).

Other processors may not have a built-in watch-dog timer and it may be necessary to get a watch-dog timer chip to use in the system. An industrial product cannot do without one! It is a *MUST!* One watch-dog timer chip available from Maxim is the MAX690CPA. It is also possible to make your own watch-dog timer from all-digital

gates as shown in Figure 6-1. The microprocessor's port-pin drives the clock input of a "one-and-only-one" pulse circuit which in turn drives the counter's reset line. If there are no watch-dog resets, the counter will reset the microprocessor. The clock input of the pulse circuit is an edge-triggered input. This is critical. The watch-dog timer must not be reset with a level signal so that it cannot be accidentally left in "reset". The software must always toggle the port-pin line.

Figure 6-1 "All Digital" Watch-Dog Timer Circuit
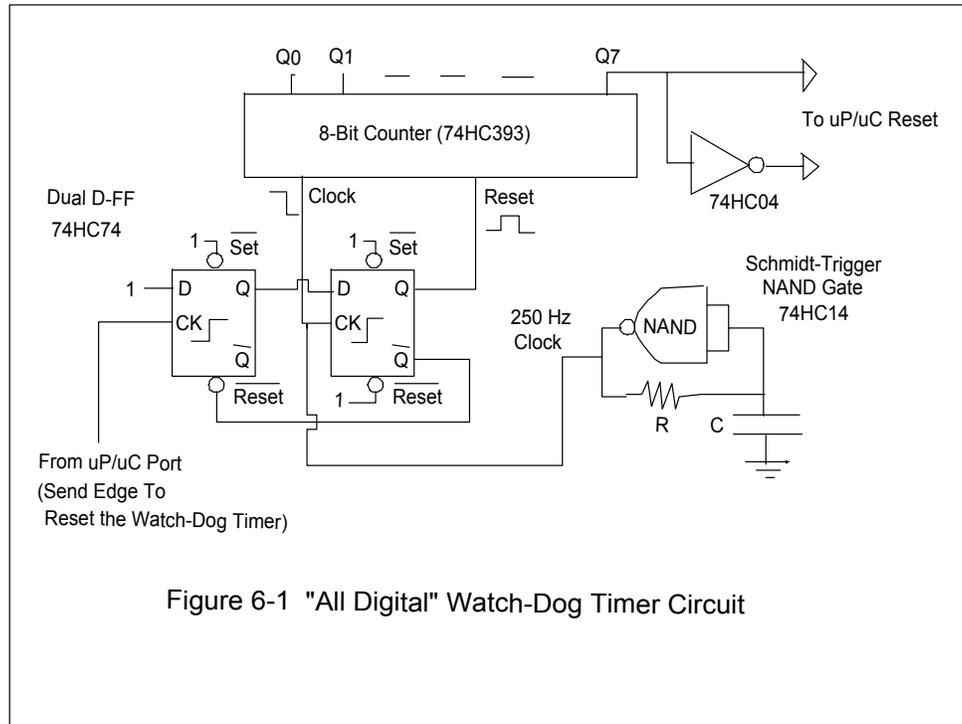
The command or command sequence to reset the watch-dog timer should be placed once-and-only-once in a program. It should be placed in the main-loop since the software must always come back to the main-loop. The time-out time of the watch-dog timer should be about one second for most applications. A good rule of thumb for the watch-dog's period is about ten times the longest software delay in the main-loop.

6.7   Event-Driven Software

Event-driven software is the central unifying concept in embedded systems programming. It is the main idea which separates ESP from traditional programming.

Event-driven software uses flags and counters in RAM to mark the progress of an input. When there are changes to the input, the flags and counters are updated to reflect

these changes. Event-driven software can control many delays and many arbitrary processes at once without waiting in loops. There are no bottlenecks in event-driven code.

An example program will show the structure and function of event-driven code. This example is for de-bouncing a push-button switch. This software routine looks for the contact and the release of the switch before declaring that the data is valid. The flowchart for this process is shown in Figure 6-2.
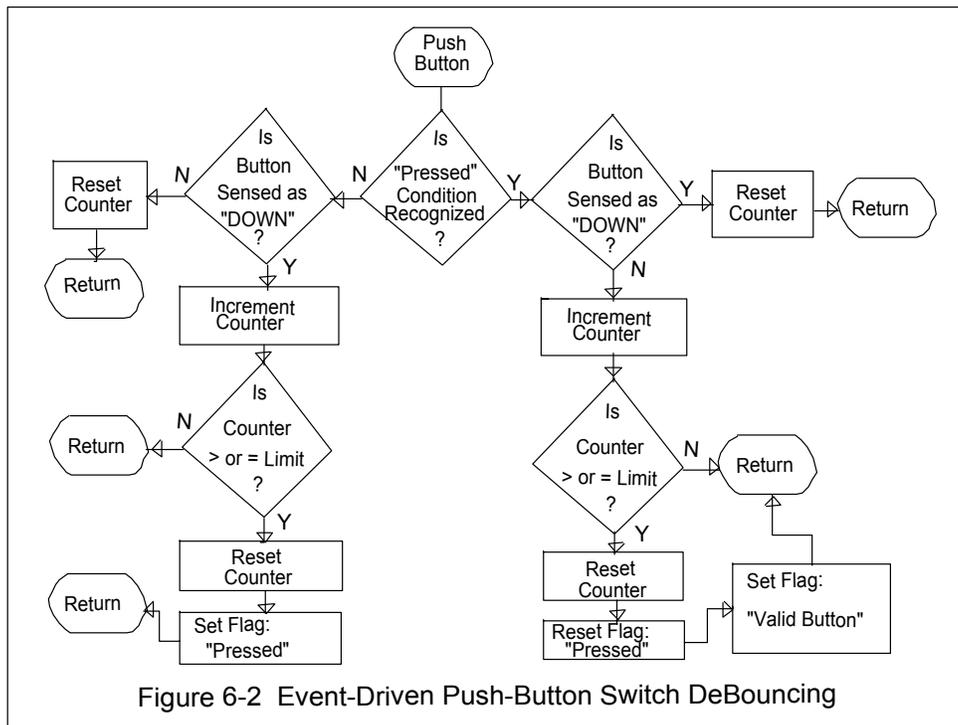
Figure 6-2 Event-Driven Push-Button Switch DeBouncing

Assume that the following routine is called every five milliseconds at a time:

```
PUSH_BUTTON:
        BTFSC       FLAGS,PRESSED  ; Test if the button is de-bounced
                                   ;    AND "ON" (Pushed)
        GOTO        WAIT_FOR_RELEASE

        BTFSC       PORTB,PUSH_BTN        ; Test the raw button state
        GOTO        WAIT_FOR_SET    ; =  Pushed ("ON")
        GOTO        RESET_COUNTER; =  "OFF", Reset De-Bounce
                                   ;    Timer
```

63

```
WAIT_FOR_SET:
        DECFSZ      WAIT_COUNTER,F ; Delay for De-Bounce
        RETURN
        BSF         FLAGS,PRESSED   ; Set "De-Bounced, ON"
        GOTO        RESET_COUNTER

WAIT_FOR_RELEASE:
        BTFSC       PORTB,PUSH_BTN        ; Test raw button state
        GOTO        RESET_COUNTER ; =  Pushed ("ON")

        DECFSZ      WAIT_COUNTER,F ; = "OFF", wait for De-Bounce
        RETURN
        BCF         FLAGS,PRESSED   ; De-Bounce Cycle Done!
        BSF         FLAGS,DATA_READY  ; Done! Data is Ready!
RESET_COUNTER:
        MOVLW       WAIT_TIME            ; Fill Initial Value of Counter
        MOVWF       WAIT_COUNTER
        RETURN
```

At the start of the routine, the software checks if the push-button has been pressed and has passed the de-bouncing in the "ON" state. If it has, the software goes to look for the release of the push-button. If not, the raw state of the switch is checked. If it is pressed ("ON"), the counter loop is run to satisfy the de-bouncing condition. If it is not pressed, the counter is reset. If the "ON" state de-bouncing is satisfied, the "PRESSED" flag is set, the counter is reset, and the routine will then look for the de-bouncing in the released state. Once the release is complete, the "DATA_READY" flag is set to register a complete "press and release" of the push-button.

At no time does this routine do any waiting. It is called, it checks for conditions, it performs its actions, and in each case, it exits.

The time the routine takes to run is negligible and many similar routines can be run with the illusion of being simultaneous.

Event-driven software can also be used with internal events, such as for flashing LEDs and for generating waveforms.

An application that is event-driven will spend 99 percent of its time doing nothing! It acts only on the change of internal and external events and has no waiting loops. Event-driven techniques make working with the watch-dog timer easy and the resulting program is very maintainable!

6.8   Interrupts

Interrupts are a hardware process whereby a piece of hardware can cause the processor to execute a subroutine at a special address. The process makes the CPU drop whatever it is doing at the time and run the subroutine. The PIC16F877 has a total of 14 interrupts related to its peripheral functions. The mechanics of how to use them will be shown in Chapter 8 (PIC Peripherals).

Many processors (including the PIC) allow the interrupts to be "enabled" and "disabled" under software control. In some processors, there are interrupts which cannot be disabled and will respond "knee-jerk" fashion at all times. Interrupts must be used with extreme care and can cause many severe software/hardware problems if they are abused.

Why are interrupts used? An interrupt is the fastest way to get the processor's attention to work on an urgent problem. It should be used only in this case. If a timer or counter is set to run as a time-base for the system (the PIC can also do this), an interrupt is an ideal way to respond to the timer/counter and re-initialize it. If there is a peripheral that must be serviced very quickly and there is no other way to do it, then an interrupt must be used.

I regard interrupts as "the method of last resort" in any programming situation. The potential for abuse when using interrupts is very severe. The problems they can cause can be extremely difficult to track-down and solve. If you can avoid using interrupts, I strongly advise that you do not use them. Event-driven techniques are far easier to work with in every way.

One particularly nasty abuse of interrupts is to use them for keypads and switch de-bouncing. This is *the* most inappropriate use of interrupts. There is no way that a key needs an interrupt to process it, since it works at *millisecond* speeds and a human key operator is much slower.

Interrupts and their service-subroutines should *NEVER* be used to reset the watch-dog timer. The interrupt is a "knee-jerk" response and a subroutine. If the service-subroutine is called from an infinity loop, it will return to that infinity loop upon its return from the interrupt, thereby defeating the whole purpose of having the watch-dog timer in the first place!

6.9   Slow Inputs and Outputs

As a general rule of thumb, an input or an output is "slow" if changes to or from it occur in one millisecond or more time. Such things as DIP switches, push-button switches, telephone-ringing sensors, LEDs, multiplexed LED digits, relays, motors, temperature sensors, and heating elements are considered "slow" I/O.

Slow inputs and outputs should be sampled in one, and only one, place in the software, and, preferably, should be done in the background routine.

All slow inputs should be buffered, processed, and sorted by special routines before their information is made available to the rest of the program. All slow outputs should do the same before being presented to the outside world.

There are several reasons for doing this. If there were design changes to the hardware where the input or output port-pins are swapped, only these special routines would have to be changed instead of the whole program! These routines could descramble the inputs and outputs so that they are in uniform order (e.g., Input-0 = Bit-0, Input-1 = Bit-1, …). Some inputs and outputs may be active-low or active-high. These routines can convert every input and output to an "active-high" state. This is another example of layered software and information hiding.

After the slow inputs are buffered, processed, sorted, and made uniform, the results should be put into RAM. Any time that the program needs the information, it can check these RAM bytes. The program should never check the bit directly at the port. The slow outputs, in a similar way, should be buffered, processed, sorted, and made uniform from a RAM location when it is to be sent to the output port. Changing a bit in these RAM locations is equivalent to sending it out to a port. Only the background routine and the special input/output routines should interface with the port directly.

6.10   Software Time Measurement

Although the PIC and other processors have hardware for measuring time-delays and pulse-widths, it is still useful to have software techniques for doing these things, in that they can be made more fault-tolerant than the hardware versions.

A simple software time-measurement loop is as follows:

```
TIME_MEASURE:
        CLRW                            ; Reset Time-Measure Counter
TIME_LOOP:
        BTFSC        PORTB,EVENT_SENSE  ; Look For The Event
        RETURN
        MOVWF        TEMP
        INCF         TEMP,W       ; Increment (W) Time-Measure Counter
        BTFSS        STATUS,Z     ; Check for End of Loop
        GOTO         TIME_LOOP
        RETLW        0xFF
```

The time-measurement is returned in the W register. The total loop-time in this routine is seven instruction cycles.

A faster and higher resolution software time-measurement process is as follows:

```
TIME_MEASURE:
        BTFSC       PORTB,EVENT_SENSE  ; Look for Event
        RETLW       0
        BTFSC       PORTB,EVENT_SENSE
        RETLW       1
        BTFSC       PORTB,EVENT_SENSE
        RETLW       2
---- And So On For The Desired Time Length -----
```

This "non-loop" has a resolution of only two instruction cycles instead of seven in the first loop. It takes up much more space, but it runs very quickly and has very high time resolution.

6.11   Hashing

Hashing is a table-search technique that uses the input data directly as the address or index of the table item to find. Actually, we have already seen hashing in action through the look-up tables in Chapter 5. It is not only fast, but it also runs in uniform-time (i.e., the time it takes to find a table entry is the same for each table entry). The security system at the end of Chapter 5 also introduced us to "jump tables". This is a very important concept since it gives you the ability to "GOTO" a variable address in program memory.

Conversion tables for converting from ASCII to seven-segment codes are very common in all programs. Also, rapid scrambling or de-scrambling of data can be done in these tables. The key is to use the data as the index of the table. Whole waveforms can be stored in these tables for the production of music and other sounds.

An improved jump table is as follows:

```
DO_JUMPS:
        MOVWF       TEMP            ; Multiply (W) by three
        BCF         STATUS,C
        RLF         TEMP,F
        ADDWF       TEMP,W
        ADDWF       PCL,F           ; Index the Jump Table
        BCF         PCLATH,4    ; (We could use PAGESEL here)
        BSF         PCLATH,3
        GOTO        SOME_HIGH_JUMP1
        BSF         PCLATH,4
        BSF         PCLATH,3
        GOTO        SOME_HIGH_JUMP2
----- And So On For More Long Jumps -----
```

This allows you to do "GOTO"s to any memory location.

6.12   Waveform Encoding

In some applications a microprocessor or a microcontroller may need to be used as a modem and must generate a coded waveform. This is an excellent example of where software can be used to mimic hardware. If the waveform is a "burst", or, short packet of a bit-stream, the routine to do it can be a short loop. If the waveform is to be a continuous, running stream of data, the routine must use event-driven techniques.

For example, consider a Frequency-Shift-Keying (FSK) burst encoder that uses two frequencies as 2400 Hz and 2250Hz for logic "1" and logic "0", respectively. Assume that the data rate is 150 Baud and that the data structure is: A preamble of ten "1"s, a "0", twenty-five (25) data bits, and an end bit, for a total of 37 bits to send. In this timing a "1" will be sent as 16 cycles of 2400 Hz and a "0" will be sent as 15 cycles of 2250 Hz. This time delay between bits is not critical but the frequency stability *is* critical and is set as a half-cycle as a "delay then complement output line state". This gives 32 half-cycles of 2400 Hz and 30 half-cycles of 2250 Hz.

The following routine will do the above FSK:

```
FSK:
        MOVLW       DATA_BIT_ARRAY          ; Get Address of Data
        MOVWF       FSR                     ; Set-Up Indirect ADDR
SEND_2400:
        MOVLW       32      ; Do 32 Half-cycles of 2400 Hz
        MOVWF       COUNT
LOOP_2400:
        CALL        DELAY_SEND_2400         ; Do Half-Cycle
        DECFSZ      COUNT,F
        GOTO        LOOP_2400
        MOVF        INDF,W      ; Get Next Bit To Send
        INCF        FSR,F       ; Point To Next Bit To Send
        ADDWF       PCL,F       ; Look-Up Table: Do Next Bit
        GOTO        SEND_2250 ; 0 = Send Zero Bit
        GOTO        SEND_2400 ; 1 = Send One Bit
        RETURN                  ; 2 = DONE, Return
        RETURN                  ; 3 = DONE, Return

SEND_2250:
        MOVLW       30      ; Do 30 Half-Cycles of 2250 Hz
        MOVWF       COUNT
LOOP_2250:
        CALL        DELAY_SEND_2250         ; Do Half-Cycle
        DECFSZ      COUNT,F
```

```
        GOTO        LOOP_2250
        MOVF        INDF,W      ; Get Next Bit to Send
        INCF        FSR,F       ; Point To Next Bit To Send
        ADDWF       PCL,F       ; Look-Up Table: Do Next Bit
        GOTO        SEND_2250   ; 0 = Send Zero Bit
        GOTO        SEND_2400   ; 1 = Send One Bit
        RETURN                  ; 2 = DONE, Return
        RETURN                  ; 3 = DONE, Return
```

The Data Array would contain the sequence of bits starting from the preamble, and going to the end-bit. After the end-bit would be a code of "2" or "3" to stop and return.

Another burst-type data transmission technique is the Manchester code. This is best seen in Figure 6-3. This works as a kind of "differential phase-shift keying" (DPSK) on one cycle of a square wave. It is a polar waveform and it can only be used where the transmission medium can support edges and pulse-widths without distortion. Its timing in all of its parts is very critical and only a strict ratio of pulse widths as 2:1 are allowed. Assume for the following example that the data structure is: A preamble of ten "1"s, a "0", and sixteen (16) data bits. The Manchester Code waveform routine is as follows:

```
TEMP:           EQU  0x20
FLAGS:          EQU  0x21
DATA_ARRAY:     EQU  0x30

OUTPUT_BIT:     EQU  0        ; PORTC, Bit 0
SAMPLE_FLAG:    EQU  1        ; Mark Half-Cycle Where To Sample
                             ;   Data
INHIBIT_FLAG:   EQU  0        ; If Set, Inhibit State-Complement

MANCHESTER_WAVE:
        BCF         PORTC,OUTPUT_BIT      ; Reset Output Bit
        BSF         FLAGS,SAMPLE_FLAG     ; Sample on Odd Cycles
        BCF         FLAGS,INHIBIT_FLAG    ; Assume For Start,
                                         ;   Preamble
        MOVLW       DATA_ARRAY
        MOVWF       FSR                   ; Indirect ADDR

MANCHESTER_LOOP:
        CALL        DELAY                 ; Delay For Half-Cycle
        CLRF        TEMP                  ; Set-Up TEMP with
        BCF         TEMP,OUTPUT_BIT       ; Complement of INHIBIT
        BTFSS       FLAGS,INHIBIT_FLAG    ; In Position of
        BSF         TEMP,OUTPUT_BIT       ; Output Bit of PORTC
        BCF         FLAGS,INHIBIT_FLAG
```

```
        MOVF        TEMP,W                      ; Then, XOR This Bit
        XORWF       PORTC,F                     ; With Output Bit

        BTFSS       FLAGS,SAMPLE_FLAG    ; See If Sample
        GOTO        DELAY_COMPENSATION; No, Do Delay Comp
        BCF         FLAGS,SAMPLE_FLAG    ; Do Sample Now
        MOVF        INDF,W                      ; Get Data To Send
        BCF         STATUS,C
        MOVWF       TEMP                        ; Multiply By Two
        ADDWF       TEMP,W
        ADDWF       PCL,F                       ; Look-Up Table
        BCF         FLAGS,INHIBIT_FLAG   ; 0 = Send "Same" Data
        GOTO        MANCHESTER_LOOP
        BSF         FLAGS,INHIBIT_FLAG   ; 1 = Change Phase, Data
        GOTO        MANCHESTER_LOOP
        RETURN                                  ; 2 = 3 = DONE, Return
        RETURN
        RETURN
        RETURN

DELAY_COMPENSATION:
        BCF         FLAGS,INHIBIT_FLAG   ; No Phase Change
        BSF         FLAGS,SAMPLE_FLAG    ; Sample Next Time
        NOP
        NOP
        NOP
        NOP
        GOTO        MANCHESTER_LOOP
```

Whenever there is a change of data, the INHIBIT flag performs the phase-change by inhibiting the state-change on Port C, Bit zero.

Also notice that both the FSK and the Manchester encoders use one RAM byte per bit to send which is wasteful of RAM. Usually single bits in a small number of RAM bytes are used for the bits to send. Can you think of some schemes to use indirect addressing and use single bits in each byte? It could be "destructive" in that the byte got rotated and filled with zeros or it could be "non-destructive" and just sense the bits as they are. Try to think of several ways to do this.
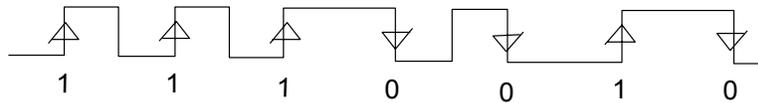
Pulse-Widths are of a Uniform Square Wave (Ratio 2:1)

Rising-Edges are Logic One (=1)
Falling-Edges are Logic Zero (=0)   Only Where Shown

Transitions Occur as a Double-Pulse-Width

Figure 6-3 Manchester Code Waveform

6.13   Waveform Decoding

Waveform decoding is the process of recovering data from an encoded, modulated waveform. The FSK signal in the previous section can be decoded with a phase-locked loop (PLL), where the control-voltage feeds the PIC's analog-to-digital converter input. Samples on the ADC are taken several times over the space of one bit to eliminate noise.

The Manchester Coded waveform can be decoded by following the highs and lows of the digital waveform. This requires getting into synchronization with the wave, measuring the pulse-widths, counting the pulses, and recovering the data by judging the phase-reversals. This process is shown in detail in Figure 6-4.

Figure 6-4 Decoding the Manchester Waveform
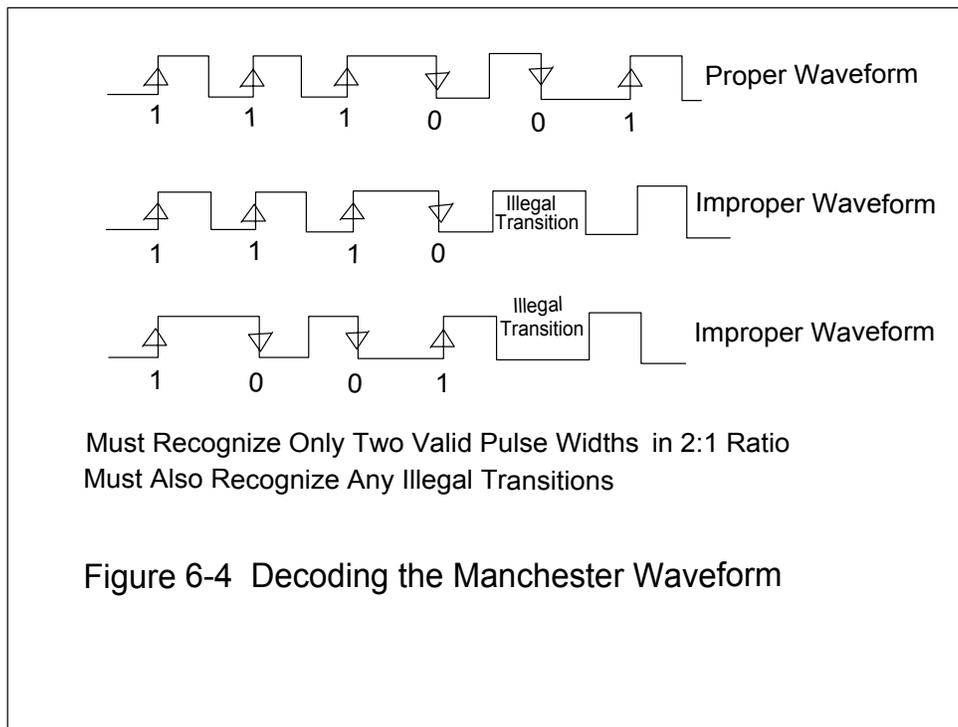
Both the FSK and Manchester data recovery processes must use hashing to place the data bits that are decoded.

A detailed example program of the Manchester decoding process is as follows:

```
MANCHESTER_DECODE:
        CALL        TIME_LOW            ; Start of Synchronization
        ADDLW       0                   ; See If W = Zero
        BTFSS       STATUS,Z
        GOTO        DO_HIGH_MEAS   ; Was In LOW, Look At Full HIGH
        CALL        TIME_HIGH          ; Was In HIGH, Complete HIGH
        CALL        TIME_LOW           ; Do Full LOW

DO_HIGH_MEAS:
        CALL        TIME_HIGH  ; Measure HIGH Pulse W = Count
        MOVWF       TIME
        CALL        TIME_LOW            ; Measure LOW Pulse W = Count
        ADDWF       TIME,F              ; Average These Counts
        RRF         TIME,W
        MOVWF       SHORT_BASE     ; Save SHORT Pulse Width
```

```
        ADDWF       SHORT_BASE,W    ; Mult By Two
        MOVWF       LONG_BASE       ; Save as LONG Pulse Width

        MOVF        SHORT_BASE,W    ; Form Threshold =
        ADDWF       LONG_BASE,W     ; Average(SHORT,LONG)
        MOVWF       THRESH
        RRF         THRESH,F

        MOVLW       NINE            ; Track The Preamble Pulses (9)
        MOVWF       PREAMBLE_CNT
PREAMBLE_LOOP:
        CALL        TIME_HIGH  ; Measure HIGH Pulse W = Count
        CALL        SORT_TIMES      ; Classify Count
        ADDWF       PCL,F
        GOTO        STILL_PREAMBLE ; 0 = SHORT, Still in PREAMBLE
        GOTO        GET_THE_DATA   ; 1 = LONG, Start of Data
        RETLW       2               ; 2 = ERROR in Timing

STILL_PREAMBLE:
        CALL        TIME_LOW        ; Measure LOW Pulse W = Count
        CALL        SORT_TIMES      ; Classify Count
        ADDWF       PCL,F
        GOTO        DEC_PREAMBLE   ; 0 = SHORT, Do Loop DEC
        RETLW       1               ; 1 = LONG, Polarity ERROR
        RETLW       2               ; 2 = ERROR in Timing

DEC_PREAMBLE:
        DECFSZ      PREAMBLE_CNT,F ; Decrement Counter
        GOTO        PREAMBLE_LOOP
        RETLW       3               ; ERROR Time-Out
GET_THE_DATA:
        MOVLW       N_DATA_BITS     ; Count Number Of Data Bits
        MOVWF       BIT_COUNT

IN_A_ZERO:
        CALL        TIME_LOW        ; Measure LOW Pulse W = Count
        CALL        SORT_TIMES      ; Classify Count
        ADDWF       PCL,F
        GOTO        STILL_IN_ZERO   ; 0 = SHORT, Still In a ZERO
        GOTO        TRANSIT_TO_ONE ; 1 = LONG, Go To ONE LOOP
        RETLW       2               ; 2 = ERROR in Timing

STILL_IN_ZERO:
        CALL        PLACE_ZERO_BIT ; Hash Store Data Bit = ZERO
        CALL        TIME_HIGH  ; Measure HIGH Pulse W = Count
        CALL        SORT_TIMES      ; Classify Count
```

```
        ADDWF       PCL,F
        GOTO        DO_DEC_ZERO     ; 0 = SHORT, Dec Zero Loop
        RETLW       1               ; 1 = LONG, ERROR Pulse
        RETLW       2               ; 2 = ERROR in Timing


TRANSIT_TO_ZERO:
        CALL        PLACE_ZERO_BIT ; Hash Store Data Bit = ZERO


DO_DEC_ZERO:
        DECFSZ      BIT_COUNT,F     ; Decrement Bit Counter
        GOTO        IN_A_ZERO
        RETLW       0               ; DONE, OK


IN_A_ONE:
        CALL        TIME_HIGH  ; Measure HIGH Pulse W = Count
        CALL        SORT_TIMES      ; Classify Count
        ADDWF       PCL,F
        GOTO        STILL_IN_ONE    ; 0 = SHORT, Still in a ONE
        GOTO        TRANSIT_TO_ZERO ; 1 = LONG, Data is a ZERO
        RETLW       2               ; 2 = ERROR in Timing


STILL_IN_ONE:
        CALL        PLACE_ONE_BIT   ; Hash Store Data Bit = ONE
        CALL        TIME_LOW        ; Measure LOW Pulse W = Count
        CALL        SORT_TIMES      ; Classify Count
        ADDWF       PCL,F
        GOTO        DO_DEC_ONE      ; 0 = SHORT, Do Dec Loop
        RETLW       1               ; 1 = LONG, ERROR Pulse
        RETLW       2               ; 2 = ERROR in Timing


TRANSIT_TO_ONE:
        CALL        PLACE_ONE_BIT   ; Hash Store Data Bit = ONE


DO_DEC_ONE:
        DECFSZ      BIT_COUNT,F     ; Decrement Bit Counter
        GOTO        IN_A_ONE
        RETLW       0               ; DONE, OK
```

This routine calls several subroutines to measure time, classify the times, and place the data bits into RAM. The time measuring routines, TIME_LOW and TIME_HIGH, work just like the seven-cycle loop of the "Time Measurement" section. The "TIME_LOW" routine returns when the input is sensed "high", while the "TIME_HIGH" routine returns when the input is sensed "low". The data placement routines work by hashing the bit counter. That is, the bit counter is multiplied by two and looked-up in a table with "BCF"'s or "BSF"'s and a "RETURN". The time classifier compares the time against the threshold and then tests the time as "Short" or "Long" to

within an absolute value difference of two using the "SORT_TIMES" routine. Then it returns with codes for, "Short", "Long", and "Error".

6.14   RAM, ROM, and Time Tradeoffs

There are three major tradeoffs of memory and time in embedded systems programming. These are:

1) "RAM vs. ROM"
2) "ROM vs. Time"
3) "Run-In-ROM vs. Run-In-RAM"

In the "RAM vs. ROM" tradeoff, the usage of RAM and ROM tends to vary inversely. For example, a calculation subroutine may use more RAM to figure intermediate values or it may use more ROM in the form of a look-up table. Event-driven programs tend to be RAM-intensive since they use flags and counters to track input and output signals.

In the "ROM vs. Time" tradeoff, the size of the ROM code varies inversely with the code-speed execution. For example, the short-loop time measurement scheme used only seven cycles of time and eight ROM words. The "non-loop" scheme could use up to 512 ROM words but had a resolution of only two cycles. The short-loop could measure times as long as 1.75 milliseconds but the "non-loop" could measure only half a millisecond.

In the "Run-In-ROM vs. Run-In-RAM" tradeoff, a processor that has the capability of running its programs from RAM has a speed and versatility advantage over processors that only run their programs in ROM. For example, if a high-speed waveform is to be sent out, the processor that only runs in ROM may not be fast enough to run it. If a processor that can run its programs in RAM uses the ROM program to synthesize a short subroutine in RAM and call it from ROM, it can run at the highest possible speeds! The PIC cannot do this exactly, but it does have field-programmable data and program memories that can exploit some of these tricks. (These will be shown in Chapter 8.)

6.15   ROM States

The idea of a "ROM State" is to make a program that "remembers" previous data values by being in different parts of the program in the program memory or ROM. That is, where you are in the program is a reflection of what the previous data was. This technique is used only for "high-speed" or "high-efficiency" applications since "ROM State" subroutines get very large for complex tasks.

An example application which uses ROM States for its most efficient coding is a "median filter", which will be discussed in full in Chapter 10 (DSP Fundamentals). The idea of a median filter is to keep five samples of data in the order they were received,

transfer them to a second data array, sort the second data array into order, and send out the middle point (the median) as the "output".

The time-optimal way to do this using ROM States is to keep part of the previous order in the second array and, when a new sample comes in, fit that sample into the mostly-sorted array so that the whole array is sorted again. First, find the oldest sample (which will be discarded) in the sorted array and mark its position in the array by doing one of five "GOTO" instructions. At the target addresses of each of these "GOTO"'s the new sample is tested against the other four sorted positions and, depending on where the new sample will fit in, five more "GOTO" instructions are used to mark the new position. In each of these program parts, in turn, only the smallest number of shifts in the data array are used to insert the new sample into the new sorted order. Twenty-five "GOTO" instructions, as above, are used altogether but the code is *not* redundant. The code is specific to the tasks to be done.

If the median filter is done without ROM States, in the traditional way, the running-time is 113 instruction-cycles and the subroutine takes up about 80 program memory words. If the median filter *does* use ROM States, the running-time is 57 instruction-cycles and the subroutine uses 320 program memory words. Using ROM States cuts the running-time in half but quadruples the program size.

6.16   Limitations of C/C++

The major selling-point of using a C/C++ compiler is that it is said to produce object code that is very nearly like that of an assembly language program while allowing the user to write in machine-independent code. While this is true for traditional programs, high-end ESP programs, and medium-end ESP programs, it is *not* true, in general, for low-end ESP programs.

The C/C++ language can be used successfully in low-end embedded programs which are not timing-critical. Low-end systems that need to work at high speeds or high efficiencies *cannot* use C/C++ because the compiler produces code which is far inferior to assembly language code. Optimum-time low-end code is necessarily a function of the code geometry and is by no means "just an algorithm".

For example, how would a C/C++ compiler for the PIC16F877 fill a twenty-element RAM array with zeros when it is given the statement:

"for(k=0;k < 20;k++) Array[k] = 0;" ?

It would probably produce a loop that counts to twenty and stores zeros in the array by using indirect addressing. This is fine for an application which is not timing-critical but it is useless for one that is. What would be the fastest way to fill a twenty-element array in RAM with zeros using the PIC assembly language? Use twenty "CLRF ARRAYn"

statements in a row! How many standard C/C++ compilers will produce twenty "CLRF"s in a row when given the "for"-loop above? None! In general, how many standard C/C++ compilers for the PIC can make the distinction between making "slow-code" for one part of the program and "fast-code" for another part? None!

If special instructions, keywords, or classifiers were added to the C/C++ language specifically for use in low-end systems to tell the compiler what "speed" of code to use, the resulting compiler would *still* need some advanced artificial intelligence software to get the right code. The blanket statement that, "C/C++ produces object code just like assembly code", when it is applied to low-end systems, is absurd!

Someone will say to me, "All your examples use the 4 MHz PIC when there is a 20 MHz PIC available. Why not go with the 20 MHz PIC and stop complaining about C/C++?" That reply is OK for "all-digital" systems or ones that have shielding and filters (at some extra cost). But what if the product has sensitive analog circuits or it is to be used in an RF-sensitive environment? The RF noise produced by a 20 MHz PIC may be prohibitive! A 4 MHz system may work marginally, but a 20 MHz system may not work at all!

Another idea is to use a 20 MHz PIC and under-clock it to run at, say, 5 MHz, to increase the speed of the system if the maximum speed of 20 MHz cannot be used. This too may have problems.

When I did embedded systems programming and system-design in the security systems industry, our company tried to use our radio receiver with another company's security panel. When we hooked the receiver up to the panel we got a transmitting range of about two inches! The panel used a microprocessor with what was then a fast speed of 12 MHz and its busses were multiplexed. It turned out that the chip they used to demultiplex the busses had signal rise-times of two-nanoseconds! This alone produced enough RFI to block our signal. We substituted a slower demultiplexer chip and we got ranges of about 25 feet! The 12 MHz clock speed never changed. This is why fast chips may be just as bad as high clock speeds.

Embedded systems design, especially at the low-end, is inherently a multi-disciplinary field. Electromagnetic interference and electromagnetic compatibility (EMI/EMC) problems *must* be considered *at the start* of the design process. Often the easiest way to get rid of noise and EMI/EMC problems is to recognize the potential for them at the start of the design process and prevent them from happening in the first place!

Never assume that a system that works at a "low" speed will also always work at a "high" speed.

The special techniques in this chapter are used for "high-speed" operations. In cases where a low clock-speed is used, these are the software techniques that are time-optimal. They make the most efficient use of the processor's time. Using fast chips and C/C++ in a low-end system may not be practical.

Chapter 7:   Advanced ESP

7.0   Chapter Summary

Section 7.2 discusses sine wave generation by the direct digital synthesis method. Section 7.3 uses section 7.2's results to generate Touch Tone/DTMF signals. Section 7.4 covers software generation of pulse width modulation. Section 7.5 covers ADPCM data compression method. Section 7.6 discusses ideas for testing and practical embedded systems.

7.1   Introduction

This chapter will look at some useful techniques, tools, testing methods, and system ideas for advanced ESP. A technique for generating sine waves is developed and is an essential part of an embedded systems programmer's tool-kit. A data compression technique for speech signals is also developed. Testing methods and system ideas are discussed in full.

7.2   Sine Wave Generation

One way to generate high-quality sinusoidal signals is to use the Direct Digital Synthesis method (DDS). This process, also called the "phase-addition method", can generate high frequencies with high resolution and high spectral purity. It requires a digital-to-analog converter (DAC) and an analog low-pass filter to shape the sine wave. The DDS process can be done in hardware or in software.

To develop this technique, consider a look-up table with 256 entries containing a complete sine wave. If one value were taken and sent out to a DAC with a low-pass filter at the DAC output, and this was repeated with a table-increment of one (1) at a rate of ten kilohertz (10 kHz), there would be a sine wave at the filter output of about 39 Hz or roughly 10000 Hz / 256. If the table-increment were increased to two (2), instead of one (1), the resulting output sine wave frequency would be 78.125 Hz or roughly 10000 Hz / 128. We could also say Freq = 10000 Hz / (256 / 2).

What would happen if the table-increment were fifteen (15)? What frequency would be produced and would it still be a sine wave? First, it *would* still be a sine wave. Though the sequence of values in the look-up table would be "choppy", the low-pass

filter would smooth-out the "chop" and produce a clean sine wave. Its frequency would be as follows:

Freq = 10000 / (256 / Table-Increment)
    = 10000 / (256 / 15)
    = 585.9375 Hz
    = 15 * (39 Hz), roughly.

What would be the largest possible frequency that could be produced in this system? It turns out that the largest frequency is exactly one-half of the ten-kilohertz data rate. The minimum number of points needed to produce a sine wave is only two (2). (These points must be at the positive and negative peaks of the sine curve to get the maximum amplitude at the output.) The corresponding table-increment for this case is 128 and gives a frequency of 10000 / (256 / 128) = 5 kHz. (The formal name for the statement that "the maximum output frequency is half of the data rate" is the Nyquist Sampling Theorem. We will see this again in Chapter 10 (DSP Fundamentals).)

Is it possible to produce a frequency less than the 39 Hz? Yes! What if we did a table-increment of one (1) as before but at *every other* time step? This would effectively give a table-increment of "one-half". This would produce a frequency of half of the 39 Hz or 19.53 Hz. If the table-increment of one (1) were done at every *fourth* time step, the output frequency would be one-fourth of the 39 Hz or 9.766 Hz.

Is it possible to produce a frequency that is half way in between the 39 Hz and its double, 78 Hz? Yes! Repeat the following sequence of table-increments over-and-over: Do a table-increment of one (1) at one time step then a table-increment of two (2) at the next time step. This would give an effective table-increment of "one-and-a-half" and an output frequency of 10000 / (256 / 1.5) = 58.594 Hz.

Let's generalize this idea.

Suppose that the sine-table, the DAC, the filter, and the ten-kilohertz data rate are all the same as before. Suppose now that there is a 16-bit register that holds the result of each "sum-of-table-increments" (an "Accumulator"). Suppose that only the upper eight bits of this accumulator will be used to index the sine wave look-up table. Suppose that there is also a 16-bit register that holds the table-increment value which may now be a 16-bit number. The process is now to repeatedly add the 16-bit table-increment value to the 16-bit accumulator over-and-over but let only the upper eight bits of the accumulator do the indexing of the sine wave look-up table. The effect of the lower eight bits of the 16-bit table-increment and the 16-bit accumulator is to provide what would be "fractional" table-increments, relative to the previous "whole-value/alternate time-step" scheme of before.

For example, if the 16-bit table-increment is 0x0100, this would only increment the upper eight bits of the accumulator, and the output frequency would be 39 Hz. Also if the 16-bit table-increment is 0x0180, the following sequence of values would be

produced in the 16-bit accumulator: 0x0000, 0x0180, 0x0300, 0x0480, 0x0600, … and so on. The series of table-increments in the upper eight bits of the accumulator is 0, 1, 3, 4, 6, … and so on. This corresponds to the alternate steps of one and two in the previous examples and the output frequency would be 58.594 Hz.

In general, we can get the ratio of "Frequency-to-Increments" as:

F-to-I = Data Rate / Maximum-Number-of-Register-Values
$$= 10000 \text{ Hz} / (2 ** 16)$$
$$= 10000 \text{ Hz} / 65536$$
$$= 0.152588 \text{ Hz/Inc.}$$

This value is also the lowest frequency that can be produced by this system. (This corresponds to a Table-Increment of 0x0001.)

We can use this ratio to figure out how much of a table-increment we need to produce any frequency. For example, suppose we want a frequency of 777 Hz:

Increment = 777 Hz / F-to-I
$$= 5092.1$$
$$= 5092 \text{ (Rounded Down)}.$$

Therefore, the frequency for this table-increment is Freq = 5092 * F-to-I = 776.98 Hz.

Here is one example of how this algorithm could be coded on the PIC:

```
SINE_DDS:
            MOVLW       DELAY_LOW
            MOVWF       TIME_LOW            ; Set Fixed Duration
            MOVLW       DELAY_HIGH          ; For Sine Wave
            MOVWF       TIME_HIGH
SINE_LOOP:
            CLRF        CARRY               ; Reset Store for Carry Bit
            MOVF        INC_LOW,W           ; Get Low-Half of INC
            ADDWF       ACCUM_LOW,F         ; Add to Low ACCUM
            BTFSC       STATUS,C            ; Get Carry Bit
            INCF        CARRY,F
            MOVF        INC_HIGH,W          ; Get High-Half INC
            ADDWF       CARRY,W             ; Add Carry Bit
            ADDWF       ACCUM_HIGH,F        ; Add to High ACCUM
            MOVF        ACCUM_HIGH,W
            CALL        SINE_LOOK_UP
            MOVWF       DIGITAL_TO_ANALOG
            CALL        DELAY
            DECFSZ      TIME_LOW,F          ; Decrement Duration
            GOTO        SKIP                ; In Constant Time
```

```
                DECFSZ      TIME_HIGH,F
SKIP:

                GOTO        SINE_LOOP
                RETURN
```

The total running-time of this loop must be 100 microseconds to get the ten kilohertz data rate of the example (10 kHz).

7.3   Dual-Tone-Multi-Frequency (DTMF) Signaling

Telephones in the USA use DTMF dialing signals or "Touch Tone" signals. As its name implies, this scheme uses two tones at a time to represent a symbol to dial. The DTMF tones are split into a "high" frequency group and a "low" frequency group and the two tones that are sent are as "one tone from each group". The symbols and their frequencies are as follows:

Symbol	Low Frequency	High Frequency
0	941 Hz	1336 Hz
1	697 Hz	1209 Hz
2	697 Hz	1336 Hz
3	697 Hz	1477 Hz
4	770 Hz	1209 Hz
5	770 Hz	1336 Hz
6	770 Hz	1477 Hz
7	852 Hz	1209 Hz
8	852 Hz	1336 Hz
9	852 Hz	1477 Hz
* (Star)	941 Hz	1209 Hz
# (Pound)	941 Hz	1477 Hz

Each of these tones must fit in a bandwidth of two-percent (2%) in order to be recognized by the telephone company.

The DTMF tones can easily be generated by using the DDS technique. Two DDS processes are run in parallel and the two sine values they produce are added together and sent to the DAC. The same sine wave table is used but with half the maximum amplitude as before so that the adding of the two sine values will not need a "divide-by-two" to fit the DAC. The code for the DTMF DDS process is as follows:

```
DTMF_BY_DDS:
            MOVLW       DELAY_LOW
            MOVWF       TIME_LOW            ; Set Duration of DTMF
            MOVLW       DELAY_HIGH
            MOVWF       TIME_HIGH
DTMF_LOOP:
            CALL        DO_HIGH_TONE_DDS
            CALL        SINE_LOOK_UP
            MOVWF       SINE_HOLD
            CALL        DO_LOW_TONE_DDS
            CALL        SINE_LOOK_UP
            ADDWF       SINE_HOLD,W
            MOVWF       DIGITAL_TO_ANALOG
            CALL        DELAY
            DECFSZ      TIME_LOW,F
            GOTO        SKIP
            DECFSZ      TIME_HIGH,F
SKIP:
            GOTO        DTMF_LOOP
            RETURN
```

The DDS process is abbreviated here but is the same as before in every way.

7.4   Pulse-Width Modulation

A simple way to get around the need for a DAC in many applications is to use pulse-width modulation (PWM). A PWM signal represents an analog value by varying the duration of digital pulses. Just as a DAC uses variable voltages over constant time to express analog values, PWM uses digital pulses over variable time. Integrating the DAC signal and the PWM signal over time will give the same results. A PWM output needs only a single digital output pin. An analog low-pass filter is still required, however.

The PIC16F877 already has two built-in hardware PWM units. Even so, it is useful to know how to do PWM in software in case the processor you are using does not have a hardware PWM unit. The basic PWM program is as follows:

```
PWM_ROUTINE:
            MOVWF       SAMPLE1     ; Call with W = PWM Output
            MOVWF       SAMPLE2
            BSF         PORTC,OUTPUT_BIT   ; Assume This Output
PWM_LOOP1:
            DECFSZ      SAMPLE1,F  ; Send "High" for Duration W
            GOTO        PWM_LOOP1
```

```
                BCF             PORTC,OUTPUT_BIT
PWM_LOOP2:
                INCFSZ          SAMPLE2,F  ; Send "Low" for
                                         ;    Duration 256 – W
                GOTO            PWM_LOOP2
                RETURN
```

The total time this routine takes to run is about 770 microseconds. If each PWM DAC
value took this long, the data rate would be 1300 Hz. Accordingly, only frequencies up
to 650 Hz could be represented. If the total time were only 100 microseconds, a signal of
five kilohertz (5 kHz) could be represented, but the total range and resolution in the
above software loop would be only 32 steps. That is a maximum value of 32 could be
sent as an output value.

        This is the main problem and tradeoff in using PWM. It does not matter if the
PWM is done in hardware or in software. The same problem and tradeoff exists for both
cases. It is a fundamental problem.

        The software PWM process can be improved by using different software
techniques such as the "non-loop" time-delay process demonstrated in Chapter 6. This
will allow for a 100 microsecond data rate and a range and resolution of one hundred
(100). The full range of values is from zero to 100. This PWM program is as follows:

```
PWM_PROCESS:
                MOVWF       LOW_DURATION
                SUBLW       D'100'                  ; Form W = 100 – W
                ADDWF       PCL,F                   ; For High Duration
                BSF             PORTC,OUTPUT_BIT
                BSF             PORTC,OUTPUT_BIT
                BSF             PORTC,OUTPUT_BIT
        ---- Continue "BSF"s for a total of 100 "BSF"s ------
                BSF             PORTC,OUTPUT_BIT
                MOVF        LOW_DURATION,W
                ADDWF       PCL,F
                BCF             PORTC,OUTPUT_BIT
                BCF             PORTC,OUTPUT_BIT
                BCF             PORTC,OUTPUT_BIT
        ---- Continue "BCF"s for a total of 100 "BCF"s -----
                BCF             PORTC,OUTPUT_BIT
                RETURN
```

For example, if W = 90 when this routine was called, the "ADDWF PCL,F" after the
"SUBLW" would do a "GOTO" to the point only ten steps away. This would allow 90 of
the "BSF"s to run. When the "Low Duration" value of 90 is given to the "ADDWF", this
would do a "GOTO" to a point of 90 steps away, and allow only ten "BCF"s to run. This
solution is another example of the "ROM vs. Time" tradeoff in embedded systems

programming. It is much longer and uses more than 200 ROM words, but it is faster and allows the greatest resolution.

There is an assembly directive that can simplify the fast PWM process called "FILL" which fills in repeated instructions in program memory. Its form looks like:

FILL   (assembly instruction),number-of-times

For example, "FILL   (NOP),20" will fill in twenty "NOP"'s into program memory.

The fast PWM loop can be re-coded as follows:

```
PWM_PROCESS:
        MOVWF       LOW_DURATION
        SUBLW       D'100'
        ADDWF       PCL,F
        FILL        (BSF  PORTC,OUTPUT_BIT),100
        MOVF        LOW_DURATION,W
        ADDWF       PCL,F
        FILL        (BCF  PORTC,OUTPUT_BIT),100
        RETURN
```

Before the sound cards came out for the IBM-PC, PWM was used by some companies to send speech to the PC's built-in speaker attached to a digital output port pin. The data rate was high enough that the speaker could not respond at that frequency and it did not emit a high-pitched squeal. Clear speech came out of the PC's speaker.

7.5   ADPCM Data Compression

ADPCM is a coding technique that allows an easy way to compress speech signals in real-time by a factor of 2:1 or better. The compression allows some loss of the exact information, however. ADPCM stands for "Adaptive Differential Pulse Code Modulation".

ADPCM encodes the difference between successive signal samples and a running approximation to the signal. It is also adaptive in that if the signal is too large or too small relative to the approximation of the signal, the range or scale is changed for the next sample. Changes in volume are automatically compensated for and are reflected in the coding. The encoding does introduce some noise, but there are filtering techniques that can remove the noise and give very clear speech in the output (these filtering techniques will be shown in detail in Chapter 10 --- DSP Fundamentals).

Suppose that speech is sampled at an appropriate data rate and is read as 8-bit samples. After the ADPCM process, the data can be reduced to four-bits and then two of these four-bit samples can be packed into one byte. The approximation to the signal is stored in an 8-bit accumulator initially set to zero. When a signal value comes in, the difference between it and the accumulator is found and is tested against a range of allowed values. The best-fit of the difference to those values is found and a "code" or an "index" is used to represent the best-fit value. If the best-fit value is at the maximum or the minimum of the range of values, the range of values is doubled or halved, respectively, for the next input sample when it is read the next time. The value of the accumulator is updated with the best-fit value in the table by adding it to the accumulator.

For example, if the initial range is of 15 values from the set {-7, -6, -5, …, -1, 0, 1, …, 5, 6, 7}, doubling this range will give a range of values as {-14, -12, -10, …, -2, 0, 2, …, 10, 12, 14}. However, the range will not be doubled beyond a maximum range nor will it be halved below a minimum range.

It should be noted that although the ADPCM process will reproduce the input from the output, it does introduce noise and is a nonlinear process. Changes in volume are accounted for but the process of doing so is not instantaneous. The "codes" or "index" values mentioned above are four-bits wide and *are* the compressed data values.

The ADPCM compression-phase program is as follows:

```
ADPCM_COMPRESS:
        SUBWF   ACCUM,W  ; W = ACCUM – W
        SUBLW   0        ; W = W – ACCUM
        MOVWF   DIFF       ; Store the Difference
        CALL    GET_STEP_CODE ; Double-Index Table
                            ;    Look-Up
                            ; on Scale & Difference
        MOVWF   CODE
        CALL    GET_STEP   ; Double-Index Table Look-Up
                            ; on Scale & Code: Get Best-Fit
        ADDWF   ACCUM,F    ; Update ACCUM, Form Sum
        CALL    GET_SCALE ; Double-Index Table Look-Up
                            ; Judge Max/Min Range, Get
                            ; New Scale Index
        MOVWF   SCALE
        BTFSS   FLAGS,DATA_READY
        GOTO    ADPCM_COMBINE ; Pack the Code, Set
                                ;   "Ready"
        NOP
        MOVF    CODE,W     ; Set New Half-Byte
        ANDLW   0x0F
        MOVWF   HALF_OUT  ; Store New Half Code
        BCF     FLAGS,DATA_READY ; Data NOT Ready Yet
```

```
                RETLW      0xFF                    ; Return an Invalid Code


ADPCM_COMBINE:
                SWAPF      CODE,W      ; Put 2nd Code in Upper Half Byte
                ANDLW      0xF0
                IORWF      HALF_OUT,W       ; Combine two Codes
                BSF        FLAGS,DATA_READY
                RETURN                          ; W = Packed Byte Codes
```

The "GET_STEP_CODE" routine is a double-indexed table look-up with "DIFF" and "SCALE" as the indices. First, do the "SCALE" index as:

```
GET_STEP_CODE:
                MOVF       SCALE,W
                ADDWF      PCL,F
                GOTO       SCALE0
                GOTO       SCALE1
                GOTO       SCALE2
                GOTO       SCALE3
                GOTO       SCALE4
```

The "SCALEn"s are for 256-entry tables indexed with the "DIFF" value. The object of each one is to return the "CODE" value for the closest approach (best-fit) values in the following table:

SCALEn	Max/Min Values	Delta Steps
0	+-7	1
1	+-14	2
2	+-28	4
3	+-56	8
4	+-112	16

There are 15 codes for each of the SCALEn tables. These codes run as zero for the most negative value and 14 for the most positive value. The reason why the double hashing is used and why the tables are allowed to take up so much memory is so that this routine can run at the highest possible speeds. If this were not so, the time it would take to find the codes, the steps, and the scales would be prohibitive and the ADPCM process would be too slow to run in a 4 MHz PIC. A 20 MHz PIC might not have this problem and a smaller ADPCM routine could be used.

Once the "CODE" is found and the "SCALE" is given, these are used to get the "STEP" or "Best-Fit Value", update the accumulator, and adjust the "SCALE" for the next time. The "CODE"s are then packed in groups of two to a byte and returned in W to the calling program. The value of "SCALE" must not be disturbed by the rest of the program between subroutine calls. The "SCALE" and the accumulator must be initialized to zero and the "Data-Ready" flag must be "set" (=1) initially.

The ADPCM expansion process is similar to the compression process except that the "CODE" is *given* as the compressed data to be expanded. This process looks like:

```
ADPCM_EXPAND_NEW_CODE:
        MOVWF     HOLD_CODE          ; W = New Packed Code
ADPCM_EXPAND_CODE2:
        ANDLW     0x0F               ; Isolate First Code
        MOVWF     CODE
        CALL      GET_STEP           ; Get Corresponding
                                     ; Best-Fit Value
        ADDWF     ACCUM,F            ; Update Accumulator
        CALL      GET_SCALE          ; Get Next Scale
        MOVWF     SCALE
        MOVF      ACCUM,W            ; Current Accum = Output
        RETURN                       ; W = Expanded Data


ADPCM_EXPAND_SECOND_OLD_CODE:
        SWAPF     HOLD_CODE,W    ; Get 2nd Code
        GOTO      ADPCM_EXPAND_CODE2
```

The expansion is done in two parts. Once for a new code byte to get the first expansion and then once for that same byte to get the second expansion.

7.6   Test Functions and System Ideas

Embedded systems should make liberal use of test pins and test functions. They are useful in all phases of development, production, testing, and field-testing. Special DIP-switches or header pins should be added to the system for testing purposes alone. Fifteen or twenty percent of the available program code space should be reserved, if possible, for testing and de-bugging aids.

The simplest, most common, and most general testing and de-bugging aids use LEDs. An LED can be made to flash at a given rate to show a system's activity. A non-flashing LED would indicate a system failure. PWM can be used on an LED to make it bright or dim, which can also serve as a system indicator. An LED can flash when the watch dog timer gets its reset pulse. In an encoder or a transmitter, an LED can show when a data-word is transmitted or, in a decoder or receiver, an LED can flash when a data-word is decoded.

If the system uses seven-segment LED digits, the system can display messages for testing and de-bugging. Detailed messages for showing the data and the system status can be displayed.

An oscilloscope with a variable-delay sweep can be used to display the system's data and status using a serial stream of pulses. A single digital output pin can be used to send this information. A simple sequence of instructions can be used to send out pulses. For example:

```
BCF        PORTC,TEST_PIN
BTFSC      RAM_WORD,DATA_BIT
BSF        PORTC,TEST_PIN
CALL       DELAY1
BCF        PORTC,TEST_PIN
CALL       DELAY2
```

Also, for events which occur too quickly for the human eye to see, a pulse like that above can be generated, held active for a few hundred milliseconds, and then removed. This aids in the detection of transient events like a receiver's "valid data decode" signal.

A security system could be made to report its status information or it could be set up with pseudo-random time delays for activating its alarms. If it can dial telephones, it could call the factory or office and send a complete set of systems information for that day. A special alarm mode could be reserved for units that get frequent or repeated resets.

A system can often incorporate utility functions like a voltmeter, a frequency counter, a timer, an alarm clock, a signal generator, a waveform generator, a pulse generator, a noise generator, or a logic analyzer. Any of these things may be useful in a field-testing situation.

If your embedded system interfaces with another (larger?) system, primitive "look-alike" signals can be generated in your system to test it "as if" you had the other system. That is, send diminutive mock-ups of the other system's signals to mimic its actions so that you don't need to carry that system around with you to test your system.

Systems which are highly interactive with user menus can be built with a series of interconnected jump-tables. A time-out feature should be included to partially reset the command sequence if the user does not enter a command, does not finish responding to the system in the way that the system "expects", or if the user makes an error. Such a system should be user friendly. The time-out can take the user back to a previous menu or later to the main menu but give a warning before doing so.

Embedded systems can be made to be adaptive and have self-testing and self-diagnostic features. For example, a decoder or a receiver could include the code for the encoder and produce its own mock-up of a "transmitted signal with noise". The decoder/receiver could then try to decode and get calibrated with its own test signal.

EEPROMs can be used to store calibration information and other system settings for its normal operation. The system can be built with its own calibration routines to set-up a new or updated system.

Sometimes one processor isn't powerful enough to perform all of the required actions that the system needs. There may be severe time bottlenecks. The solution may be to use a second or even, a third processor. Always watch out for the possibility that your system may be over burdened and that the best solution might be to include more processors.

Some kinds of system failures may be statistical in nature. It may be necessary to measure probability distributions and statistical averages in order to diagnose the problem.

Gaussian white noise can be used to *improve* a stable, DC-valued ADC reading by plotting a histogram of the samples and comparing the histogram to a known Gaussian distribution. The histogram is just a count of the number of times the data occurs. This is accomplished by finding the upper and lower limits of the noise signal and setting up a RAM array for the values within these limits (start with the RAM array reset to zeros). The data is sampled and used to hash the RAM array then the value of the RAM at the hashed address is incremented. After this process is repeated several hundred times, the discreet histogram is formed. The discreet histogram values can be used to interpolate between the ADC steps. It is possible to use a ten-bit ADC with this technique and produce the equivalent of a 16-bit ADC! This technique of using noise to improve a signal's quality or measurement is called "dithering". Figure 7-1 gives an illustration of this process.

For processors that do not have ADCs built-in, it is possible to measure DC-valued voltages using dithering on a one-bit ADC. The one-bit ADC is just a comparator. The voltage to measure is fed in on one input and Gaussian white noise is fed in on the other input. This produces a set of random, square wave pulses that the processor can measure. The time duration, frequency, and overall time are measured and are then compared to a known Gaussian distribution. The position on the Gaussian curve can be found and the voltage can be determined to an almost arbitrary accuracy. It is not uncommon to get measurements and an accuracy of up to 24-bits by using this method.

One lesson to be learned from the concept of dithering is that noise itself can be used as a valuable signal in its own right. There are of course many times that noise is bad but it can sometimes be used to a great advantage.

$$f(X) = \frac{1}{\sqrt{2Pi}\ s}\ e^{-\left(\frac{X}{s}\right)^2}$$

f(X)

1

X

0

Sampled Data Points

20   87   234   378   370   220   79

-1.5   -1.0   -0.5   0.0   0.5   1.0   1.5   X
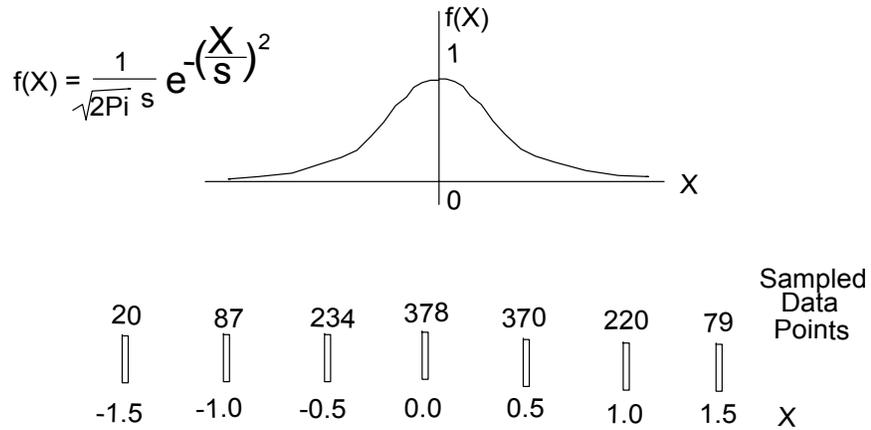
Notice the Asymmetry About the Peak of 378

Fitting a Gaussian Curve Gives X-Offset = 0.23 and S = 1.0

Figure 7-1 Gaussian Probability Density Function and a
Set of Sampled Values

Chapter 8:   PIC Peripherals and Interrupts

8.0   Chapter Summary

Section 8.1 gives an overview of the PIC's peripherals. Section 8.2 covers the input/output ports. Section 8.3 discusses the PIC's interrupt system. Section 8.4 discusses the analog to digital converter and the analog multiplexer. Section 8.5 covers the PIC's built-in watch-dog timer. Sections 8.6, 8.7, and 8.8 cover the counters/timers. Section 8.9 covers the capture mode operations. Section 8.10 covers the compare mode operations. Section 8.11 discusses the PIC's hardware pulse width modulators. Section 8.12 covers the parallel slave port. Section 8.13 discusses reading and writing the data EEPROM. Section 8.14 discusses reading and writing the program memory. Section 8.15 discusses the data protection modes. Section 8.16 covers the CONFIGURATION word and its settings. Section 8.17 discusses the PIC's sleep and reset modes.

The PIC16F877 is more than just a CPU with some RAM and ROM. There are also several useful peripheral hardware modules built-in to make it a complete computer control system. Many products can be made that use the PIC as a complete single-chip solution to its system design.

This chapter will focus on the peripherals and special functions of the PIC. The details and instructions on how to use them will be given in full with examples.

8.1   Overview of the PIC Peripherals

1) Input/Output Ports
There are five port-sets as Ports (A,B,C,D,E) with bits and pins that may be set in software as inputs or outputs. These pins are also shared with other peripheral functions and to use these other functions requires setting up the input/output ports for compatibility. Thirty-three (33) input/output port pins are available.

2) Interrupts
As briefly noted in Chapter 6, an interrupt is a way for peripherals and other hardware to capture the attention of the CPU and have it call a subroutine to service the hardware with the user's software. The subroutine address is at a fixed location in program memory and is built-in to the CPU. There are a total of 14 possible interrupts and each of them can be enabled or disabled in software.

3) Analog-to-Digital Converter & The Analog Multiplexer
The PIC contains a ten-bit ADC and has as many as eight available analog input channels. These analog inputs are shared with the port pins of Port A and Port E.

It is also possible to select among these pins a place to attach an external voltage reference in the case where the user does not want to use the internal voltage reference of five volts. This feature is also software selectable.

4) The Watch-Dog Timer
As noted in Chapter 6, the PIC contains its own independent watch-dog timer which can be disabled not by software but by a setting in the downloading process. The watch-dog timer has the power to reset the PIC when it overflows. The user's job in the software is to provide the watch-dog timer with regular commands to reset the watch-dog timer so that it will not reset the PIC.
The purpose of the watch-dog timer is to make sure the software does not get trapped in infinity loops and thus the software is made much more reliable by using the watch-dog timer. In addition to the watch-dog timer is a prescaler which can extend the watch-dog's time-out period.

5) Timer0, Timer1, and Timer2
These three timers can count clock pulses or count external pulses on the PIC's port pins. They are programmable and have prescalers and sometimes postscalers to modify their counts and counting processes. They often serve as time-bases for other peripherals or for providing regular interrupts to the user's software.

6) Capture Mode (Two of them)
The capture mode modules are hardware-controlled ways to measure pulse-widths with reference to the system clock.

7) Compare Mode (Two of them)
The compare mode modules compare Timer1's count to a fixed, but user-programmable, register value. When that value is reached, the hardware can send out a pulse, trigger an interrupt, or do some kind of "special event" within the PIC such as resetting and reloading Timer1 and/or starting the ADC. This is also useful for setting up a time-base for the software.

8) Pulse-Width Modulation (Two of them)
The PWM modules use Timer2 to generate signals on a PIC output pin and serve as DACs. Both PWM modules have a maximum of ten bits of resolution.

9) Parallel Slave Port
Port D can be used as a bi-directional, microprocessor-type, data-bus port. The PIC is controlled as a slave to three external control signals which are manipulated by the microprocessor or other device.

10) EEPROM Data Memory
The PIC has 256 bytes of non-volatile EEPROM and can be programmed in the PIC software independent of a device programmer.

11)   FLASH Program Memory

In a similar way to the EEPROM Data Memory, the FLASH Program Memory can also be programmed in software independent of a device programmer. New program features and updates can be downloaded to the PIC's software while the unit is in the field.

12)   Code and Data Protection

The PIC can be configured to lock-out attempts to read or write its FLASH and EEPROM Data Memories. This can be set only during the downloading process.

13)   Resets and Sleep Mode

The PIC has several modes of resets and status conditions to identify which of them has occurred. Also, there is a power-saving mode called "sleep" which can be initiated in software to power-down the CPU. This is useful in battery powered applications.

In all of the sections to follow, the special function registers and the bits that control the peripherals are shown in detail in Appendix C.

8.2   Input/Output Ports

There are five input/output ports as Port (A, B, C, D, E). Most of these act alike but some are different and use different features and options.

8.2.1   Port A

There are six (6) Port A pins and each can be set as an input or an output. All of the Port A pins except RA4 have a shared use with the analog multiplexer (MUX). The RA4 pin can be a Schmitt Trigger input (hysteresis) or an open-drain output.

The first step in configuring Port A is to select which pins are to be set as digital input/output and which are to be analog inputs. This is done with the lower four bits of the ADCON1 register file (RAM). These settings are shown in Figure 8-1. If we want all of Port A to be digital, the setting is 0x06.

Of the pins which are digital input/output the selection of "input" vs. "output" is made with the TRISA register file. If a Port A pin is to be an "input", the TRISA bit corresponding to the Port A pin/bit must be set to one (=1). If it is to be an "output", the TRISA bit must be set to zero (=0).

PCFG3:PCFG0 are bits 3,2,1,0 in ADCON1

PCFG3: PCFG0	AN7 RE2	AN6 RE1	AN5 RE0	AN4 RA5	AN3 RA3	AN2 RA2	AN1 RA1	AN0 RA0
0000	A	A	A	A	A	A	A	A
0001	A	A	A	A	Vref+	A	A	A
0010	D	D	D	A	A	A	A	A
0011	D	D	D	A	Vref+	A	A	A
0100	D	D	D	D	A	D	A	A
0101	D	D	D	D	Vref+	D	A	A
0110	D	D	D	D	D	D	D	D
0111	D	D	D	D	D	D	D	D
1000	A	A	A	A	Vref+	Vref-	A	A
1001	D	D	A	A	A	A	A	A
1010	D	D	A	A	Vref+	A	A	A
1011	D	D	A	A	Vref+	Vref-	A	A
1100	D	D	D	A	Vref+	Vref-	A	A
1101	D	D	D	D	Vref+	Vref-	A	A
1110	D	D	D	D	D	D	D	A
1111	D	D	D	D	Vref+	Vref-	D	A

D = Digital I/O   A = Analog Input Channel

Figure 8-1 ADCON1 "Analog vs Digital" Selection Codes

For example, if we want all of the Port A bits as "digital" and RA0, RA1, RA2 are to be outputs and RA3, RA4, and RA5 are to be inputs, the code would be as follows:

```
BANKSEL        PORTA        ; Bank 0
CLRF           PORTA        ; Reset Port A Before Configure
BANKSEL        ADCON1       ; Bank 1
MOVLW          0x06         ; Select "All Digital" on Port A
MOVWF          ADCON1
MOVLW          0x38         ; RA(0,1,2) = Out
MOVWF          TRISA        ; RA(3,4,5) = In
BANKSEL        PORTA        ; Bank 0
```

8.2.2   Port B, Port C, Port D

None of Port B, Port C, or Port D is shared with the analog multiplexer and none need be configured with the ADCON1 register file. There are TRISB, TRISC, and TRISD register files which control the "input" vs. "output" selections of each of these port pins. These work in the same way as Port A. The Port B pins also have a user-selectable "weak pull-up" option that can be enabled by clearing the "/RBPU" bit of the "OPTION_REG" register file. (Do "BCF OPTION_REG,RBPU".) This feature is automatically disabled when a Port B pin is configured as an output.

8.2.3   Port E

The Port E pins also share their pins with the analog multiplexer as in Port A and its use is identical with Port A. (Use the ADCON1 and TRISE register files.)

8.3   Interrupts

The PIC16F877 has a total of 14 sources of interrupts. Each of these has an "Interrupt Enable Bit" which must be set (=1) to enable or use the interrupt and an "Interrupt Flag", which is set (=1) automatically when the interrupt is activated. There is also a "Global Interrupt Enable Bit" and a "Peripheral Interrupt Enable Bit". The "Global" must be set (=1) before any of the other interrupts will be enabled. The "Peripheral" must be set (=1) before any peripheral interrupt will be enabled. Figure 8-2 shows the schematic for the PIC's interrupt logic.

When an interrupt occurs the CPU treats it like a subroutine "CALL" and the program-counter is set for the program address, 0x0004. The user must place the interrupt-handling subroutine at this address (this subroutine is called, the "Interrupt Service Routine" or "ISR"). A special instruction is used to return from an ISR: It is the "RETFIE" instruction.

Before returning from an interrupt, the user must reset the "Interrupt Flag" which was set (=1) when the interrupt occurred. If this is *not* done, the interrupt hardware will automatically and immediately cause another interrupt to occur when the CPU executes the "RETFIE" instruction! It is also possible to *inadvertently* cause an interrupt to occur when the user sets (=1) an "Interrupt Enable Bit". If the corresponding "Interrupt Flag" is set (=1) when the user sets the "Interrupt Enable Bit", an inadvertent or accidental interrupt will occur!

Figure 8-2 PIC16F877 Interrupt Logic Tree
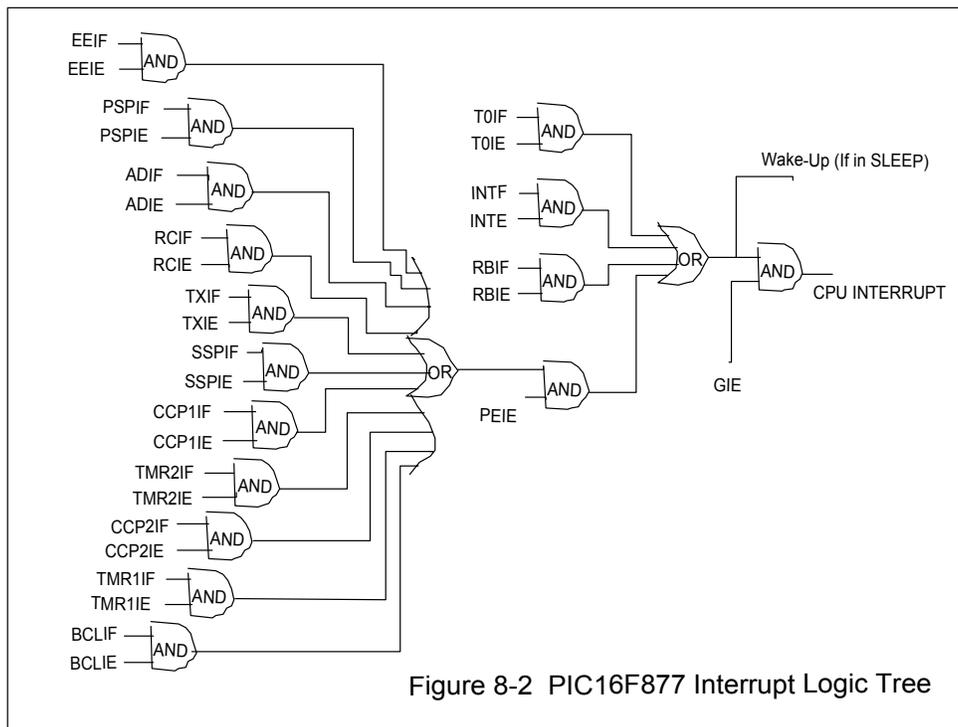
The user's response to an interrupt in the ISR must do two things:

1) The contents of the W register and the "STATUS" register must be saved in RAM.
2) The set of "Interrupt Flags" being used must be searched to determine which "Interrupt Flag" caused the interrupt to occur.

Before returning from an interrupt, four things must be done in the following order:

1) When the peripheral which caused the interrupt has been serviced, the peripheral's "Interrupt Flag" must be reset by the software.
2) All of the other "Interrupt Flags" of the ones being used, must be checked to see if they are reset (=0). If any *are* set, another interrupt by a different peripheral has been called and it must be serviced. When it has been serviced, go back to Step 1.
3) Restore the W and "STATUS" registers.
4) Do the "RETFIE" instruction.

Note that the worst-case response-time for any interrupt, when it has been initiated, is four instruction cycles in duration.

The code to save the W and "STATUS" registers has some tricks to it. The code is as follows:

```
MOVWF      W_TEMP              ; Save W
SWAPF      STATUS,W            ; Get STATUS into W
MOVWF      STATUS_TEMP    ; Save STATUS
```

The code to restore these registers is as follows:

```
SWAPF      STATUS_TEMP,W  ; Restore STATUS
MOVWF      STATUS
SWAPF      W_TEMP,F            ; Restore W
SWAPF      W_TEMP,W
```

The trick is that we can't use "MOVF" anytime we want to move data from a register file to the W register since the "MOVF" instruction affects the "Z" (Zero) STATUS Flag! Doing so would destroy the "Z"-Flag state we are trying to preserve! This is why the "SWAPF" instructions are used. (This code is repeated in Appendix F.)

Now let's look at two specific interrupts and their options. The first is the external interrupt ("INT") which is located on pin 33 and is shared with Port B, Bit Zero (RB0). To use the "INT", Port B, Bit Zero must be set-up with TRISB to be an "input" port pin (TRISB, Bit 0 = 1). If this were *not* so, and if Port B, Bit Zero were an "output" pin, whatever value was on the pin would jam the "INT" input. The "INT" interrupt is edge-sensitive and the user may select one of either the positive or the negative edges using the "INTEDG" bit in the "OPTIONS_REG" register file. If "INTEDG" = 1, this selects the positive or rising edge. If "INTEDG" = 0, this selects the negative or falling edge.

The code sequence to activate the "INT" interrupt is as follows:

```
BANKSEL    TRISB          ; Bank 1
BSF        TRISB,0        ; Set Port B, Bit Zero as an "Input"
BSF        OPTIONS_REG,INTEDG  ; Set "Rising" Edge
BCF        INTCON,INTF              ; Reset "Interrupt Flag" of INT
BSF        INTCON,INTE             ; Set "Interrupt Enable Bit" of INT
BSF        INTCON,GIE              ; Set "Global" Int Enable
```

To reset the "INT"'s "Interrupt Flag" when the ISR is done, do, "BCF   INTCON,INTF".

Another interrupt is the "RB Port-Change Interrupt". When this is enabled, any change on Port B (RB7, RB6, RB5, RB4) will cause an interrupt. These bits must be selected as inputs with TRISB. The code to enable this interrupt is:

```
BCF           INTCON,RBIF        ; Reset "Interrupt Flag"
BSF           INTCON,RBIE        ; Set "Interrupt Enable Bit"
BSF           INTCON,GIE         ; Enable "Global" Interrupt Enable
```

To reset the "Interrupt Flag", do, "BCF   INTCON,RBIF".

In the ISR for this interrupt, Port B must be read to prevent a latch-up of the PortB bit-change that caused the interrupt to occur. Reading Port B between interrupts may cause a mistake in the interrupt process.

There are interrupts available for each peripheral device. These will be discussed when the peripherals are covered.

8.4   ADC and Analog MUX

The PIC's ADC can convert an analog voltage to a ten-bit number. The analog multiplexer (MUX) will allow up to 8 analog input channels to be converted by the ADC. The voltage range limits may be taken internally as "+5 Volts and Ground", or the user can supply an external voltage reference on the analog input channel pins.

The ADC module uses two control registers in the register file map to set-up the ADC and the analog MUX. These are "ADCON0" and "ADCON1". The ADC may be used with or without interrupts.

The first step in using the ADC is to set-up the analog MUX inputs which are shared with Port A and Port E. This was discussed under using Port A by selecting a code from Figure 8-1 and putting that four-bit code into the lower four bits of "ADCON1". For the selected analog MUX inputs, the TRISA and TRISE registers must be set-up so that those Port A and Port E pins are configured as "inputs". If they are configured as "outputs", a voltage of either five volts or ground will jam the ADC.

The next step is to select the ADC conversion clock rate using the "ADCS1" and "ADCS0" bits of the "ADCON0" register. The selections are as follows:

ADCS 1:0	Clock Rate
(0,0)	Fosc / 2
(0,1)	Fosc / 8
(1,0)	Fosc / 32
(1,1)	Internal RC Clock (6 microseconds)

The rule is that the Clock Rate period must not be less than 1.6 microseconds. For Fosc = 4 MHz this gives Fosc / 8 = 500 kHz and a period of 2.0 microseconds. Therefore, select ADCS 1:0 as (0,1).

If the ADC is to be used with interrupts, the following code sequence will set this:

```
BANKSEL    PIR1          ; Bank 0
BCF        PIR1,ADIF     ; Reset "Interrupt Flag" for ADC
BANKSEL    PIE1          ; Bank 1
BSF        PIE1,ADIE     ; Set "Interrupt Enable Bit" for ADC
BSF        INTCON,PEIE   ; Set Peripheral Interrupt Enable
BSF        INTCON,GIE    ; Set Global Interrupt Enable
```

The results of the ADC conversion are ten bits long and are split into two, eight-bit register file bytes: "ADRESH" for the high bits and "ADRESL" for the low bits. The storage of the ten-bit result can be formatted as "Left-Justified" or "Right-Justified". In the "Left-Justified" format, the 8 most significant bits are placed in "ADRESH" and the two least significant bits are placed in bits 6 and 7 of the "ADRESL" register. In the "Right-Justified" format, the two most significant bits are placed in bits 1 and 0 of the "ADRESH" register and the 8 least significant bits are placed in "ADRESL". The user selects the format by setting or clearing the "ADFM" bit of the "ADCON1" register as follows:

ADFM = 1   =   Right Justified
ADFM = 0   =   Left Justified.

The ADC must also be turned "on" with the command, "BSF ADCON0,ADON". It can be turned "off" by using "BCF", if needed, to conserve power.

When the ADC is used, the analog input channel must be selected. There are eight channels and only one may be selected at a time with the "CHS2:CHS0" bits of "ADCON0" as follows:

CHS2, CHS1, CHS0	Channel	Name
(0,0,0)	Channel 0	AN0
(0,0,1)	Channel 1	AN1
(0,1,0)	Channel 2	AN2
(0,1,1)	Channel 3	AN3
(1,0,0)	Channel 4	AN4
(1,0,1)	Channel 5	AN5
(1,1,0)	Channel 6	AN6
(1,1,1)	Channel 7	AN7

When the single channel to be used is selected, as above, the ADC can be started as:

```
BCF        PIR1,ADIF    ; Reset "Interrupt Flag" for ADC
BSF        ADCON0,GO    ; Start the ADC
```

The "ADIF" interrupt flag is set when the ADC conversion process is finished. If the "ADIE" interrupt enable bit is set (=1), setting the "ADIF" will cause an interrupt to

occur. Even if the "ADIE" interrupt enable bit is *not* set, the "ADIF" interrupt flag will be set (=1) when the ADC is finished and it can be used independently of the interrupt system. The "ADIF" can be tested by the software to see if the ADC conversion is finished.

Let's look at an example program that uses the ADC. Suppose that we want to use only Channel Zero (AN0) and display the Left Justified, most significant results on Port B.

First, here is an ADC example using interrupts:

```
                LIST   P=16F877
                INCLUDE "P16F877.INC"

FLAGS:      EQU         0x20   ; User's Flags, Bit 0 = "Data Ready"
ADC_DATA: EQU         0x21   ; User's ADC Data Hold
DATA_READY:  EQU        0        ; Data Ready = Bit 0

                ORG         0x0000
                GOTO        INIT
                ORG         0x0004
                GOTO        INT_SERVICE

                ORG         0x0006
INIT:
                BANKSEL     PORTB        ; Bank 0
                CLRF        PORTB        ; Reset PORTB
                BANKSEL     TRISB        ; Bank 1
                CLRF        TRISB        ; Port B = All Outputs
                MOVLW       0x0E         ; Chan 0, AN0, Left Justify
                MOVWF       ADCON1
                BSF         TRISA,0      ; RA0 = Input
                BANKSEL     PORTB        ; Bank 0
                MOVLW       0x41         ; ADC = "on", Chan 0 Select,
                MOVWF       ADCON0       ; Clock = Fosc / 8

                BCF         PIR1,ADIF    ; Reset "Interrupt Flag" for ADC
                BANKSEL     PIE1         ; Bank 1
                BSF         PIE1,ADIE    ; Set ADC Int Enable Bit
                BANKSEL     PORTB        ; Bank 0
                BSF         INTCON,PEIE  ; Set Peripheral Int Enable
                BSF         INTCON,GIE   ; Set Global Int Enable
                BCF         FLAGS,DATA_READY ; Reset Data Ready
                                              ; Flag
                BSF         ADCON0,GO              ; Start the ADC
```

```
MAIN:
          CLRWDT                    ; Reset Watch-Dog Timer
          BTFSS     FLAGS,DATA_READY  ; Check if ready
          GOTO      MAIN

          BCF       FLAGS,DATA_READY  ; Reset Data Ready
          MOVF      ADC_DATA,W      ; Get ADC Data
          MOVWF     PORTB           ; Send to Port B
          GOTO      MAIN

INT_SERVICE:
          ----- Save Registers ----- (See Appendix F)---
          MOVF      ADRESH,W        ; Get Raw ADC Results
          MOVWF     ADC_DATA        ; Save in User's Hold
          BCF       PIR1,ADIF       ; Reset "Int Flag"
          BSF       FLAGS,DATA_READY ; Set Data Ready
          BSF       ADCON0,GO              ; Start ADC Again
          ----- Restore Registers -----
          RETFIE                    ; Return From Interrupt
          END
```

Now, here is a non-interrupt ADC example:

```
          LIST   P=16F877
          INCLUDE "P16F877.INC"

          ORG       0x0000
          BANKSEL   PORTB       ; Bank 0
          CLRF      PORTB       ; Reset PORTB
          BANKSEL   TRISB       ; Bank 1
          CLRF      TRISB       ; Port B = All Outputs
          MOVLW     0x0E        ; Chan 0, AN0, Left Justify
          MOVWF     ADCON1
          BSF       TRISA,0     ; RA0 = Input
          BANKSEL   PORTB       ; Bank 0
          MOVLW     0x41        ; ADC = "on", Chan 0 Select,
          MOVWF     ADCON0      ; Clock = Fosc / 8
MAIN:
          BCF       PIR1,ADIF   ; Reset "ADC Done"
          BSF       ADCON0,GO ; Start ADC
          CLRWDT                ; Reset Watch-Dog Timer

TEST:
          BTFSS     PIR1,ADIF   ; Is ADC Finished?
```

```
GOTO        TEST

MOVF        ADRESH,W  ; Get Raw ADC Data
MOVWF       PORTB        ; Send to Port B
GOTO        MAIN
END
```

The ADC has its own sample-and-hold amplifier with a built-in capacitor and the total conversion-time is a combination of the amplifier settling-time, the sample-and-hold capacitor charging-time, the temperature, and the successive approximation convergence-time of the ADC clocked logic. The detailed conversion-time breakdown is as follows:

Time of Amplifier Settling = 2 microseconds.
Time of S & H Charging = (Chold)*(Ric + Rss + Rs)*(ln(1/2047))
                                  = 16.5 microseconds.
Time of Temperature Coef = (T – 25)*(0.05 microseconds/ Celsius Degree)
                                    = 2.5 microseconds (worst case)
Time of ADC Logic = 12*(ADC Clock Period)
                          = 12*(2 microseconds) = 24 microseconds.

Total ADC Conversion-Time = 45 microseconds.

Therefore, the ADC can be cycled at a rate of 22.2 kHz.

The voltage step-resolution of a ten-bit ADC over a voltage range of "+5 Volts to Ground" is figured as:

Voltage Resolution = ((+5 Volts) – (0 Volts)) / ((2 ** 10) – 1) = 4.89 millivolts per step.

In general, the voltage resolution is the difference of the voltage ranges divided by the number of steps minus one.

8.5   Watch-Dog Timer

The watch-dog timer built-in to the PIC runs with its own RC oscillator and has a typical minimum time-out period of 7 milliseconds which in general is too short an amount of time. There is a programmable prescaler available that can multiply this period by 128 to give a total time-out period of 850 milliseconds, which is very good for most applications.

The software must reset the watch-dog timer before it overflows and resets the PIC. This is done with the "CLRWDT" instruction. If the watch-dog timer *does* reset the

PIC, there is a "STATUS" bit, "/TO", which indicates that this has occurred. If "/TO" = Zero (=0), the watch-dog timer has reset the PIC.

The watch-dog timer can be enabled or disabled using the "CONFIGURATION" word or in the down-loading process. The watch-dog timer should *NEVER* be disabled! Also, *NEVER* use an Interrupt Service Routine to reset the watch-dog timer!

The user can select the watch-dog timer prescaler by using the "PSA" bit in the "OPTION_REG" register. This is done by setting the "PSA" (=1) and selecting the amount of scaling desired as:

PS2, PS1, PS0	Scale Ratio (WDT)
(0,0,0)	1:1
(0,0,1)	1:2
(0,1,0)	1:4
(0,1,1)	1:8
(1,0,0)	1:16
(1,0,1)	1:32
(1,1,0)	1:64
(1,1,1)	1:128

The prescaler may also be configured to work with the Timer0 module. However, you cannot use it for both the watch-dog timer *and* the Timer0 module at the same time. The better choice, in my opinion, is to keep the prescaler tied to the watch-dog timer since this greatly improves the watch-dog timer.

8.6   Timer0

Timer0 is an 8-bit counter/timer which can be driven from the Fosc/4 clock, a prescaled Fosc/4 clock, or an external pin source called, "T0CKI", which is shared with Port A, RA4. Timer0 is readable and write-able and its output can generate an interrupt, "T0IF". If Timer0 is used as a counter (with "T0CKI" input), the user can select if it should count on the rising edge or the falling edge.

The programmable prescaler may be used on either the timer mode or the counter mode. This prescaler is the same one that is used by the watch-dog timer and it cannot be used by both Timer0 *and* the watch-dog timer at the same time.

Since the "T0CKI" input is shared with the Port A, RA4 pin, the user must set-up Port A, RA4 as an "input" with TRISA, if the counter mode is to be used.

Whenever the Timer0 register, "TMR0", is written to, there is an "inhibit" on its incrementing process for the next two instruction cycles. For example, if the timer

overflows and we want to re-initialize it so that it will reach an effective count of 250, we must write a value of "8" instead of "6" to allow for the extra delay. (This does not apply if the prescaler is set to a ratio larger than two since its effects will not be seen.)

The prescaler is selected by using the "PSA" bit in the "OPTION_REG" register. The "PSA" bit is cleared (=0) and the scale ratio is set as:

PS2, PS1, PS0	Scale Ratio (Timer0)
(0,0,0)	1:2
(0,0,1)	1:4
(0,1,0)	1:8
(0,1,1)	1:16
(1,0,0)	1:32
(1,0,1)	1:64
(1,1,0)	1:128
(1,1,1)	1:256

To set Timer0 as a "timer" do "BCF   OPTION_REG,T0CS". To set Timer0 as a "counter" use "BSF". In the "counter" mode do "BCF   OPTION_REG,T0SE" to select counting on the falling edge and do "BSF" to select counting on the rising edge.

Let's look at an example program that uses the Timer0 module as a "timer" that serves as a time-base and generates an interrupt every four milliseconds. The prescaler is used at a ratio of 1:32.

```
                LIST   P=16F877
                INCLUDE "P16F877.INC"

FLAGS:          EQU        0x20            ; User flags, use bit 0
                ORG        0x0000
                GOTO       INIT

                ORG        0x0004
                GOTO       TIMER0_ISR

                ORG        0x0006
INIT:
                ----- Do Other Inits Prior to Doing Timer0 -----
                BANKSEL    OPTION_REG       ; Bank 1
                MOVLW      0x04             ; T0 Prescale, Timer, 1:32
                MOVWF      OPTION_REG
                BANKSEL    PORTB            ; Bank 0
                MOVLW      D'131'           ; Set count for 125
                MOVWF      TMR0
                MOVLW      0xA0             ; Enable T0 & Global INTs
                MOVWF      INTCON
```

```
                CLRF            FLAGS

        MAIN:

                        ------ Do Program Steps -----
                        ------ Test For "FLAGS, Bit 0" ------

                        ORG             0x0200
        TIMER0_ISR:
                        ------ Save Registers ------(See Appendix F)----
                        MOVLW           D'131'                  ; Reset Count for 125
                        MOVWF           TMR0
                        BCF             INTCON,T0IF             ; Reset T0 INT Flag
                        BSF             FLAGS,0                 ; Set User Flag: INT Occur
                        ------ Restore Registers -------
                        RETFIE                                  ; Return from INT
                        END
```

8.7   Timer1

Timer1 is a 16-bit counter/timer with its own dedicated prescaler and an option to use an external oscillator in place of the Fosc/4 clock. There is also an option to synchronize the external clock to the internal clock, if it is so desired. Timer1 can generate interrupts, if they are enabled. The two Timer1 counter registers, "TMR1H" and "TMR1L", are the most significant and least significant bytes, respectively, and are both readable and write-able.

To use Timer1, it must first be turned "on" by doing, "BSF   T1CON,TMR1ON". The "counter" vs. "timer" mode may be selected as, "BCF   T1CON,TMR1CS", to select the Fosc/4 timer input. Doing "BSF" of the same will select the external input on pin "T1CKI" which is shared with Port C, bit RC0 which must be set-up as an "input" with TRISC.

The external crystal oscillator is connected to the, "T1OSO" and "T1OSI", pins and the oscillator module can be turned "on" by doing, "BSF   T1CON,T1OSCEN". The external clock source may be synchronized by doing, "BCF   T1CON,T1SYNC". The "T1OSO" and "T1OSI" pins are shared with Port C, pins RC0 and RC1. These must be selected as "inputs" with TRISC.

The Timer1 prescale selection is done as:

T1CKPS1:T1CKPS0	Scale Ratio
(0,0)	1:1
(0,1)	1:2
(1,0)	1:4
(1,1)	1:8

Since Timer1 counts in two halves, the timer should be turned "off" when new values are written to the timer registers. Do this as "BCF   T1CON,TMR1ON".

Examples of using Timer1 will be shown in the "Capture" and "Compare" modes of operation.

8.8   Timer2

Timer2 is an 8-bit timer that has a user programmable prescaler and postscaler. There is also a period register, "PR2", which can be set-up and matched to the Timer2 register "TMR2". When a match occurs, "TMR2" is reset and an interrupt can be generated. Timer2 is used mostly for generating precise and exotic clock frequencies.

The "TMR2" and "PR2" registers may be read and written to at any time.

The Timer2 prescale selection is done as:

T2CKPS1:T2CKPS0	Scale Ratio
(0,0)	1:1
(0,1)	1:4
(1,0)	1:16
(1,1)	1:16

The Timer2 postscaler selection is done as:

TOUTPS3:TOUTPS0	Post-Scale Ratio
(0,0,0,0)	1:1
(0,0,0,1)	1:2
(0,0,1,0)	1:3
(0,0,1,1)	1:4
(0,1,0,0)	1:5
(0,1,0,1)	1:6
(0,1,1,0)	1:7
(0,1,1,1)	1:8
(1,0,0,0)	1:9
(1,0,0,1)	1:10
(1,0,1,0)	1:11
(1,0,1,1)	1:12

(1,1,0,0)	1:13
(1,1,0,1)	1:14
(1,1,1,0)	1:15
(1,1,1,1)	1:16

An example program which uses Timer2 will be shown in the "Pulse-Width Modulator" section.

8.9   Capture Mode

The capture mode is a hardware method of measuring pulse-widths and delays between pulses. There are two capture mode modules and both of them use the 16-bit timer, Timer1, for their time-bases. Timer1 must be set-up prior to using the capture mode modules.

The capture mode modules use the "CCP1" and "CCP2" pins, which are shared with Port C, pins RC2 and RC1, respectively. These pins must be set-up as "inputs" with the TRISC register.

The capture mode inputs may look for the following "events" on the input pins:

1) Every falling edge.
2) Every rising edge.
3) Every 4th rising edge.
4) Every 16th rising edge.

When one of these events occurs, the Timer1 values are placed into "CCPR1H" and "CCPR1L" or "CCPR2H" and "CCPR2L" as the high and low bytes of Timer1's count, respectively. Interrupts *must* be used since, if another event occurs before the previous event's data can be read, that previous data will be overwritten and destroyed.

If Timer1 is being used with an external clock, it *must* be set-up as "synchronized" or else the capture process may fail.

The capture mode modules use the "CCP1CON" and "CCP2CON" control register files to select the types of events for each module. The lower four bits of each are used for this selection and are coded as:

CCP1M3:CCP1M0
CCP2M3:CCP2M0   Meaning
        (0,0,0,0)   CCP1 or CCP2 disabled, reset.
        (0,1,0,0)   Every Falling Edge
        (0,1,0,1)   Every Rising Edge
        (0,1,1,0)   Every 4th Rising Edge
        (0,1,1,1)   Every 16th Rising Edge

Switching between these modes or events while the interrupts are enabled may cause false triggering of an interrupt.

Note that the capture, compare, and PWM modules are *not* independent of each other and must not be used without considering the possible conflicts of their usage.

An example program that uses the capture mode and Timer1 is as follows. Suppose that the "CCP1" input (Port C, RC2) is used with the "Every Rising Edge" event to measure pulse-widths.

```
                LIST  P=16F877
                INCLUDE "P16F877.INC"


LOW_HALF:       EQU          0x20  ; Pulse-Width, Low
HIGH_HALF:      EQU          0x21  ; Pulse-Width, High
FLAGS:          EQU          0x22  ; User Flags
EDGE:           EQU          0     ; Flag: First Edge Found
READY:          EQU          1     ; Flag: Data Ready


                ORG        0x0000
                GOTO       INIT
                ORG        0x0004
                GOTO       CCP1_ISR

                ORG        0x0006
INIT:
                BANKSEL    PORTC       ; Bank 0
                CLRF       INTCON      ; Clear All INT Flags
                CLRF       TMR1L       ; Reset Timer1
                CLRF       TMR1H
                MOVLW      0x01        ; Timer1 ON, Prescale = 1
                MOVWF      T1CON
                BANKSEL    TRISC             ; Bank 1
                BSF        TRISC,2     ; RC2 = Input
                BSF        PIE1,CCP1IE ; Enable CCP1 INT Enable Bit
                BANKSEL    PORTC       ; Bank 0
                BCF        PIR1,CCP1IF      ; Reset CCP1 INT Flag
```

```
            MOVLW      0x05          ; Set Event: Every Rising Edge
            MOVWF      CCP1CON
            CLRF       FLAGS         ; Reset User Flags
            MOVLW      0xC0          ; Enable Global & Peripheral
            MOVWF      INTCON        ; Interrupts

    MAIN:
            ----- Look For FLAGS,READY = 1-------
            ----- Use This Data When Ready ----------

    CCP1_ISR:
            ----- Save Registers ------(See Appendix F)-----
            BTFSC      FLAGS,EDGE       ; First Edge Found?
            GOTO       GET_DATA_ISR     ; Yes: Copy Data, Ready
            BSF        FLAGS,EDGE       ; No: Reset Timer1
            CLRF       T1CON            ; Turn Off Timer1
            CLRF       TMR1L            ; Reset Timer1
            CLRF       TMR1H
            BSF        T1CON,TMR1ON     ; Turn Timer1 On
            GOTO       DONE_ISR
    GET_DATA_ISR:
            MOVF       CCPR1L,W         ; Get Low Data
            MOVWF      LOW_HALF
            MOVF       CCPR1H,W         ; Get High Data
            MOVWF      HIGH_HALF
            BSF        FLAGS,READY      ; Set Data Ready
    DONE_ISR:
            BCF        PIR1,CCP1IF      ; Reset INT Flag
            ----- Restore Registers -----
            RETFIE                      ; Return From INT
            END
```

8.10   Compare Mode

There are two compare mode modules that test the value of the 16-bit timer, Timer1, against a 16-bit, user specified, threshold and, when there is a match, an output pin is set/cleared, an interrupt is generated, or a "special event" is triggered. There are two compare mode modules, "CCP1" and "CCP2", which each have distinct "special events". The special event for "CCP1" is to reset and restart Timer1. The special event for "CP2" does the same as for "CCP1" but also starts the ADC. Interrupts may also be generated when they are enabled.

The same rules, registers, interrupts, and pins are used here as in the "capture modes" except that Port C, RC1 and RC2, must be configured as "outputs" with TRISC rather than "inputs".

The "CCP1CON" and "CCP2CON" register bits have different meanings and settings in "compare mode". They are:

CCP1M3:CCP1M0	Meaning / Action
(1,0,0,0)	CCP1 = 1 and CCP1IF = 1
(1,0,0,1)	CCP1 = 0 and CCP1IF = 1
(1,0,1,0)	CCP1IF = 1 (Only)
(1,0,1,1)	CCP1IF = 1 and "Reset/Restart Timer1"

CCP2M3:CCP2M0	Meaning / Action
(1,0,0,0)	CCP2 = 1 and CCP2IF = 1
(1,0,0,1)	CCP2 = 0 and CCP2IF = 1
(1,0,1,0)	CCP2IF = 1 (Only)
(1,0,1,1)	CCP2IF = 1 and "Reset/Restart Timer1" and "Start ADC"

Where "CCP1IF" and "CCP2IF" are the interrupt flags of each module.

As it was stated in the "capture mode": The capture, compare, and PWM modes are *not* independent of each other and must be checked for conflicts of usage.

An example program that uses the compare mode and Timer1 is as follows. Suppose we want to use the "CCP1" module with its special event to produce regular interrupts at intervals of one millisecond. This is a good alternative to using Timer0 as a time-base since the Timer0 prescaler can be freed to work with the watch-dog timer.

```
                LIST   P=16F877
                INCLUDE "P16F877.INC"


FLAGS:       EQU        0x20          ; User Flags
LIMIT_LOW: EQU          0xE8          ; LSB of 1000
LIMIT_HI:    EQU        0x03          ; MSB of 1000
READY:       EQU        0             ; FLAGS, Bit 0, "INT Occurred"


                ORG        0x0000
                GOTO       INIT

                ORG        0x0004
                GOTO       COMPARE_ISR

                ORG        0x0006
        INIT:
```

```
        BANKSEL    PORTC       ; Bank 0
        CLRF       INTCON      ; Reset Main INT Enables
        CLRF       T1CON       ; Turn Timer1 "Off"
        CLRF       TMR1L       ; Reset Timer1
        CLRF       TMR1H
        BSF        T1CON,TMR1ON ; Turn Timer1 "on", Scale = 1
        MOVLW      LIMIT_LOW   ; Set Compare Register Limit
        MOVWF      CCPR1L
        MOVLW      LIMIT_HI
        MOVWF      CCPR1H
        BCF        PIR1,CCP1IF ; Reset Compare Mode INT Flag
        MOVLW      0x0B        ; Set CCP1 Special Event
        MOVWF      CCP1CON
        CLRF       FLAGS       ; Reset User Flags
        BSF        INTCON,GIE  ; Set Global INT Enable
        BSF        INTCON,PEIE ; Set Peripheral INT Enable
        BANKSEL    TRISC       ; Bank 1
        BSF        PIE1,CCP1IE     ; Enable CCP1 Interrupt
        BANKSEL    PORTC       ; Bank 0

MAIN:
        BTFSS      FLAGS,READY  ; INT Occurred?
        GOTO       MAIN
        BCF        FLAGS,READY  ; Yes: Process Loop
        CLRWDT                  ; Reset Watch-Dog Timer
        ------ Do Rest of Program Loop -------
        GOTO       MAIN

COMPARE_ISR:
        BSF        FLAGS,READY     ; Set INT "Ready" Flag
        BCF        PIR1,CCP1IF     ; Reset CCP1 INT Flag
        RETFIE                     ; Return From INT
        END
```

Notice that the registers did not need to be saved and restored as before since the special event automatically resets and restarts Timer1. No manipulations of the W register and the STATUS flags were needed.

8.11   Pulse-Width Modulation (PWM)

There are two PWM modules in the PIC and both can serve as ten-bit Digital-to-Analog Converters (DACs). Their outputs are on the "CCP1" and "CCP2" pins, or, PortC, RC2 and RC1, respectively. These must be set-up as "outputs" using TRISC.

The PWM modules use Timer2 rather than Timer1. Also, the lower four selection-bits of "CCP1CON" and "CCP2CON" must be set to (1,1,0,0) to select the PWM modes.

In operation, TMR2 is compared with the limit set in PR2. To work over a full bit-range, PR2 = 0xFF. For a given value of Fosc and a desired PWM frequency, this full bit-range may not be possible but try it first as a baseline.

PWM Period = ((PR2) + 1)*4*Tosc*(TMR Prescale).

If Fosc = 4 MHz, the PWM Period is 256 microseconds. This gives a PWM frequency of 3.9 kHz, which may be acceptable for some applications, but it is not acceptable for speech output. Speech output needs a PWM frequency of about 8 kHz.

If PR2 = 127, the PWM frequency = 7.8 kHz, which is good enough for speech outputs. However, the bit-range is reduced to only 9 bits, but this is still very good quality for sound.

To send an output sample in PWM1, enter the MSB value in "CCPR1L" and the LSBs in "CCP1CON<5,4>". When the PWM1 is ready for the next data to send, it copies these values into the "CCPR1H" register automatically. PWM2 works in a similar way but with "CCPR2L" and "CCP2CON<5,4>". The PWM period and resolution are the same as for PWM1.

An example program which uses PWM1 and Timer2 is as follows:

```
            LIST   P=16F877
            INCLUDE "P16F877.INC"

USER_DATA:      EQU         0x20            ; Output Data for PWM1
FLAGS:          EQU         0x21            ; User Flags
READY:          EQU         0         ; Flags, Bit 0, Ready to Send

            ORG         0x0000
            GOTO        INIT

            ORG         0x0004
            GOTO        PWM_ISR

            ORG         0x0006
INIT:
            BANKSEL     PORTC               ; Bank 0
            CLRF        INTCON              ; Reset Main INT Enables
            CLRF        USER_DATA           ; Reset Data To Send
            CLRF        PORTC               ; Reset Port C
            MOVLW       0x04                ; Turn "on" TMR2
```

```
        MOVWF     T2CON                ; No Scales
        BANKSEL   TRISC                ; Bank 1
        BCF       TRISC,2              ; RC2 = Output, Use CCP1
        MOVLW     D'127'               ; PR2 = 127
        MOVWF     PR2
        BANKSEL   PORTC                ; Bank 0
        MOVLW     0x0C                 ; Set CCP1 = PWM1
        MOVWF     CCP1CON
        BCF       PIR1,TMR2IF          ; Reset TMR2 INT Flag
        BSF       INTCON,GIE           ; Enable Global INTs
        BSF       INTCON,PEIE          ; Enable Peripheral INTs
        CALL      SEND_PWM             ; Send a Zero
        BANKSEL   TRISC                ; Bank 1
        BSF       PIE1,TMR2IE          ; Enable TMR2 INT
        BANKSEL   PORTC                ; Bank 0

MAIN:
        ----- Get Data To Send  (Put it in W ) ------

WAIT:
        BTFSS     FLAGS,READY          ; INT Occurred?
        GOTO      WAIT                 ; --- No, Wait For INT
        BCF       FLAGS,READY          ; --- Yes, Get More Data
        MOVWF     USER_DATA            ; Set New Data to Send
        GOTO      MAIN

PWM_ISR:
        ----- Save Registers -----(See Appendix F)------
        BSF       FLAGS,READY  ; "INT" = "Data Sent"
        CALL      SEND_PWM
        BCF       PIR1,TMR2IF    ; Reset TMR2 INT Flag
        ----- Restore Registers -----
        RETFIE                         ; Return From INT

SEND_PWM:
        BCF       CCP1CON,4            ; Reset LSBs of Data
        BCF       CCP1CON,5
        BTFSC     USER_DATA,0          ; Copy LSB to LSB, Bit 5
        BSF       CCP1CON,5
        BCF       STATUS,C             ; Rotate Right
        RRF       USER_DATA,W          ; Fit MSB to 7 Bits
        MOVWF     CCPR1L               ; Send MSB
        RETURN
        END
```

8.12   Parallel Slave Port

Port D can be operated as a "parallel slave port" which is a bi-directional, microprocessor data-bus port controlled with the "/CS", "/RD", and "/WR" control lines. These lines are shared with Port E and the TRISE register must configure all of the Port E pins as "inputs". Since Port E also shares its lines with the analog input channels, the "ADCON1" register must be selected per Figure 8-1 so that the Port E lines are as "digital". When the PSP is selected, the TRISD may be ignored – it is not needed.

The PSP selection and status bits are located in the TRISE register. To select the PSP, do "BSF   TRISE,PSPMODE". The PSP may generate interrupts upon a read or a write.

In the discussion that follows about PSP "reads" and PSP "writes", the "read" and "write" are defined relative to the microprocessor that controls the PIC's PSP. The microprocessor "writes" data to the PIC's PSP so that the PIC can "read" its data. That is, a PSP "write" is for the PIC to *get* information from the PSP. The microprocessor "reads" the PIC's PSP so that the PIC can "write" data to the microprocessor. That is, a PSP "read" is the way the PIC *sends* information to the microprocessor. Relative to the PIC, a "write" in an "input" and a "read" is an "output".

A "write" to the PSP occurs when the "/CS" and "/WR" lines are made low by the external device ("microprocessor"). This causes the "IBF" flag in the TRISE register to be set (=1), where "IBF" means "Input Buffer Full". The "write" is completed when either the "/CS" or the "/WR" are made high. This causes the interrupt flag, "PSPIF", to be set, which generates an interrupt, if it is enabled. The "IBF" flag is reset automatically when Port D is read in software. The "IBOV" flag ("Input Buffer Overflow") in the TRISE is set if a *second* "write" is attempted before the first Port D data is read. The "IBOV" flag must be reset in software.

A "read" from the PSP is for sending data from the PIC to the external device. The user does this by writing data to Port D and waiting for the external device to pick it up. Writing data to Port D causes the "OBF" flag in the TRISE register to be set (=1), where "OBF" means "Output Buffer Full". A "read" from the PSP occurs when the "/CS" and the "/RD" lines are made low by the external device. When this occurs, the "OBF" flag is reset immediately so that the PIC's software "knows" that the external device got the data. When either the "/CS" or the "/RD" lines are made high, the interrupt flag, "PSPIF", is set which generates an interrupt if it is enabled.

An example program that uses the PSP is as follows:

```
          LIST   P=16F877
          INCLUDE "P16F877.INC"

IN_DATA:          EQU        0x20        ; Data From PSP "Write"
OUT_DATA:         EQU        0x21        ; Data To PSP "Read"
```

```
                  ORG         0x0000
INIT:

                  BANKSEL     TRISE         ; Bank 1
                  MOVLW       0x06          ; ADCON1 – Select "All Digital"
                  MOVWF       ADCON1
                  MOVLW       0x17          ; Set PSP Mode, Port E=INPUTS
                  MOVWF       TRISE
                  BANKSEL     PORTD         ; Bank 0

MAIN:

                  ----- Get Data to Send -----
                  BANKSEL     TRISE         ; Bank 1
                  BTFSC       TRISE,OBF  ; Got Data?
                  GOTO        NOT_READY_YET1
                  BANKSEL     PORTD         ; Bank 0
                  BTFSS       PIR1,PSPIF  ; "Read" Complete?
                  GOTO        NOT_READY_YET1
                  MOVF        OUT_DATA,W  ; Ready to Send Next Data
                  MOVWF       PORTD
                  BCF         PIR1,PSPIF  ; Reset INT Flag
                  GOTO        WHEREVER1


                  ----- Prepare to Get New Data -----
                  BANKSEL     TRISE         ; Bank 1
                  BTFSS       TRISE,IBF   ; New Data In?
                  GOTO        NOT_READY_YET2
                  BANKSEL     PORTD         ; Bank 0
                  BTFSS       PIR1,PSPIF  ; "Write" Complete?
                  GOTO        NOT_READY_YET2
                  MOVF        PORTD,W     ; Get Data
                  MOVWF       IN_DATA
                  BCF         PIR1,PSPIF  ; Reset INT Flag
                  GOTO        WHEREVER2

                  END
```

8.13   EEPROM Data Memory

The PIC has 256 bytes of EEPROM that the user can read and write using software. This is ideal for long-term data since the EEPROM is a non-volatile memory. The data EEPROM can support up to one-hundred-thousand "erase/write" data cycles. The reading of the EEPROM data is fast but the worst-case writing-time for one byte is eight milliseconds. A watch-dog timer induced reset can abort the writing process so it is important to make sure the watch-dog timer is set with a time-out period greater than its minimum of seven milliseconds.

Both the data read and data write subroutines can be done in cookbook fashion and are as follows:

```
READ_EEPROM:                    ; Call with Address in W
            BANKSEL    EEADR      ; Bank 2
            MOVWF      EEADR      ; Set Address
            BANKSEL    EECON1     ; Bank 3
            BCF        EECON1,EEPGD  ; Do "Data EEPROM"
            BSF        EECON1,RD     ; Start "Read"
            BANKSEL    EEDATA     ; Bank 2
            MOVF       EEDATA,W   ; Get Data in W
            BANKSEL    PORTB      ; Bank 0
            RETURN
```

The data write subroutine uses two RAM locations: "ADDR" and "VALUE" to hold the address and the data value to write, respectively. Reset the watch-dog timer and disable all interrupts before calling this subroutine. This routine returns a value of zero in the W register if the "write" was successful.

```
WRITE_EEPROM:
            BANKSEL    EECON1     ; Bank 3
            BTFSC      EECON1,WR ; Is Previous "Write" Done?
            RETLW      0x01       ; No, Return W = Non-Zero
            BANKSEL    EEADR      ; Bank 2
            MOVF       ADDR,W     ; Get Address
            MOVWF      EEADR      ; Set Address
            MOVF       VALUE,W    ; Get Data to Write
            MOVWF      EEDATA     ; Set Data to Write
            BANKSEL    EECON1     ; Bank 3
            BCF        EECON1,EEPGD  ; Data EEPROM Select
            BSF        EECON1,WREN   ; Do "Write Enable"
            MOVLW      0x55       ; "Lock & Key" Combinations
            MOVWF      EECON2     ;(****** See Below ******)
            MOVLW      0xAA
            MOVWF      EECON2
```

```
                BSF             EECON1,WR         ; Start "Write" Process
                NOP
                BCF             EECON1,WREN  ; Disable "Writes"
                RETLW           0x00              ; OK, Return W = Zero
```

The code sequence labeled "Lock & Key" may seem confusing at first. It is a sequence of actions selected by Microchip to prevent the possibility of writing data to the EEPROM by mistake. Using this code sequence makes writing data to the EEPROM a conscious and deliberate act. There is nothing special about the code words that are sent to the "EECON2" register. They were chosen arbitrarily by Microchip to do the job. What matters is that these two codes must be sent in this order and one sent immediately after the other. This sequence "unlocks" the "Write-to-EEPROM" process.

8.14   FLASH Program Memory

In a similar way to the Data EEPROM, the FLASH Program Memory can also be read from and written to in software. However, the FLASH can only support one-thousand "erase/write" data cycles. The worst-case writing-time is still the same as the Data EEPROM at eight milliseconds. While the writing process is active, the program memory cannot be accessed for the normal running of the software – the CPU "freezes" until the "write" is complete. Again, you must reset the watch-dog and disable all of the interrupts before calling the "writing" subroutine.

Also, since the FLASH program memory has 13-bit addresses and 14-bit data words, two bytes, each, are needed to hold the address and data for a FLASH read or FLASH write. Assume that the user's registers that hold this data are as:

```
        ADDRH:ADDRL   --- For the Address
        DATAH:DATAL   --- For the Data.
```

Both the FLASH read and the FLASH write subroutines can be done in cookbook fashion. They are as follows:

```
        FLASH_READ:
                BANKSEL    EEADR      ; Bank 2
                MOVF       ADDRL,W    ; Get/Set Low Address
                MOVWF      EEADR
                MOVF       ADDRH,W
                MOVWF      EEADRH
                BANKSEL    EECON1     ; Bank 3
                BSF        EECON1,EEPGD ; Select FLASH Memory
                BSF        EECON1,RD  ; Start "Read" Process
                NOP                   ; Delay two cycles
                NOP
```

```
            BANKSEL    EEDATA       ; Bank 2
            MOVF       EEDATA,W  ; Get Data, Low & High
            MOVWF      DATAL
            MOVF       EEDATH,W
            MOVWF      DATAH
            RETURN


FLASH_WRITE:
            BANKSEL    EEADR        ; Bank 2
            MOVF       ADDRL,W      ; Get/Set Address, High & Low
            MOVWF      EEADR
            MOVF       ADDRH,W
            MOVWF      EEADRH
            MOVF       DATAL,W      ; Get/Set Data to Write
            MOVWF      EEDATA
            MOVF       DATAH,W
            MOVWF      EEDATH
            BANKSEL    EECON1       ; Bank 3
            BSF        EECON1,EEPGD  ; Select FLASH Memory
            BSF        EECON1,WREN  ; "Write" Enable
            MOVLW      0x55         ; "Lock & Key" Combination
            MOVWF      EECON2       ; (**** Same Idea as EEPROM **)
            MOVLW      0xAA
            MOVWF      EECON2
            BSF        EECON1,WR         ; Start "Write" Process
            NOP                          ; Delay two cycles
            NOP
            BCF        EECON1,WREN ; Disable "Writes"
            RETURN
```

8.15   FLASH Code & Data EEPROM Protection

The CONFIGURATION word at address 0x2007 contains bit settings that block read/write access to the FLASH program memory and the Data EEPROM. This is a security function to prevent software piracy. These bits can be specified in the program or in the down-load process. Other options allow restricted areas of FLASH memory to be set-up.

8.16   The CONFIGURATION Word

The CONFIGURATION word is located at the address 0x2007 and may be specified in the assembly language program or specified during the down-load process. It has 14 bits and is broken-down as follows:

Bit 13, Bit 5, "CP1"
Bit 12, Bit 4, "CP0"

CP1:CP0   Code Protection (FLASH)
  (1,1)   Code Protect OFF
  (1,0)   0x1F00 to 0x1FFF Protected
  (0,1)   0x1000 to 0x1FFF Protected
  (0,0)   0x0000 to 0x1FFF Protected

Both sets of bits (13,5) and (12,4) must have the same values.

Bit 11, "DEBUG"
    = 1 = "In-Circuit Debugger Disabled"
    = 0 = "Enabled"

Bit 9, "WRT"
    = 1 = "UnProtected FLASH, Can be written under software control"
    = 0 = "Cannot be written under software control"

Bit 8, "CPD"
    = 1 = Data EEPROM Code Protect Off
    = 0 = Code Protected

Bit 7, "LVP" = "Low Voltage In-Circuit Serial Programming Enable Bit"
    = 1 = "Enabled"
    = 0 = "Disabled"

Bit 6, "BODEN"
    = 1 = "Brown-Out Reset Enabled"
    = 0 = "Disabled"

Bit 3, "/PWRTE"
    = 1 = "Power-Up Timer Enabled" = "Power-On Reset Enabled"
    = 0 = "Disabled"

Bit 2, "WDTE"
    = 1 = "Watch-Dog Timer Enabled"
    = 0 = "Disabled"

Bit 1, Bit 0, "FOSC1" and "FOSC0"

FOSC1:FOSC0	Oscillator Selection Bits
(1,1)	RC Oscillator (No External Components Needed)
(1,0)	HS Oscillator (4 MHz to 20 MHz XTAL)
(0,1)	XT Oscillator (200 kHz to 4 MHz XTAL)
(0,0)	LP Oscillator (32 kHz to 200 kHz XTAL)

8.17   Sleep Modes & Reset Modes

The PIC has a power-saving mode where the CPU can be deactivated only to be re-activated later but without losing RAM data or register data. This is called the "Sleep" mode and it is initiated by executing the "SLEEP" instruction in the user's software. Processor "resets" and interrupts can take the PIC out of "Sleep" mode. These will be discussed shortly.

There are several kinds of processor "resets" in the PIC. A list of these is as follows:

1) Power-On Reset (POR)
2) /MCLR --- Normal Operation
3) /MCLR --- From Sleep
4) Watch-Dog Timer (WDT) --- Normal Operation
5) Watch-Dog Timer (WDT) --- From Sleep
6) Brown-Out Reset (BOR)

The "power-on reset" can be enabled to wait until the DC power has come up to a safe level before initializing the PIC. Likewise, the "brown-out reset" can be enabled to look for drops in the DC voltage powering the PIC and cause a reset to occur. The "power-on" reset, the "brown-out" reset, and the watch-dog timer are all enabled or disabled from the CONFIGURATION word. The "/MCLR" reset pin (pin 1) is held high with a pull-up in normal operation.

Each of these resets has "flag" or "status" bits that allow the user to detect which of these reset has occurred from software. These bits are located in the "PCON" and "STATUS" registers. A table of their states and meanings is as follows:

/POR	/BOR	/TO	/PD	Meaning/ Significance
(0)	(x)	(1)	(1)	Power-On Reset
(0)	(x)	(0)	(x)	Illegal
(0)	(x)	(x)	(0)	Illegal
(1)	(0)	(1)	(1)	Brown-Out Reset
(1)	(1)	(0)	(1)	Watch-Dog Timer Reset
(1)	(1)	(0)	(0)	Watch-Dog Timer, Wake From Sleep
(1)	(1)	(u)	(u)	/MCLR Reset, Normal Operation
(1)	(1)	(1)	(0)	/MCLR or INT, Wake From Sleep

Where "x" = "Don't Care" and "u" = "unchanged". The program-counter's value on a reset is 0x0000. The PC's value on a wake-up is the increment of its value when the PIC entered "Sleep".

The interrupts which can awaken the PIC from "Sleep" when they are enabled are as follows:

1) External INT pin
2) Parallel Slave Port (Read/ Write)
3) Timer1
4) CCP1/CCP2 Special Events
5) Serial Peripheral Interface (Chapter 9)
6) SPI / I2C Slave Mode (Chapter 9)
7) USART (Chapter 9)
8) ADC
9) EEPROM Writes

Chapter 9:   PIC Peripherals, Serial Communications Ports

9.0   Chapter Summary

Section 9.2 discusses the USART in all of its modes. Section 9.3 covers the SPI master mode. Section 9.4 covers the SPI slave mode. Section 9.5 covers the I2C in two examples of its master and slave modes.

9.1   Introduction

The PIC has three major serial communications modes:

1) USART
The USART is a "Universal Synchronous/Asynchronous Receiver/Transmitter". It performs RS-232 serial communications with the IBM-PC serial port when it is operated in its asynchronous mode. It can also work in a master or slave synchronous mode but only in a half-duplex form.

2) Master Synchronous Serial Port, Serial Peripheral Interface
The MSSP/SPI mode is a simple 8-bit serial input/output used for working with shift-registers and other simple serial interfaces. It is *not* used for RS-232. Like the USART, it also has a master or slave mode.

3) Master Synchronous Serial Port, Inter-Integrated Circuit
The MSSP/I2C mode is a more complex serial communications mode that is supported by many off-the-shelf integrated circuits. It is intended for more complex systems where there are several master mode devices and many slave mode devices. There is a complicated communications protocol that links each master to each slave. It is *not* used for RS-232.

9.2   USART (Overview)

The USART performs RS-232 serial communications. This can be done with another PIC, a microprocessor serial port, or the IBM-PC's serial port.

The USART can operate with 8-bit data or 9-bit data. It may be configured to work in one of three modes:

1) Asynchronous (Full-Duplex)
2) Synchronous, Master (Half-Duplex)
3) Synchronous, Slave (Half-Duplex)

The "TXSTA" register governs the status and control of the transmitter, while the "RCSTA" register governs the same for the receiver. The data to transmit is placed in the "TXREG" and the received data is placed in the "RCREG" register. There is also the "SPBRG" register which is used to select the baud rate ("data clock-speed").

Interrupts may be generated by both the transmitter and the receiver. These will be shown later as they are needed.

9.2.1   USART (Asynchronous Mode, Full-Duplex)

The first step in using the USART in the asynchronous mode is to activate the USART by setting (=1) the "SPEN" bit in the "RCSTA" register. The Port C pins, RC6 and RC7, must be configured with the TRISC register as "output" and "input" respectively. Setting the "SPEN" bit enables the "TX" pin (RC6) as the transmitted data output and the "RC" pin as the received data input.

Next, the "SYNC" bit of the "TXSTA" register must be cleared (=0) to select the "Asynchronous" mode.

Eight-bit vs. nine-bit transmission and reception is done with the "TX9" bit of the "TXSTA" register and the "RX9" bit of the "RCSTA" register, respectively. Setting each (=1) enables the nine-bit operation while clearing them (=0) enables the eight-bit operation. If the nine-bit operation *is* enabled, the ninth transmitted data bit, "TX9D", must be set-up in the "TXSTA" register, and the ninth received data bit, "RX9D", can be found in the "RCSTA" register.

The baud rate must be selected next. There are two speeds for the baud rate model. The user enters a value "X" in the "SPBRG" register according to the formulas:

Low-Speed Baud Rate = Fosc / (64 * (X + 1))
High-Speed Baud Rate = Fosc / (16 * (X + 1)).

The "BRGH" bit in the "TXSTA" register selects between these "low-speed" and "high-speed" models. If BRGH = 1, the "high-speed" is selected and BRGH = 0 selects the "low-speed". The selection of these speeds is somewhat arbitrary in that one or the other may give a better approximation to the desired baud rate, and this is really what counts.

For example, suppose we want a baud rate of 19200 and the Fosc = 4 MHz. For the "low-speed", this gives SPBRG = 2 and the approximate baud rate = 20833, which has a percent error of 8.5%. For the "high-speed", this gives SPBRG = 12 and the

approximate baud rate = 19230 which has a percent error of 0.16%. Therefore, the "high-speed" selection is the better choice.

Last, the receiver must be enabled by setting (=1) the "CREN" bit in the "RCSTA" register.

The overall data format for the USART in the asynchronous mode is transmission and reception with the least-significant-bit first, in a non-return-to-zero (NRZ) format, with the bit sequence as "start-bit, data-bits, stop-bit", and with no internally-generated parity bits.

The transmitter is enabled by setting (=1) the "TXEN" bit in the "TXSTA". To send data for output, write the data to the "TXREG" register. (If the nine-bit mode is set, the "TX9D" data bit must be set-up first and then write the other part, the byte, to the "TXREG" register.

There is a "transmitter" interrupt flag, "TXIF", which is set when the "TXREG" register is empty and waiting for the next byte to send. That is, when a data byte is placed in the "TXREG" register, the byte is shifted out, and, when the "TXREG" is empty, the interrupt flag is set. The "TXIF" flag cannot be reset in software. It is only reset by writing another byte to the "TXREG" register. If there is no more data to transmit, the interrupt process can only be disabled by disabling the "TXIE" interrupt-enable bit.

To receive a data byte, the "receive" interrupt flag, "RCIF", is set when a new byte is received, and will generate an interrupt if it is enabled. The "RCIF" bit is automatically reset when the software reads the data from the "RCREG" data register. (If the nine-bit mode is set, the "RX9D" data bit in the "RCSTA" register must be read *before* reading the "RCREG" data.)

Another feature that is available in the nine-bit asynchronous mode is "automatic address detection". This is used for "multi-PIC" or "multi-device" operations where each device has a nine-bit "address" and it will respond to the transmitting PIC only when the address matches the address sent to it.

To select this mode of operation, set-up the PIC in nine-bit asynchronous mode and set (=1) the "ADDEN" bit in the "RCSTA" register. Also enable the receiver interrupt with the "RCIE" bit (RCIE = 1). When the interrupt occurs, read the nine data bits, interpret them as an address, and see if the address matches the user-defined address. If it does, clear (=0) the "ADDEN" bit to allow the software to read the data to follow. The address is distinguished from the data by setting the ninth-bit of the address to one (=1). The ninth-bit of the data will be zero (=0).

It should be noted that the asynchronous USART mode is halted if the CPU enters the "Sleep" state.

There are two receiver-error condition bits that need to be discussed. These are the "OERR" bit meaning "Overrun Error" and "FERR" meaning "Framing Error". Both of these bits are in the "RCSTA" register.

The "RCREG" register where the data comes in is double-buffered so that it can store two bytes in succession. If a third data byte comes in, without the first two having been read, the "OERR" bit will be set (=1) to indicate "Overrun Error". When this happens, the third data byte is lost and the whole receive process is inhibited. To correct and reset this error, the "RCREG" register must be emptied and the "OERR" can be reset only indirectly by doing:

```
        BCF    RCSTA,CREN
        BSF    RCSTA,CREN.
```

The "Framing Error" condition, indicated by the "FERR" bit is set (=1) if a received data byte has an illegal "Stop" bit indicating that the data is illegal. This error condition does not inhibit the receive process and does not need to be cleared. A new "FERR" value will appear when the "RCREG" data register is read. Therefore, check the "FERR" bit *before* reading the "RCREG" to get the current "FERR" value.

The example program that follows is for an 8-bit, asynchronous USART mode with receiver interrupts, transmitter polling, and a baud rate of 19200.

```
              LIST   P=16F877
              INCLUDE "P16F877.INC"

RCV_DATA:     EQU       0x20        ; Received Data
FLAGS:        EQU       0x21        ; User Flags
READY:        EQU       0           ; Flag: RCV Data Ready
FRAME:        EQU       1           ; Flag: Frame Error

              ORG       0x0000
              GOTO      INIT

              ORG       0x0004
              GOTO      RCV_ISR

              ORG       0x0006
INIT:
              BANKSEL   PORTC       ; Bank 0
              CLRF      PORTC
              BCF       PIR1,RCIF   ; Reset RCV INT Flag
              MOVLW     0x80        ; Enable USART, 8-Bit
              MOVWF     RCSTA       ; --- RCV Mode
```

```
        BANKSEL    TRISC              ; Bank 1
        BCF        TRISC,6            ; RC6 = Out
        BSF        TRISC,7            ; RC7 = In
        MOVLW      D'12'              ; Set Baud Rate = 19200
        MOVWF      TXSTA
        BSF        PIE1,RCIE          ; Set RCV Int Enable Bit
        BANKSEL    PORTC              ; Bank 0
        BSF        INTCON,PEIE        ; Enable Peripheral Ints
        BSF        INTCON,GIE         ; Enable Global Ints
        BSF        RCSTA,CREN         ; Enable RCV
        CLRF       FLAGS              ; Reset User Flags
        BSF        TXSTA,TXEN         ; Enable XMTR

MAIN:
        ------ When Ready To Send Data ---------
        ------ Put Data in (W) ----------------------

        MOVWF      TXREG              ; Send Data
WAIT:

        BTFSS      PIR1,TXIF          ; Wait for XMT to Finish
        GOTO       WAIT

        ------ If Data is Received --------------------
        BTFSS      FLAGS,READY        ; Is Data Ready to Read?
        GOTO       NO_DATA_YET
        MOVF       RCV_DATA,W         ; Get the Data
        BCF        FLAGS,READY

RCV_ISR:
        ------ Save Registers ------- (See Appendix F)
        BCF        FLAGS,FRAME        ; Reset Previous FERR
        BTFSC      RCSTA,FERR         ; Check for Frame Error
        BSF        FLAGS,FRAME        ; Set "FRAME" if FERR=1
        MOVF       RCREG,W            ; Get New RCV Data
        MOVWF      RCV_DATA           ; Store in User's Data
        BSF        FLAGS,READY        ; Set User's Data Ready
        ------ Restore Registers ---------------------
        RETFIE                        ; Return From Int
        END
```

9.2.2   USART (Synchronous, Master Mode)

Both the synchronous master mode and the synchronous slave mode run the USART as "half-duplex" meaning that the data cannot be transmitted and received at the same time. The primary difference between the master mode and the slave mode is that the master mode generates the serial clock while the slave mode receives the external serial clock.

Many features, bits, and conditions of the synchronous master mode are similar to the asynchronous mode. This section will illustrate only the differences between these modes.

The baud rate generator works with a model and formula different from the asynchronous mode and the "BRGH" bit, which was the speed control, has no meaning. The baud rate is:

Baud Rate = Fosc / (4 * (X + 1))

Where "X" is the value written into the "SPBRG" register. This allows for much higher baud rates.

The "Sleep" mode halts the synchronous master mode. The "Address Detection" mode is not available in the synchronous master mode. The Port C pins, RC6 and RC7, are set up as before. The clock (CK) is sent on RC6 while the data is both sent and received one way at a time, on RC7 (DT).

The "CSRC" bit and the "SYNC" bit, both of the "TXSTA" register, must be set (=1) for the synchronous master mode.

The operation of the USART in the synchronous master mode is more complicated due to the restriction of doing only half-duplex communications. The "TXEN" bit of the "TXSTA" register and the "CREN" bit of the "RCSTA" register must *not* both be set (=1) at the same time – only one or the other is to be set (=1) at any one time. To transmit data, the "TXEN" bit must be set (=1) and the "CREN" bit must be cleared (=0). To receive data, the "TXEN" bit must be cleared (=0) and the "CREN" bit must be set (=1). All other things are equal. In the receive mode, the data is sampled on the falling edge of the clock and, in transmit mode, the data is shifted on the rising edge of the clock and is stable on the falling edge.

One exception is the possibility of receiving one single byte during the transmission of (typically) a stream of transmitted data bytes. This is done by setting the "SREN" bit (=1) of the "RCSTA" register while the "TXEN" bit of the "TXSTA" register is set (=1). When a single byte is received, the "SREN" bit is cleared (=0) automatically.

9.2.3   USART (Synchronous, Slave Mode)

This section will illustrate the differences between the synchronous slave mode and the other two USART modes.

Since the synchronous slave mode does not generate a serial data clock, but receives it externally, there is no need to set-up the "SPBRG" register to generate baud rates. The Port C pins, RC6 and RC7, must *both* be set-up as "inputs" using the TRISC" register. The "CSRC" bit of the "TXSTA" register must now be cleared (=0) for the synchronous slave mode.

The "single byte receive" option using the "SREN" bit is disabled in the slave mode. The "either-or" nature of the "TXEN" and "CREN" bits is still the same as in the synchronous master mode.

The reception or transmission of data in the slave mode can awaken the CPU from "Sleep".

9.3   Serial Peripheral Interface (SPI, Master Mode)

The SPI master mode allows 8-bit data to be synchronously transmitted and received at the same time. This data transfer happens on three pins as follows:

Serial Data Out = SDO = RC5 = Output
Serial Data In   = SDI = RC4 = Input
Serial Clock   = SCK = RC3 = Output.

Where the Port C pins, RC5, RC4, and RC3, are set-up with the TRISC register as above.

The "SSPSTAT" and "SSPCON" registers are used to control the SPI module, while the "SSPBUF" register is used for the input and output data.

The first step in setting-up the SPI in master mode is to set the lower four bits of the "SSPCON" register as follows:

SSPM3:SSPM0	Function / Meaning
(0,0,0,0)	SPI, Master, SCK = Fosc / 4
(0,0,0,1)	SPI, Master, SCK = Fosc / 16
(0,0,1,0)	SPI, Master, SCK = Fosc / 64
(0,0,1,1)	SPI, Master, SCK = Timer2 / 2

Next, the "CKP" bit in the "SSPCON" register must be set-up along with the "CKE" and "SMP" bits of the "SSPSTAT" register. This is done according to the waveforms of Figure 9-1. An example of the issues involved with this selection will be given later.
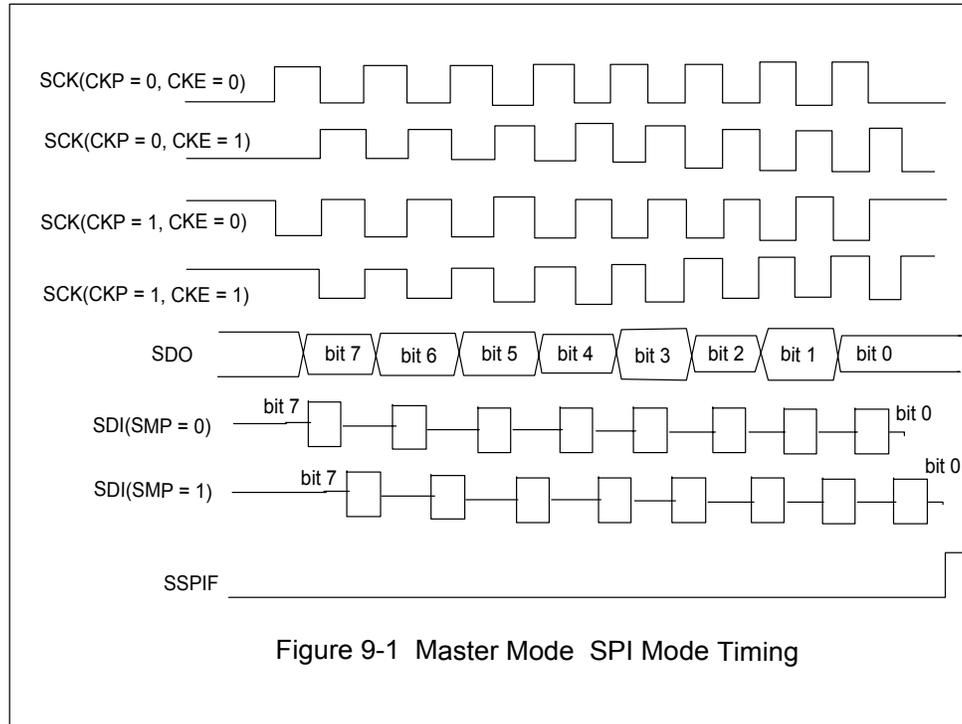
Figure 9-1 Master Mode SPI Mode Timing

The last step is to set (=1) the "SSPEN" bit of the "SSPCON" register to enable the MSSP/SPI module.

Sending data for output (on the SDO line) is done by writing the data to the "SSPBUF" register. Getting data from the input (on the SDI line) is done by doing a "dummy" write or a "real" write to the "SSPBUF" register. It is impossible to read data for input *without* writing data for output! When the data transfer process is finished, the "SSPIF" interrupt-flag is set (=1). If the corresponding interrupt-enable bit, "SSPIE", is set (=1), an interrupt will be generated.

Now let's look at some hardware interfacing issues and the selection of bits per Figure 9-1. A simple input/output circuit is shown in Figure 9-2. A "serial-in, parallel-out" (SIPO) shift-register (74HC164) is used to capture the SPI's serial output data. A "parallel-in, serial-out" (PISO) shift-register (74HC165) is used to feed the SPI's serial input pin (SDI). Notice that the "clock" or "shift" input of the 74HC165 runs from the

SPI's clock (SCK) through an inverter. Both of the shift-registers "clock" on the rising-edge.

Figure 9-2 Serial-Out/Serial-In with the 74HC164 & 74HC165
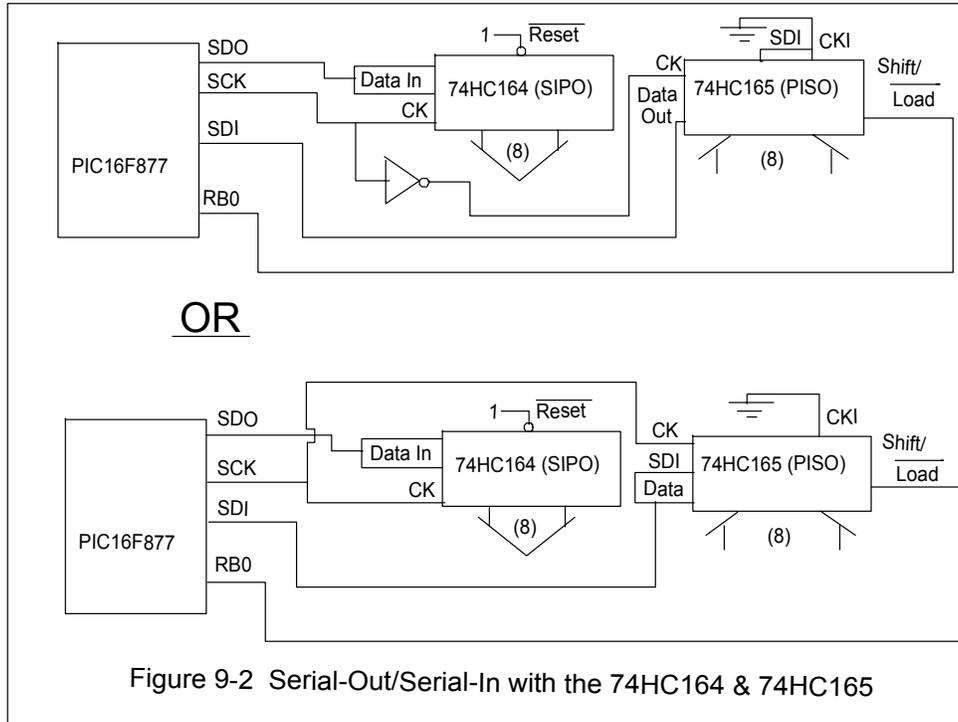
The selection bits for this circuit are:

CKP = 0, CKE = 1, and SMP = 0.

This produces the second "SCK" waveform from the top of Figure 9-1. This is ideal for the "output" part since the "SDO" data is stable when the "SCK" makes a rising edge. If the 74HC165 is used without the inverter on its clock, a potential problem exists. The rising edge would then be used to sample the input data and shift the shift-register at the same time! In general this is a bad design practice. The problem can be fixed by using the inverter as above, or by selecting "SMP = 1" and feeding the 74HC165's serial output line back into its serial input line. The latter would cause the data to be shifted an extra space and the PIC would have to rotate the bit-pattern back into its proper place.

Another circuit situation is shown in Figure 9-3. This circuit uses an "AND" gate to inhibit the clock-input of a SIPO, 74HC164 shift-register to prevent data from being output to it while data from the 74HC165 is being read in.

Figure 9-3 Serial-Out/Serial-In with Gated Clock
To Inhibit Serial Output
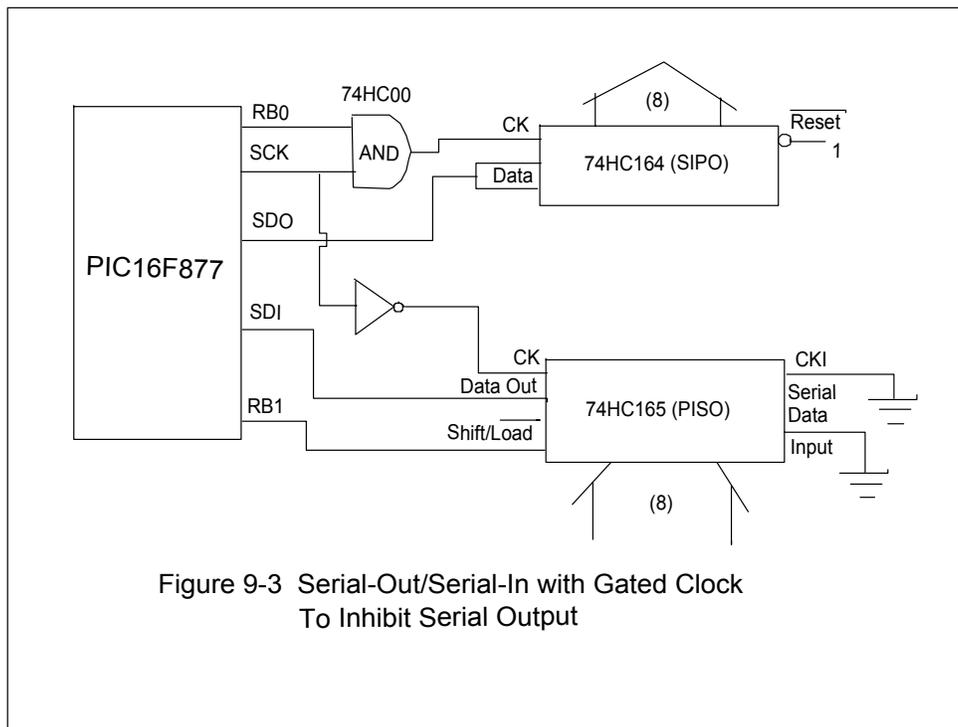
Since both the 74HC164 and the 74HC165 have serial data inputs, they may be cascaded in series (serially) for multiple-byte inputs and outputs.

An example program that uses the SPI master mode and Figure 9-3 is as follows:

```
        LIST   P=16F877
        INCLUDE "P16F877.INC"


DATA_RCV:       EQU        0x20        ; Storage for RCV Data
OUT_ENABLE:     EQU        0           ; Port B, RB0 – Enable Out
SHIFT_LOAD:     EQU        1           ; Port B, RB1 =
                                       ;    1 = SHIFT
                                       ;    0 = LOAD

        ORG        0x0000
INIT:
        BANKSEL    TRISC        ; Bank 1
        MOVLW      0x10         ; RC5 = RC3 = Out
        MOVWF      TRISC        ; RC4 = In
        CLRF       TRISB        ; Port B = Outputs
        BCF        SSPSTAT,SMP  ; SMP = 0
        BSF        SSPSTAT,CKE  ; CKE = 1
```

```
            BANKSEL    PORTC              ; Bank 0

            CLRF       SSPCON             ; SPI, Master, Fosc / 4
            BSF        SSPCON,SSPEN  ; Enable SPI

MAIN:

            ------- Put Data to Send in W ---------
            CALL       SPI_SEND           ; Do Data Read / Write

            CALL       SPI_READ           ; Do Data Read (No Write)

SPI_READ:
            BCF        PORTB,OUT_ENABLE ; Disable Write
            GOTO       SPI_SKIP
SPI_SEND:
            BSF        PORTB,OUT_ENABLE ; Enable Write
SPI_SKIP:
            BCF        PORTB,SHIFT_LOAD       ; "Load"
            BSF        PORTB,SHIFT_LOAD       ; "Shift"
            BCF        PIR1,SSPIF             ; Reset Int Flag
            MOVWF      SSPBUF                 ; "Send" Data

SPI_WAIT:
            BTFSS      PIR1,SSPIF         ; Wait Until Done
            GOTO       SPI_WAIT

            MOVF       SSPBUF,W           ; Get RCV Data
            MOVWF      DATA_RCV           ; & Save It
            BCF        PIR1,SSPIF         ; Reset Int Flag
            RETURN
            END
```

9.4   Serial Peripheral Interface (SPI, Slave Mode)

There are many similarities between the SPI master mode and the SPI slave mode. Only the differences will be stated here. The main difference is that the "SCK" serial-clock line does not generate the clock but receives it as an input on the "SCK" line. Also, there is a fourth serial port line, "/SS", meaning "slave-select" which can be used to activate or deactivate the slave SPI unit, if the option to use it is enabled.

The summary of the SPI serial pins in slave mode is:

Serial Data Out = SDO = RC5 = Output
Serial Data In   = SDI = RC4 = Input
Serial Clock   = SCK = RC3 = Input
/Slave Select   = /SS =   RA5 = Input (TRISA & ADCON1)

The lower four bits of the "SSPCON" register now select as:

SSPM3:SSPM0   Function / Meaning
        (0,1,0,0)   SPI, Slave, /SS Enabled
        (0,1,0,1)   SPI, Slave, /SS Disabled

The "CKP" and "CKE" bits are selected according to the waveforms in Figure 9-4 and Figure 9-5. The "SMP" bit must always be zero (=0). Also if "CKE" is set (=1) then the "slave select" (/SS) *must* be enabled.

Figure 9-4 SPI Mode Timing (Slave Mode, CKE = 0)

133

Figure 9-5 SPI Mode Timing (Slave Mode, CKE = 1)
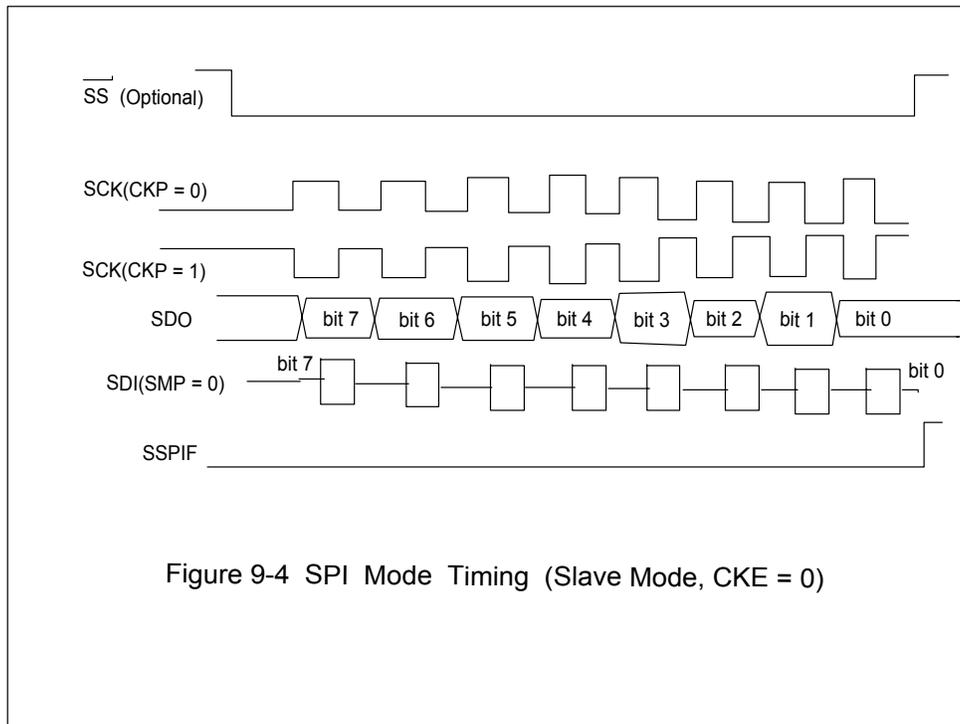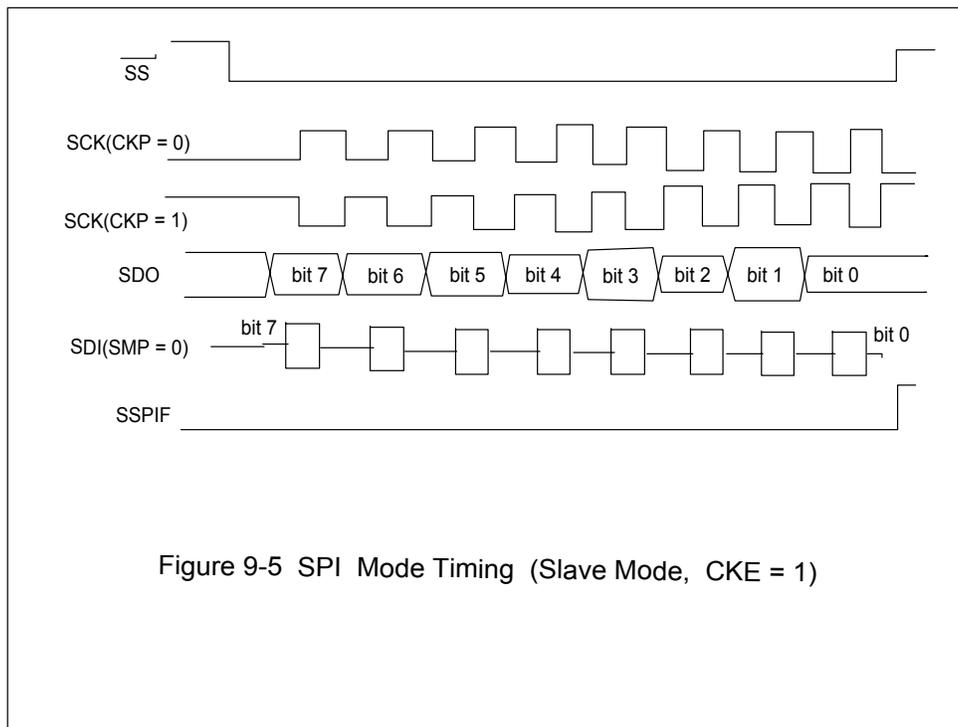
If the "slave select" pin is enabled, the controlling device must set this pin "low" (= 0 = Ground) to activate the slave PIC's SPI module. If the "slave select" line is enabled and is held "high" (= 1 = +5 Volts), the slave PIC will ignore the "SCK" input clock and none of its data will be transferred.

All of the other features of the SPI master mode are the same as in the SPI slave mode.

9.5   I2C System Overview

The I2C system communicates on the "SCL" and "SDA" pins which are shared with the Port C, pins RC3 and RC4, respectively. The "SCL" is the data clock and the "SDA" is the data line. These pins must both be set-up as "inputs" with the TRISC register before configuring the I2C modes. Furthermore external pull-up resistors (minimum resistance is 1.7 K-Ohms) must be attached to each of these pins for the proper operation of the I2C module. All of the above is true for the I2C module in all of its modes, master or slave.

In the I2C system each slave device has a unique "address" code which identifies it for the master device when the master device wants to access it. Many slave devices may be used in the system and there may be several *master* devices, too. They all share the same "SCL" and "SDA" lines which are "open-drain" outputs along with sensors for "inputs". This makes bi-directional information transfers possible at any time.

The slave's addresses may be either 7-bits long or 10-bits long depending on the devices to be used or its settings. The PIC I2C module supports both lengths in either of the master or slave modes.

The slave devices in the I2C system cannot *initiate* a data transfer to the master(s). It can only read or write data to the master when the master calls it and only one master can do this at any one time.

Let's look at the I2C communication process in its most general form. Suppose, for example, for simplicity we are working with 7-bit slave addresses.

Suppose that the master device is the PIC in the I2C system operating in the master mode and it wants to write data to a slave. The master PIC first looks for activity on the "SDA" and "SCL" lines to see if *another* master device is using the system. If and when there is no activity, the master sends out nine (9) clock pulses on the "SCL" line and sends out eight (8) bits on the "SDA" line. The eight bits are the 7-bit slave address and a "read/write" bit which is set (=1) for a "read" or reset (=0) for a "write". Here it is reset (=0). Then the master PIC holds the "SDA" line "high" (=1) and looks for an "acknowledge" state from the slave also on the "SDA" line (this is possible since the "SDA" line uses "open-drain" outputs). If the slave does *not* acknowledge the master, it will hold the "SDA" line "high" (=1) and the master knows that the slave refuses the master's attempt to "write". If the slave *does* acknowledge the master, it will bring the "SDA" line to ground, or "low" (=0), and the master will proceed to send its data (more on this later).

When the slave device (suppose it is a PIC in slave mode) receives the 7-bit address and the "read/write" bit it checks the address contained in the "SSPADD" register which was set-up by the slave's user's software. If there is a match, an "acknowledge" is sent; if not, no "acknowledge" is sent.

If the slave gives an "acknowledge" to the master, the master sends out nine (9) "SCL" clock pulses and the eight data bits to be sent on the "SDA" line. The ninth clock pulse is used to sense *another* "acknowledge" state from the slave. If the slave does not "acknowledge" the master, the data "write" process is stopped, and both the master and the slave go into an "idle" state. If there is an "acknowledge" from the slave, the above process repeats, but sending a new address is not necessary.

Suppose that the master wants to get data ("read") from the slave. The whole process is nearly identical, but with three (3) exceptions. The first is that the "read/write" bit is set (=1) for doing a "read". The second is that when the slave gives the

"acknowledge" when it gets its address code from the master, the slave holds the "SCL" line "low" (=0) to inhibit the master from sending out clock pulses on the "SCL" line. These are the clock pulses the master uses to synchronize the data "read" from the slave. The master is inhibited until the slave is ready to send its data. When it is ready, it releases the "SCL" line by letting it go "high" (=1). The third is that *master* is now the one that sends the "acknowledge" out on the ninth master clock pulse for the slave to receive. If there is an "acknowledge" from the master, the slave gets ready to send another byte. Again, sending a new address is not necessary.

In both the "write" and "read" processes a "useless" or "dummy" byte is sent after the last "valid-data" byte "acknowledge" is sent so that it can be refused and the data transfer process will then stop.

The I2C module uses five user-accessible registers for its data, status, and control functions:

1) SSPCON --- Control #1
2) SSPCON2 --- Control #2
3) SSPSTAT --- Status
4) SSPBUF --- Transmit and Receive Data
5) SSPADD --- Slave Address or Master Baud-Rate Setting

The first step in setting-up the I2C module is to set-up Port C pins RC3 and RC4 as "inputs" by using the TRISC register. Next, set-up the lower four bits of the "SSPCON" register as:

SSPM3:SSPM0	Function / Meaning
(0,1,1,0)	I2C Slave Mode, 7-Bit Address
(0,1,1,1)	I2C Slave Mode, 10-Bit Address
(1,0,0,0)	I2C Master Mode
(1,0,1,1)	I2C Firmware Controlled Master Mode / Slave Idle
(1,1,1,0)	I2C FCMM, 7-Bit Addr, Start/Stop Interrupts
(1,1,1,1)	I2C FCMM, 10-Bit Addr, Start/Stop INTs

In the master mode, set-up the baud rate as "X" in the "SSPADD" register as:

Baud Rate = $Fosc / (4 * (X + 1))$

Where the baud rate is usually one of 100 kHz, 400 kHz, or 1 MHz.

In either master or slave modes, set-up the "CKE" bit in the "SSPSTAT" register as:

0 = I2C Input Levels
1 = SMBus Input Levels.

Also, set-up the "SMP" bit which is also in the "SSPSTAT" register, to control the slew-rates of the "SCL/SDA" pins as:

1 = Slew-Rate Disabled (100 kHz or 1 MHz baud rates)
0 = Slew-Rate Enabled (400 kHz baud rate).

Last, set (=1) the "SSPEN" bit in the "SSPCON" register to enable the I2C serial port module.

9.5.1   I2C Slave Mode

The great complexity of the I2C communications module makes it difficult to describe not only the available options but the sequences in which they are used. This is best seen with an example program using the 7-bit I2C in slave mode.

```
                LIST  P=16F877
                INCLUDE "P16F877.INC"

FLAGS:      EQU         0x20           ; User's Flags
RCV_DATA: EQU           0x21           ; Hold for User's Rcved Data
ADDR_LOW: EQU           0x3D           ; Slave's Address (Arbitrary Here)
ACCEPT:     EQU         0       ; Flags: Bit 0 = "Accept Next Data"

                ORG         0x0000
                GOTO        INIT

                ORG         0x0004
                GOTO        SLAVE_ISR

                ORG         0x0006
    INIT:
                BANKSEL     PORTC       ; Bank 0
                CLRF        FLAGS       ; Reset User's Flags
                BANKSEL     TRISC           ; Bank 1
                BSF         TRISC,3     ; Make RC3 & RC4 "Inputs"
                BSF         TRISC,4
                BCF         SSPCON2,GCEN ; Disable "General Call"
                            ; (This would reserve Address = 0 as
                            ; a cause for an interrupt, if enabled)
                BANKSEL     PORTC       ; Bank 0
                MOVLW       0x06           ; Select "7-Bit Slave" Mode
                MOVWF       SSPCON
```

```
        BANKSEL    SSPSTAT    ; Bank 1
        BCF        SSPSTAT,SMP ; Enable Slew-Rate Control
        BCF        SSPSTAT,CKE ; Set I2C Signal Levels
        MOVLW      ADDR_LOW ; Set The Slave's Address
        MOVWF      SSPADD
        BANKSEL    PORTC      ; Bank 0
        BCF        PIR1,SSPIF ; Reset I2C Int Flag
        BANKSEL    TRISC      ; Bank 1
        BSF        PIE1,SSPIE ; Enable I2C Interrupts
        BANKSEL    PORTC      ; Bank 0
        BSF        INTCON,PEIE ; Enable Peripheral Interrupts
        BSF        SSPCON,SSPEN ; Enable I2C MSSP Module
        BSF        INTCON,GIE ; Enable Global Interrupts


MAIN:
        --- Do Program & Wait to be Interrupted ----

SLAVE_ISR:
        --- Save Registers ---(See Appendix F)---
        BANKSEL    SSPSTAT    ; Bank 1
        BTFSC      SSPSTAT,DA ; Is Word = Address?
        GOTO       DO_DATA    ; --- No, Process the Data
        BTFSC      SSPSTAT,RW ; --- Yes, Is This a "Write" Op?
        GOTO       WRITE_DATA ; --- Yes, Do Write

        BANKSEL    PORTC      ; --- No, Prepare To Get Data
        BSF        FLAGS,ACCEPT ; Set Flag to Accept Next
                                          ; Data
        MOVF       SSPBUF,W   ; Discard Address Transmitted
        BCF        SSPCON,SSPOV ; Reset Overflow Flag
ISR_RETURN:
        BCF        PIR1,SSPIF ; Reset I2C Interrupt Flag
        --- Restore Registers ----
        RETFIE                        ; Return From Interrupt

WRITE_DATA:
        BANKSEL    PORTC      ; Bank 0
        MOVF       SSPBUF,W   ; Discard Address Transmitted
        BCF        SSPCON,CKP ; SCL = 0, Inhibit Master
        CALL       GET_DATA_TO_SEND ; W = Data
        MOVWF      SSPBUF     ; Send Data
        BSF        SSPCON,CKP ; Release Master
        GOTO       ISR_RETURN
```

```
DO_DATA:
            BANKSEL    PORTC        ; Bank 0
            BTFSC      FLAGS,ACCEPT ; Accept This Data?
            GOTO       YES_ACCEPT

            MOVF       SSPBUF,W   ; Discard The Data
            BCF        SSPCON,SSPOV ; Reset Overflow (Send
                                         ; ACKN)
            GOTO       ISR_RETURN

YES_ACCEPT:
            MOVF       SSPBUF,W   ; Get The Data
            MOVWF      RCV_DATA   ; Store Data in User's Data Hold
            BCF        FLAGS,ACCEPT ; Refuse Next Data
            BSF        SSPCON,SSPOV ; Set Overflow, No ACKN
                                         ; For Next Time
            GOTO       ISR_RETURN
            END
```

9.5.2   I2C Master Mode

As in the slave mode, the operation of the master mode is best seen with an example program. Assume that we are using the "I2C Firmware Controlled Master Mode with 7-Bit Addresses and Start/Stop Interrupts Enabled" at a baud rate of 100 kHz.

```
            LIST   P=16F877
            INCLUDE "P16F877.INC"


FLAGS:      EQU        0x20         ; User's Flags #1
FLAGS2:     EQU        0x21         ; User's Flags #2
RCV_DATA:   EQU        0x22         ; User's Rcved Data Hold
XMT_DATA:   EQU        0x23         ; Hold For Data to Transmit
CALLED_SLAVE:  EQU     0x24         ; Current Slave's Address

BAUD:       EQU        0x09         ; Value for Baud Rate = 100 kHz
SLAVE1:     EQU        0x78         ; First Slave's Address (Arbitrary)
SLAVE2:     EQU        0x6E         ; 2nd Slave's Address (Arbitrary)
SLAVE3:     EQU        0x2C         ; 3rd Slave's Address (Arbitrary)

; FLAGS ----------------------------------------------------------
DUMMY_WRITE: EQU       0            ; Do a Dummy Write
START_OK: EQU          1            ; "Start" Process is OK
ADDR_ACK EQU           2            ; Address is ACKNed
```

```
MASTER_ACK: EQU        3               ; Master's ACKN is Given


; FLAGS2 -----------------------------------------------------------------
BYTE_FOR_XMT: EQU    0        ; Main Program is Attempting XMT
GET_A_BYTE: EQU      1               ; Main is to Receive a Byte
ADDR_ERROR: EQU      2               ; Address Error – Slave Refused
XMT_DATA_OK: EQU     3               ; Transmitted Data Process OK
WRITE_ERROR: EQU     4               ; Slave Refuses Data-Write
RCV_OK:     EQU      5               ; RCV Data Process OK
BUS_COL:    EQU      6               ; Bus Collision Error Occurred

;-----------------------------------------------------------------------

            ORG         0x0000
            GOTO        INIT

            ORG         0x0004
            GOTO        INT_RESPONSE

            ORG         0x0006
INIT:

            BANKSEL     PORTC        ; Bank 0
            CLRF        FLAGS        ; Reset User's Flags #1
            CLRF        FLAGS2       ; Reset User's Flags #2
            CLRF        INTCON       ; Reset Main INT Flags
            BANKSEL     TRISC        ; Bank 1
            BSF         TRISC,3      ; Make RC3 & RC4 "Inputs"
            BSF         TRISC,4
            BANKSEL     PORTC        ; Bank 0
            MOVLW       0x0E         ; 7-Bit FCMM Start/Stop Ints
            MOVWF       SSPCON
            BANKSEL     TRISC        ; Bank 1
            BCF         SSPSTAT,CKE ; I2C Signal Levels
            BSF         SSPSTAT,SMP ; Slew-Rate Control Disabled
            MOVLW       BAUD         ; Set Baud Rate = 100 kHz
            MOVWF       SSPADD
            BANKSEL     PORTC        ; Bank 0
            BCF         PIR1,SSPIF  ; Reset I2C Interrupt Flag
            BCF         PIR2,BCLIF  ; Reset Bus Collision Int Flag
            BANKSEL     TRISC        ; Bank 1
            BSF         PIE1,SSPIE  ; Enable I2C Interrupts
            BSF         PIE2,BCLIE  ; Enable Bus Collision Interrupts
            BANKSEL     PORTC        ; Bank 0
            BSF         INTCON,PEIE ; Enable Peripheral Interrupts
            BSF         SSPCON,SSPEN ; Enable I2C MSSP Module
            BSF         INTCON,GIE ; Enable Global Interrupts
```

MAIN:

----- Run the Main Program and Look at the FLAGS2 byte -----
----- This is the "Macro-Status" for the I2C Process --------------

To Write to a Slave Device:
1) Check if "BYTE_FOR_XMT" and "GET_A_BYTE" are clear
2) Get the Slave's address and put it in "CALLED_SLAVE"
3) Get the byte for to send, put it in "XMT_DATA"
4) Set the "BYTE_FOR_XMT" flag
5) Call the "SET_START" subroutine
6) Check if "XMT_DATA_OK" flag is set and no error flags are set (Data Sent OK). Then Reset "FLAGS2".
7) Possible Errors Are:
   a) ADDR_ERROR --- Slave does not accept address given
   b) WRITE_ERROR --- Slave does not accept the data
   c) BUS_COL --- A Bus Collision occurred.

To Read from a Slave Device:
1) Check if "BYTE_FOR_XMT" and "GET_A_BYTE" are clear
2) Get the Slave's address and put it in "CALLED_SLAVE"
3) Set the "GET_A_BYTE" flag
4) Call the "SET_START" subroutine
5) Check if "RCV_OK" flag is set and no error flags are set (Received Data OK). Get the Data in "RCV_DATA". Then Reset FLAGS2.
6) Possible Errors Are:
   a) ADDR_ERROR --- Slave does not accept address given
   b) BUS_COL --- A Bus Collision Occurred.

;--

INT_RESPONSE:

---- Save Registers -----(See Appendix F)----
CHECK_INT_FLAGS:
```
        BANKSEL    PORTC      ; Bank 0
        BTFSC      PIR2,BCLIF ; Bus Collision?
        GOTO       BUS_COLLISION
        BTFSC      PIR1,SSPIF ; I2C Interrupt?
        GOTO       DATA_COMMUNICATIONS
        ---- Restore Registers ------
        RETFIE                ; Return from Interrupt
```

```
BUS_COLLISION:
        MOVF       SSPBUF,W   ; Discard SSPBUF Data
        BCF        SSPCON,WCOL ; Reset Error Flags (In I2C)
        BCF        SSPCON,SSPOV
        BANKSEL    SSPCON2    ; Bank 1
        BSF        SSPCON2,PEN ; Send "Stop" Condition Signal
        BANKSEL    PORTC       ; Bank 0
        BSF        FLAGS2,BUS_COL ; Set User Flag: Bus
                                  ; Collision
        BCF        PIR2,BCLIF  ; Reset All INT Flags
        BCF        PIR1,SSPIF
        GOTO       CHECK_INT_FLAGS


;-------------------------------------------------------------------------------

DATA_COMMUNICATIONS:
        BANKSEL    PORTC       ; Bank 0
        BTFSS      FLAGS2,BUS_COL ; Was Previous Bus
                                  ;  Col?
        GOTO       CHECK_STATUS2 ; No

        CLRF       FLAGS   ; Return, Main Software Must Clear
        BCF        PIR1,SSPIF   ; The Previous Bus Collision
        GOTO       CHECK_INT_FLAGS ; Condition Before New
                                   ; Communications Will Start.

CHECK_STATUS2:
        BANKSEL    SSPSTAT     ; Bank 1
        BTFSS      SSPSTAT,S  ; "Start" Condition Occurred?
        GOTO       CHECK_STATUS3

        BCF        SSPSTAT,S  ; Reset The "Start" Status flag
        BANKSEL    PORTC       ; Bank 0
        BTFSS      FLAGS2,BYTE_FOR_XMT ; Data Transmit?
        GOTO       CHECK_FOR_RECEIVE

SEND_SLAVE_ADDRESS:
        MOVF       CALLED_SLAVE,W ; Do XMT, Send Slave
                                  ;  Addr
        MOVWF      SSPBUF
        BSF        FLAGS,START_OK ; "Start" is OK
RESET_RET:
        BCF        PIR1,SSPIF
        GOTO       CHECK_INT_FLAGS

CHECK_FOR_RECEIVE:
```

```
        BTFSS       FLAGS2,GET_A_BYTE ; Data Receive?
        GOTO        RESET_RET
        BANKSEL     SSPCON2    ; Bank 1
        BSF         SSPCON2,RCEN ; Enable Receive
        BANKSEL     PORTC        ; Bank 0
        GOTO        SEND_SLAVE_ADDR


;-------------------------------------------------------------------------------

CHECK_STATUS3:
        BANKSEL     SSPSTAT    ; Bank 1
        BTFSS       SSPSTAT,P  ; "Stop" Condition Occurred?
        GOTO        CHECK_STATUS4

DO_A_RESET:
        BCF         SSPSTAT,P  ; Reset "Stop" Condition Flag
        BANKSEL     PORTC        ; Bank 0
        BCF         SSPCON,WCOL ; Reset I2C Error Flags
        BCF         SSPCON,SSPOV
        CLRF        FLAGS
        BCF         PIR1,SSPIF
        GOTO        CHECK_INT_FLAGS


;-------------------------------------------------------------------------------

CHECK_STATUS4:
        BANKSEL     PORTC        ; Bank 0
        BTFSS       FLAGS2,BYTE_FOR_XMT
        GOTO        CHECK_STATUS5



        BANKSEL     SSPSTAT    ; Bank 1
        BTFSC       SSPSTAT,DA ; Data or Address RCVed?
        GOTO        DO_DATA_PROCESS1

        BTFSS       SSPSTAT,ACKSTAT ; Get ACKN?
        GOTO        NO_ACKN1
        BANKSEL     PORTC        ; Bank 0
        BSF         FLAGS,ADDR_ACK
        BCF         SSPCON,SSPOV  ; Reset Overflow
        MOVF        XMT_DATA,W        ; Get Data To Send
        MOVWF       SSPBUF              ; Send Data
        BCF         PIR1,SSPIF
        GOTO        CHECK_INT_FLAGS
```

```
NO_ACK1:
            BANKSEL     PORTC       ; Bank 0
            BSF         FLAGS2,ADDR_ERROR
            BANKSEL     SSPCON2     ; Bank 1
            BSF         SSPCON2,PEN     ; Send "Stop" Condition
            BANKSEL     PORTC       ; Bank 0
            BCF         PIR1,SSPIF
            GOTO        CHECK_INT_FLAGS

;------------------------------------------------------------------------

DO_DATA_PROCESS1:
            BTFSS       SSPSTAT,ACKSTAT ; Get ACKN?
            GOTO        NO_ACKN2

            BANKSEL     PORTC       ; Bank 0
            BSF         FLAGS2,XMT_DATA_OK
            BSF         FLAGS,DUMMY_WRITE
            BSF         SSPCON,SSPOV ; No ACKN Next Time
            MOVWF       SSPBUF      ; Dummy Write
            BCF         PIR1,SSPIF
            GOTO        CHECK_INT_FLAGS

NO_ACKN2:
            BANKSEL     PORTC       ; Bank 0
            BTFSS       FLAGS,DUMMY_WRITE
            GOTO        NO_DUMMY




DO_DUMMY_END:
            BCF         SSPCON,WCOL     ; Reset I2C Error Flags
            BCF         SSPCON,SSPOV
            CLRF        FLAGS
            BANKSEL     SSPCON2     ; Bank 1
            BSF         SSPCON2,PEN ; Send "Stop" Condition
            BANKSEL     PORTC       ; Bank 0
            BCF         PIR1,SSPIF
            GOTO        CHECK_INT_FLAGS

NO_DUMMY:
            BSF         FLAGS2,WRITE_ERROR
            GOTO        DO_DUMMY_END
```

```
;----------------------------------------------------------------------------

CHECK_STATUS5:
            BANKSEL    PORTC        ; Bank 0
            BTFSS      FLAGS2,GET_A_BYTE
            GOTO       DO_A_RESET

            BTFSC      FLAGS,MASTER_ACK
            GOTO       DO_DUMMY_END

            BANKSEL    SSPSTAT    ; Bank 1
            BTFSC      SSPSTAT,DA ; Data or Address?
            GOTO       DO_DATA_PROCESS2

            BTFSS      SSPCON2,ACKSTAT ; Get ACKN?
            GOTO       NO_ACKN1

            BANKSEL    PORTC        ; Bank 0
            MOVF       SSPBUF,W   ; Discard Address Received
            BSF        FLAGS,ADDR_ACK
            BCF        SSPCON,WCOL ; Reset I2C Error Flags
            BCF        SSPCON,SSPOV
            BANKSEL    SSPCON2    ; Bank 1
            BSF        SSPCON2,RCEN ; Enable RCV
            BANKSEL    PORTC        ; Bank 0
            BCF        PIR1,SSPIF
            GOTO       CHECK_INT_FLAGS

;----------------------------------------------------------------------------


DO_DATA_PROCESS2:
            BANKSEL    PORTC        ; Bank 0
            MOVF       SSPBUF,W   ; Get Data RCVed
            MOVWF      RCV_DATA   ; Store Data in User's Data Hold
            BSF        FLAGS2,RCV_OK
            BSF        MASTER_ACK
            BANKSEL    SSPCON2    ; Bank 1
            BSF        SSPCON2,ACKDT ; Send "No ACKN"
            BSF        SSPCON2,ACKEN
            BANKSEL    PORTC        ; Bank 0
            BCF        PIR1,SSPIF
            GOTO       CHECK_INT_FLAGS

;----------------------------------------------------------------------------
```

```
SET_START:
        BANKSEL    SSPCON2    ; Bank 1
        BSF        SSPCON2,SEN ; Set a "Start" Condition
        BANKSEL    PORTC       ; Bank 0
        RETURN
```
;---
```
        END
```

Chapter 10:   DSP Fundamentals

10.0   Chapter Summary

Section 10.2 looks at a simple low pass filter as a moving average. Section 10.3 looks at a similar high pass filter. Section 10.4 gives a short discussion of digital filters in the most general sense. Section 10.5 discusses aliasing and the Nyquist Sampling Theorem. Section 10.6 is a cookbook example of a practical low pass/high pass filter. Section 10.7 is a cookbook example of a practical band pass filter. Section 10.8 covers the concept of a median filter. Section 10.9 describes DTMF decoding using a series of standard band pass filters. Section 10.10 describes DTMF decoding by the deliberate use of aliasing. Section 10.11 discusses how DSP can be used for speech compression and sound effects.

10.1   Introduction

Digital Signal Processing (DSP) is usually thought of as a modern idea due to the availability of cheap computers, but its roots can be traced back to the 17th century. The main idea of DSP is to use equations to manipulate signals. That is, signals such as speech or communications waveforms, can be sampled at regular intervals of time, converted from voltage levels to numbers (ADC), and used as a sequence of these numbers, to feed a computer to run calculations on them with a given, user-defined equation. The resulting numbers are then converted back into voltage levels (DAC) to use as a new signal.

DSP techniques can be used to filter, detect, classify, encode, decode, and remove noise from signals. Why would anyone want to use DSP when other filters, detectors, and classifiers are already available? A DSP filter is an algorithm that runs on a cheap computer. It is programmable and it is flexible in that its program can be changed. For a given algorithm or program it is ultra-stable. The DSP filter does not change with temperature, component aging, component tolerances, and it is very resistant to errors in manufacturing. Part of its flexibility is in its adaptability. A much slower process can be run at the same time as the main filter program to measure the performance of the filter and then over time change the filter program to make it do a better job of filtering. Today when computers, ADCs, and DACs are very cheap, DSP is a *very* attractive idea!

10.2   An Example: A Low-Pass Filter

Let's see how a simple filter can be constructed with DSP techniques and show that it is possible to do filtering by doing calculations alone. Suppose that there is an ADC that takes 8-bit data sample-values at a regular rate of 20kHz and suppose that the signals we want to filter are sine waves.

Let the filter calculations be done this way: Let 16 ADC samples be stored in RAM in the order they were received. When a new sample comes in, it is also stored in RAM but the oldest of the 16 previous samples gets discarded. That is, at any time, there are only the 16 most recent samples, including the current sample stored in RAM. Let the "output" of the filter be the average of these 16 samples. That is, add up all of the samples together and divide the sum by 16 to get one filter "output" point or sample. This process is called a "moving-average".

Does this process do "low-pass" filtering? Let's see. Suppose that the input test signal is a sine wave at a frequency of 10 Hz. Since the sampling rate (or "the data rate") is 20 kHz, there are 2000 sample points in one cycle of the 10 Hz wave. Only 16 of these at any one time are used in the moving-average. Most of the 16 points have roughly the same values as each other. There are, at times, some differences between the points since the wave is changing over time, but at 10 Hz this change is very slow relative to the sampling rate. The filter output, or average, will be only slightly less than any one of the 16 sample values. That is, a "low" frequency will "pass" unchanged.

What happens if the input test signal is a sine wave with a frequency of 8500 Hz? Since the sampling rate is 20 kHz, there are at most only two successive samples that are "both positive" or "both negative" at any one time. If 16 successive samples were added together, most of them will cancel each other out. This sum will be made even smaller when it is divided by 16. So, for this "high" frequency, the filter output will be very small at any one time. A low-pass filter will attenuate high frequencies.

So, we have a low-pass filter just by doing the moving-average calculation!

10.3   An Example: A High-Pass Filter

This example is crude and more difficult to see, but with a relaxed view it can be seen by approximation.

Suppose that the moving-average is the same as in our high-pass filter except that, now instead of adding the 16 samples together they are alternately added to and subtracted from each other (and then the sum/difference result is divided by 16). That is, add the first, subtract the second, add the third, … and so on.

If our input test signal is the 10 Hz sine wave, most of these samples were nearly the same and consequently forming the new sum/difference will cancel these samples out. Then dividing the result by 16 will make the output even smaller. A "low" frequency is now attenuated!

The 8500 Hz signal input case is more difficult to see. Since this signal is mostly alternating in sign, or nearly so, the sum/difference process will now "subtract negatives" and "add positives" thereby reinforcing the "sum". Dividing the result by 16 will make it smaller, but the result, or output, will be much larger at any one time than for the "low" frequencies. This process is now a high-pass filter!

10.4   DSP Filters in General

DSP filters can also use past values of the outputs of the filter for an even greater filtering effect. In general, most DSP filters today use "weights" or "constant multipliers" that multiply each of the past inputs and each of the past outputs before forming the sum to be used as the filter output. Modern DSP filters are linear, recursive, difference equations. *Any* type of filter can be made by taking enough past inputs and past outputs and giving them each a proper multiplying weight. The weights modulate the filter's performance by making the response more uniform in the pass-band, giving a sharper cut-off of the undesired frequencies and overall, produce the desired filter shape.

10.5   Aliasing and the Nyquist Sampling Theorem

There is a limit to the highest frequency that can be represented in a signal when the sampling rate is set at some value. It turns out that for a given sampling rate, call it "SR", the largest frequency component of a signal to be sampled must not exceed the frequency "SR / 2". If that signal *does* have frequencies exceeding "SR / 2", those frequencies will be "reflected back" and will interfere with the valid frequencies that are less than "SR / 2". These "reflections back" are called the "alias" frequencies of a sampled signal. Unless these frequency components which are greater than "SR / 2" are removed by an analog filter prior to being sampled, they will be "aliased"; they will interfere with the frequency components which are less than "SR / 2", and there will be no way to remove them once they are sampled!

Why is this so? Why does "aliasing" occur at all? Let's go back to Chapter 7 to the part about the Direct Digital Synthesis method for generating sine waves. (Take some time to review that now.) It was said, at the time without proof, that the maximum frequency sine wave, which could be produced with DDS, was at half the sampling rate (data rate). Is this true? Assume that the system that was used in Chapter 7 is the same as the one we will use here. The sampling rate is 10 kHz and the accumulator and the table-increment are both 16-bit registers. Suppose we want to use this DDS system to produce a sine wave at 7500 Hz. The table-increment = 7500 / F-to-I = 49152 = 0xC000.

Let the accumulator start from zero and use the table-increment of 0xC000. This produces the following data:

Accumulator   Sine Wave Value
ACCUM = 0000   Zero
ACCUM = C000   - Max
ACCUM = 8000   Zero
ACCUM = 4000   + Max
ACCUM = 0000   Zero

The sampling rate is 10 kHz and this output has a period of four steps so our frequency is 2500 Hz!

On the other hand, suppose we have an ADC that samples at 10 kHz and we give it an input sine wave with a frequency of 7500 Hz. What does this look like?
See Figure 10-1. The resulting samples are the same as the DDS case: The samples are for a 2500 Hz sine wave!

Figure 10-1 Example of Aliasing When Sampling an Analog Signal
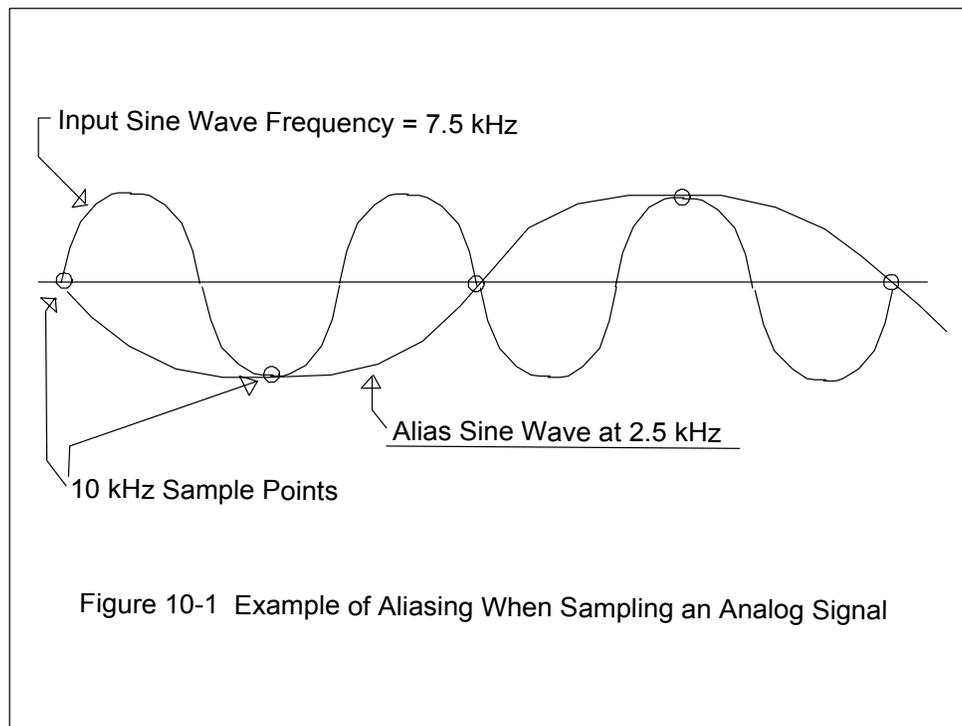
We can't tell the difference between a 2500 Hz sine wave and a 7500 Hz sine wave if the sampling rate is 10 kHz by looking at the samples they produce! We know

that the inputs are very different, but we can't tell the difference by looking at the samples alone. As far as we know there is no difference at all.

Aliasing is a fundamental fact of the nature of sampled-data systems. In most applications we wish to prevent aliasing from occurring. In *some* cases, however, we can use aliasing as a design technique. The decision to do so must be based on the size, cost and complexity of the system to be built. If aliasing is to be used in an answering machine, this is OK. If you are using DSP techniques to radar-map the surface of Venus in a Venus-orbiting satellite, *avoid aliasing at all costs!*

If the input signal waveform to a sampled-data system has a maximum frequency of "Fmax", the Nyquist Sampling Theorem says that the data can be exactly represented and recovered without aliasing if the sampling rate is at least twice the frequency of "Fmax". In practice, we use a little more than twice the frequency (say, 2.2 * Fmax), for insurance.

10.6   DSP Cookbook I --- A Simple LPF/HPF

A simple low-pass/high-pass filter can be formed with the equation:

Y(n) = X(n) – (B1) * Y(n-1).

Where Y(n) is the n-th output, Y(n-1) is the (n-1)-th output, X(n) is the n-th input, and B1 is a constant multiplying weight. If B1 is as:

B1 > 0,   Then the filter is a high-pass filter
B1 < 0,   Then the filter is a low-pass filter.

ABS(B1) < 1.0   is a MUST, or the filter will be unstable
                            (Its outputs will try to run to infinity).

In a modified frequency-domain, a "normalized" frequency called "A" is defined as:

A = (2 * Pi * f) / (Sampling Rate).

Since the frequency "f" ranges over the range 0 <= f < (SR / 2) the normalized frequency "A" will range over the range 0 <= A < Pi.

The filter "magnitude vs. normalized frequency" equation is:

Magnitude = 1 / sqrt(1 + (B1 ** 2) + (2 * B1 * cos(A))).

The maximum magnitude for this filter is Max Magnitude = 1 / (1 – B1).

The half-power (3 dB) frequency is given by:

$$\text{Cosine}(A) = 2 - ((1 - (B1 ** 2)) / (2 * B1)).$$

Note that the maximum magnitude and the filter gains get very large as "B1" is just less than one. Also, if the half-power (3 dB) frequency is to be "tight", "B1" must be slightly less than one, and therefore the gains will be large.

There are several problems that are encountered when we attempt to implement this filter on the PIC. Since "B1" is a constant, the multiply routine can be made as a "multiply-by-a-constant" using "RLF", "ADDWF", "SUBWF", or by using a look-up table (hashing). If the filter gains are too high, however, a single-byte multiply may not be sufficient. The inputs may have to be reduced (attenuated) *before* entering the filter to reduce the outputs that are made large by the large gains. This will make the small amplitudes of the signal even smaller. If the input must be a large amplitude and the gains must be large ("small bandwidth"), working with double-byte arithmetic is a MUST. Overall, the sampling rates and the PIC's oscillator frequency, "Fosc", must be used to judge how much time is allowed to do the filter calculations.

An example filter program is as follows: (Put the input into W when this is called and the output will also be in W)

```
FILTER:
            MOVWF       TEMP         ; Store input
            MOVF        OLD_OUTPUT,W  ; Get Y(n-1)
            CALL        MULTIPLY_B1
            ADDWF       TEMP,W       ; Add X(n), Get X(n) – B1 * Y(n-1)
            MOVWF       OLD_OUTPUT ; Set Y(n) as Next Y(n-1)
            RETURN                   ; W = Y(n) = Output
```

10.7   DSP Cookbook II --- A Simple BPF

A simple band-pass filter (BPF) can be produced by using the following equation:

$$Y(n) = X(n) - B1*Y(n-1) - B2*Y(n-2).$$

Where the Y()s are for the current, previous, and second-previous outputs, respectively. The X(n) is the current input and "B1" and "B2" are the constant multiplier weights. The weights "B1" and "B2" can be expressed in a different form as:

$$B1 = -2 * R * \text{Cosine}(G)$$
$$B2 = R ** 2.$$

Where "G" is the "Normalized Center Frequency Desired". The "R" value must be greater than zero and less than one. If "R" is greater than or equal to one, the filter will be unstable and the outputs will try to run to infinity.

The filter's "Magnitude vs. Normalized Frequency" equation is considerably more complicated than that of the LPF/HPF:

Magnitude = 1 / sqrt(C1 + C2 + C3)

C1 = 4 * B2 * (Cosine(A)) ** 2
C2 = 2 * B1 * (1 + B2) * Cosine(A)
C3 = B1 ** 2 + (1 – B2) ** 2

Where "A" is the normalized input frequency. The maximum gain at the center frequency is:

Max Magnitude = 1 / ((1 - R ** 2) * Sine(G)).

The bandwidth may be calculated from:

Cosine(A+) = Cosine(G) * ((1 + R ** 2)/(2 * R)) + Sine(G) * ((1 – R ** 2)/(2 * R))
Cosine(A-) = Cosine(G) * ((1 + R ** 2)/(2 * R)) – Sine(G) * ((1 – R ** 2)/(2 * R))

Bandwidth = ABS((A+) – (A-)) and A+- = Arccos(Cosine(A+-)).

The bandwidth is in the normalized frequency range.

Note that the gains and the maximum magnitude get very large as "R" is just less than one, but this is required for having narrow bandwidths. The exact same design and implementation issues are applicable here as they are in the LPF/HPF.

10.8   DSP Cookbook III --- A Median Filter

The median filter is a departure from the other filters in that there are no multiplying weights and no summation of values. The median filter is for removing impulse noise that may corrupt a signal. It removes noise spikes. It works by taking an odd number of samples, preserving their order in RAM, removing the oldest point to fill in the current point as before, but the set of samples is copied to a separate section of RAM and sorted into order from lowest to highest. Then the middle point (the median) is taken as the current output.

It is important to note that the median filter is nonlinear. Traditional frequency-domain analysis techniques will not work on it. The median filter looks in some ways like a low-pass filter but it preserves sharp transitions and edges like a high-pass filter. It

is also important to notice that the median filter *selects* but does not *calculate* in order to do its filtering action.

One application of the median filter that we will need is for cleaning-up the noise from the ADPCM expansion process (ADPCM was covered in Chapter 7). The ADPCM expansion process includes many overshoots and undershoots in the signal as it tries to approximate the original signal. These come out as noise spikes and are present in enough volume to make listening to the output difficult. The median filter removes these noise spikes and there is no noticeable noise at the median filter's output.

One implementation problem with the median filter is running the filter at high speeds. If the median filter is coded with a brute-force sorting-algorithm, it will take 113 instruction cycles to run. This is too much time and the median filter could not be used in this form without moving the PIC to a higher oscillator frequency. But, an improved algorithm will take only 57 instruction cycles, and this will make the median filter practical in this application. The details of this improved algorithm are outlined in Chapter 6 under the example of ROM States.

10.9   DSP Example I --- Standard DTMF Decoding

The "standard" way to decode DTMF signals is to use a bank of seven (7) band-pass filters to decode each single frequency. These frequencies are:

697 Hz, 770 Hz, 852 Hz, 941 Hz, 1209 Hz, 1336 Hz, and 1477 Hz.

More complete details of the DTMF signaling process can be found in Chapter 7. Each BPF will have two multiplier weights giving a total of 14 weights. Since the highest frequency to measure is 1477 Hz, the sampling rate can be set at 3 kHz to satisfy the Nyquist Sampling Theorem. This will give us 333 microseconds to do a run of all of the filters. Another requirement is to measure the outputs of each of the filters to see if they are showing high volume outputs to show if that particular frequency is present. This means that each output of each filter must be converted to an absolute-value and accumulated or "integrated" to judge if that frequency is present. This must use a double-byte buffer for each filter to make this accumulation. After about 50 milliseconds, we can stop the process and look for peaks in the accumulated filter outputs and decode them into symbols (if there were any at all – there might not be any).

If the PIC's clock oscillator is running at 4 MHz, each instruction will take one microsecond to run. This means that the complete process of all of the filters and their accumulations must run with only 333 instruction cycles. Some minor amount of overhead is also needed.

Assume that all of the weight multiplications use look-up tables with 256 entries each. This requires a total memory space of 3.5 kilo-words in ROM.

Let's look at one BPF and one double-byte add and estimate the time it takes to run and then multiply that by seven (7) to see if the process can be done in less than 333 microseconds. Let the following routine be called with the new input-data in a RAM location called, "INPUT_N":

```
BPF_FILTER:
        MOVF        OLD_OUT1,W      ; Get Y(n-1)
        CALL        WEIGHT_TABLE1   ; Get  Product B1 * Y(n-1) in W
        MOVWF       TEMP
        MOVF        OLD_OUT2,W      ; Get Y(n-2)
        CALL        WEIGHT_TABLE2   ; Get  Product B2 * Y(n-2) in W
        ADDWF       TEMP,W          ; Sum The Two Products in W
        ADDWF       INPUT_N,W  ; Add the Input to the two products
        MOVWF       RESULT          ; Store The Y(n) Temporarily

        MOVF        OLD_OUT1,W      ; Update Y(n-2) for Next Time
        MOVWF       OLD_OUT2
        MOVF        RESULT,W        ; Update Y(n-1) for Next Time
        MOVWF       OLD_OUT1

        MOVF        RESULT,W        ; Convert to ABS Value
        BTFSC       RESULT,7
        SUBLW       0

        ADDWF       ACCUM_LOW,F     ; Add to Accumulator-Low
        BTFSC       STATUS,C        ; Add Carry If Any
        INCF        ACCUM_HIGH,F
        RETURN
```

The above BPF and accumulate code takes about 40 microseconds to run. Seven of these gives 280 microseconds. If we can do all of the overhead in 53 microseconds, this solution will work. If not, we may need a faster clock speed.

10.10   DSP Example II --- Alternative DTMF Decoding

It may be easier to do the DTMF decoding without band-pass filters and without a lot of memory by using aliasing as a design technique. Each of the DTMF frequencies are sinusoidal with very tight tolerances on each frequency (2%). At any time there are *only* two frequencies present. All of the frequency signals have the same amplitude as the others. The whole waveform is just the sum of these two sine waves.

Suppose that we set the sampling rate to be exactly one of the DTMF frequencies, one at a time, and over a window of a few milliseconds. On the "exact frequency" we would sample the same corresponding part of the wave at any time – there would be no

change in the voltage of that part of the sine wave. We would alias it to a DC value! The other sine wave would also be aliased to some other sine wave frequency. If we accumulated each sample, the DC parts of one frequency alias would add together, but the other alias frequency would alternate in sign, and its samples would add to zero! If neither frequency were present, they would both be aliased to other sine waves and they would both add to zero.

Two problems may exist in this scheme. One is that the DC valued alias that we measure may be near zero and add only a small amount of value each time it gets added to the sum. The other problem is that the "other" alias sine wave may be aliased to a very low frequency and when adding its samples it may "look like" DC and thereby confuse the whole process.

The first problem can be solved by having a second sampling of data within one period with a shift of 90 degrees and run a second accumulator for it. If the first sum gives us zero, the second sum would give us the peak sine values! Between the two sums we can measure some substantial amount of DC if that frequency is present.

To solve the second problem we need to know how to predict where the second frequency would be aliased. That is, go through all of the possible combinations of aliases and see if there are any frequencies that are too low. Let's develop a rule for finding the alias frequency.

The aliasing formulas look like:

0	$\leq f \leq$	(SR / 2)	No Aliasing
(SR / 2)	$\leq f \leq$	(SR)	Alias = SR – f
(SR)	$\leq f \leq$	(3*SR/2)	Alias = f – SR
(3*SR/2)	$\leq f \leq$	(2*SR)	Alias = (2*SR) – f
(2*SR)	$\leq f \leq$	(5*SR/2)	Alias = f – (2*SR)
(5*SR/2)	$\leq f \leq$	(3*SR)	Alias = (3*SR) – f

Where "f" is the input frequency of a single sine wave and "SR" is the sampling rate.

A summary of the aliasing analysis of the DTMF tones will be given here. First, an example of one of the aliasing table results will be given as follows:

If SR = 770 Hz, Then:
        F = 697 Hz   Alias = 73 Hz
        F = 852 Hz   Alias = 82 Hz
        F = 941 Hz   Alias = 171 Hz
        F = 1209 Hz   Alias = 439 Hz
        F = 1336 Hz   Alias = 566 Hz
        F = 1477 Hz   Alias = 707 Hz

This table is the worst case of all the possible aliasing tables. Therefore, the lowest aliasing frequency of everything is 73 Hz. To get cancellation of this wave, we would need at least one cycle at a period of 13.7 milliseconds. For seven frequencies to measure we need seven periods of 13.7 milliseconds each (roughly) to give a total of 96 milliseconds to sample the frequencies. Therefore, we can judge if the DTMF signals are present and decode them in about one hundred milliseconds. There is no "second" problem.

The software to measure the DTMF frequencies is much shorter than the band-pass filter DTMF decoding method. The main loop must measure two samples per wave at a 90-degree difference. The delays that make up the period of the sampling rates can be adjusted to fit each frequency, but the delay routine can be made in a general way. This software is as follows:

```
DTMF_ALIAS:                          ; Call with Code in W
            CALL        SET_DELAYS ; Set up delay/period & clear
                                    ;   accum
DTMF_LOOP:
            CLRW                    ; Signal: Add Zero to Accum
            CALL        GET_SAMPLE1 ; Add ADC to Accum, Start
                                    ; ADC
            CALL        DELAY       ; Delay for Period / 4
            NOP
            NOP
            NOP
            MOVLW       1           ; Signal: Add ADC to Accum
            CALL        GET_SAMPLE2 ; Do 16-Bit Add to Accum
            CALL        DELAY
            NOP
            NOP
            NOP
            MOVLW       1           ; Signal: Add ADC to Accum
            CALL        GET_SAMPLE1 ; ADC is 8-Bit Data, Use Sign
                                    ;  Ext
            CALL        DELAY
            NOP
            NOP
            NOP
            CLRW                    ; Signal Add Zero to Accum
            CALL        GET_SAMPLE2
            CALL        DELAY
            BTFSS       FLAGS,DO_RETURN ; This Flag Set in Delay
            GOTO        DTMF_LOOP
            RETURN
```

The "GET_SAMPLE" routines are identical except that they work with different accumulators. The "DELAY" routine not only delays to produce sampling at one-fourth of the period in the DTMF_LOOP, but it also counts the number of these delays and sets a flag to signal the DTMF_LOOP to return to the calling routine.

```
GET_SAMPLE1:
        MOVWF       ENABLE              ; Store the Code to "Add or Zero"
        MOVF        ADRESH,W
        SUBLW       D'127'              ; Convert to Two's Complement
        BTFSS       ENABLE,0
        CLRW                            ; Code = 0, "Add Zero to Accum"
        MOVWF       ADC_VALUE
        CLRF        EXTEND              ; Clear Sign-Extend Byte
        BTFSC       ADC_VALUE,7         ; Test Sign-Bit
        COMF        EXTEND              ; Sign-Extend as "Negative"
        CLRF        CARRY
        MOVF        ADC_VALUE,W
        ADDWF       ACCUM_LOW1,F        ; Add ADC to Accum1 (Low)
        BTFSC       STATUS,C            ; Capture the Carry-State
        INCF        CARRY,F
        MOVF        CARRY,W             ; Add Carry to Accum1 (High)
        ADDWF       ACCUM_HIGH1,F
        MOVF        EXTEND,W            ; Add Sign-Extend to Same
        ADDWF       ACCUM_HIGH1,F
        BCF         PIR1,ADIF           ; Prepare & Start ADC
        BSF         ADCON0,GO
        RETURN

DELAY:
        MOVF        MASTER_WHOLE,W  ; Get "Reference Copy" of
        MOVWF       WHOLE               ; "Whole-Loop" Delay
        MOVF        FRACTION,W
        ADDWF       PCL,F
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
DELAY_LOOP:
        NOP
        NOP
        NOP
        NOP
```

```
        NOP
        DECFSZ      WHOLE,F
        GOTO        DELAY_LOOP

        DECFSZ      COUNT,F
        RETURN
        BSF         FLAGS,DO_RETURN
        RETURN
```

10.11   DSP Application: Speech Compression

In the early 1970's a company called Compressed Time, Inc. (CTI) came out with a novel machine for manipulating the play-back speeds of tape-recorded speech. The idea was to take the speech of a slow-speaking person on a tape-recording and speed it up but compensate for the shift in the speech frequencies. That is, the rate of delivery of the speech would be faster but the person would still appear to speak in a "normal" tone of voice! The opposite could also be done. That is, slow-down a fast-speaking person and have him/her appear to speak in a "normal" voice. CTI called this process, "Speech Compression".

The equipment that CTI developed was expensive and it contained a tape-recorder with several play-back heads mounted on a rotating drum. The tape would be fed at the desired speed while the drum would rotate at a speed that would sample the speech on the tape and produce a play-back speed difference so that the relative play-back speed was "normal".

Doing the same process today is made cheap and easy by using DSP techniques. Speech compression can be done very easily with the PIC although more analog filtering would be required.

Let's consider how we can do speech compression with DSP techniques.

Telephone-quality speech has a maximum frequency of about 4 kHz. To sample and recover this speech would require a sampling-rate of 8 kHz to avoid aliasing. If the speech were on a tape-recording and the play-back speed were doubled, the maximum speech frequency would also double and the new sampling-rate would be doubled to 16 kHz.

There is another kind of sampling that must be considered, however. When we speak, we hold each consonant and each vowel sound for a minimum of about 20 milliseconds. This figure is only approximate and it varies considerably from speaker to speaker. If each consonant and vowel sound were considered to be a "bit" of "information", the typical "information-rate" is about 50 Hz at most! When the tape-

recording play-back speed is doubled this information-rate is also doubled to 100 Hz at most.

Both of these processes need to be sampled in order to do speech compression. Sampling each waveform point at 16 kHz is easy but how do we sample the "information"? The answer is to fill an array of 16 kHz samples over a time-window of about 10 milliseconds to get a total of 160 sample-elements. Ten milliseconds corresponds to an "information-sampling-rate" of 100 Hz, but this is not the proper Nyquist sampling rate for 100 Hz data. In actual practice, a sampling-rate for the time-window samples may be substantially different from the Nyquist rate. The danger is in loosing information and the actual number of samples in the array may need to be larger to faithfully reproduce the speech sounds. This *will* require some experimentation to get the speech to sound the right way and with no distortion.

To play this speech back at a "normal" tone of voice, the 160 array points are sent out at an 8 kHz rate over a 20-millisecond time interval. This process is shown in Figure 10-2. The speech is recorded at 16 kHz and the 160 samples are spread-out by sending them out at 8 kHz. In this time interval an equivalent of 160 samples are *skipped* in order to send out the first 160 points. The "catch" in doing speech compression in the "Fast-to-Slow" mode is that some information is lost and cannot be recovered. But if there is a right mix of sample points and array-sample-windows, the human ear will not detect the lost information. In addition to a low-pass filter to smooth the output, a high-pass filter is needed to get rid of the 50 Hz noise that comes in when the array-sample-windows are "chopped" or "broken" on the play-back process. The human ear will hear the speaker in a normal tone of voice but at double the speed.

Let's now consider the "opposite" of the "Fast-to-Slow" mode process. That is, let's do the "Slow-to-Fast" mode process.

If the play-back speed were cut in half, the maximum speech frequency would be 2 kHz and we would need a sampling-rate of only 4 kHz. If a 20-millisecond time-window were used, this would fill an array of 80 sample points.

The "Slow-to-Fast" mode process is shown in Figure 10-3. The trick is to completely fill the 80-element array and then send it out, *twice*, each time at a rate of 8 kHz. This time there is no lost information. The human ear will hear the speaker in a normal tone of voice but at half the speed.

Another application of these techniques is to alter the tone of voice of a live, normal speaker. This gives some very strange-sounding effects. Several radio and television science fiction shows have used these techniques to radically change the speech of their actor's voices.
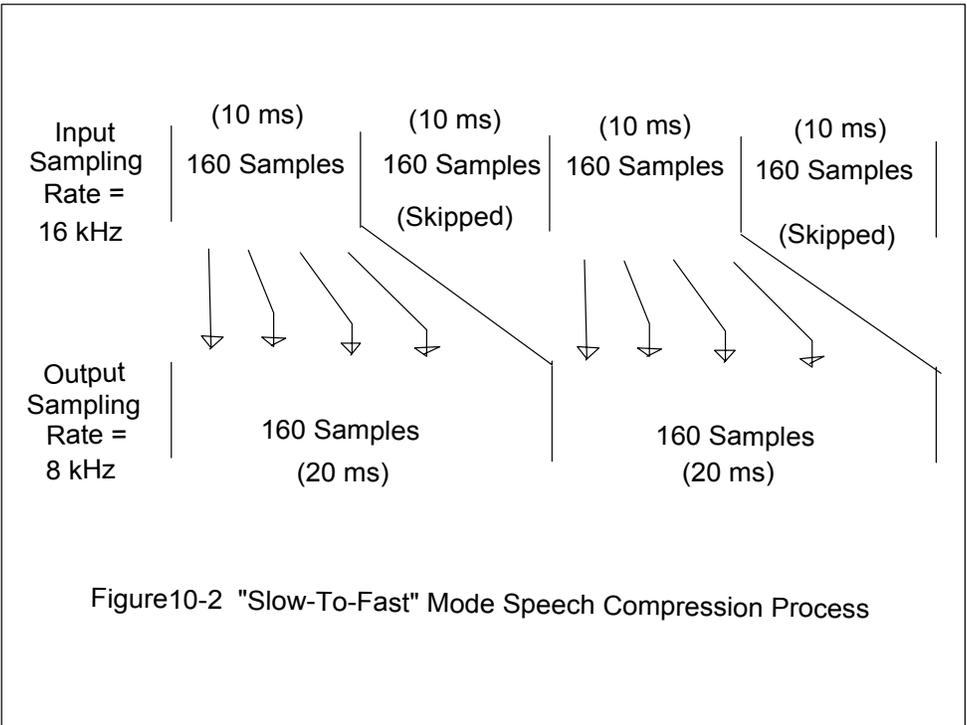
Figure10-2 "Slow-To-Fast" Mode Speech Compression Process
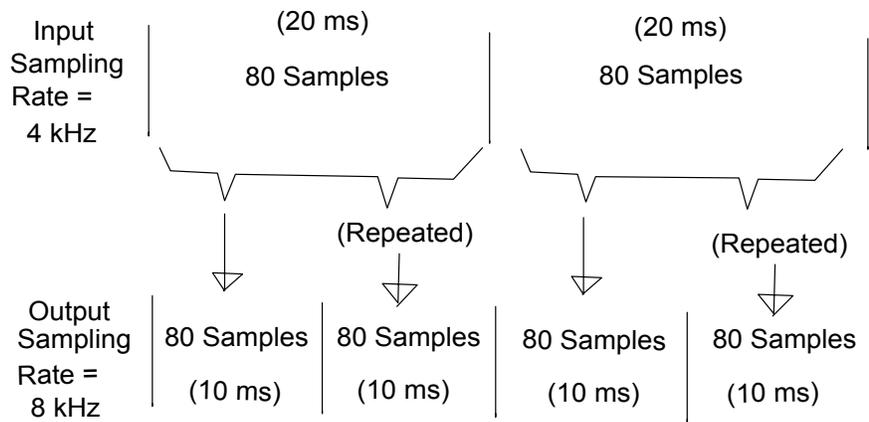
Figure 10-3 "Fast-To-Slow" Mode Speech Compression Process

Appendix A --- The PIC16F877 Instruction Set

The PIC16F877 has a total of 35 instructions, all of which are as a single word.

ADDLW   ("Add Literal to W")
Syntax: ADDLW   k
Operand: k, where 0 <= k <= 255
Operation: (W) = (W) + k
Status: C, DC, Z
Cycles: 1 Instruction Cycle
Binary Code: 11 111x kkkk kkkk
Code Example: "ADDLW   7"
Operation Example: If (W) = 3 and "ADDLW   7" is executed, (W) = 10.
Description:
        The contents of the W register are added to the constant, k, and the
        Results are placed in the W register.

ADDWF   ("Add W and file")
Syntax: ADDWF   f,d
Operands: f, where 0 <= f <= 127
                 And d, where d = {0,1}
Operation: (W) = (W) + (file)   if d = 0
                 (file) = (W) + (file)   if d = 1
Status: C, DC, Z
Cycles: 1 Instruction Cycle
Binary Code: 00 0111 dfff ffff
Code Examples: "ADDWF   SUM,W"   for d = 0
                 "ADDWF   SUM,F"   for d = 1
Operation Examples:
                 If (W) = 10 and (SUM) = 20,
                         "ADDWF   SUM,W" gives (W) = 30 and (SUM) = 20
                         "ADDWF   SUM,F"   gives (W) = 10 and (SUM) = 30
Description:
        The contents of the W register are added to the contents of the
        Register-file and the results are placed in:
                 (W) if d = 0
                 (file) if d = 1

ANDLW   ("And Literal and W")
Syntax: ANDLW   k
Operand: k, where 0 <= k <= 255
Operation: (W) = (W) & k
Status: Z
Cycles: 1 Instruction Cycle
Binary Code: 11 1001 kkkk kkkk
Code Example: "ANDLW   0x07"
Operation Example: If (W) = 0xE6 and "ANDLW   0x07" is executed,
                           (W) = 0x06.
Description:
        The contents of the W register are ANDed with the constant, k, and the
        Results are placed in the W register.

ANDWF   ("And W and file")
Syntax: ANDWF   f,d
Operands: f, where 0 <= f <= 127
                   And d, where d = {0,1}
Operation: (W) = (W) & (file)   if d = 0
                   (file) = (W) & (file) if d = 1
Status: Z
Cycles: 1 Instruction Cycle
Binary Code: 00   0101 dfff ffff
Code Examples: "ANDWF   SUM,W" for d = 0
                   "ANDWF   SUM,F"   for   d = 1
Operation Examples:
                   If (W) = 0xC7 and (SUM) = 0xE5,
                   "ANDWF   SUM,W" gives (W) = 0xC5 and (SUM) = 0xE5
                   "ANDWF   SUM,F"   gives (W) = 0xC7 and (SUM) = 0xC5
Description:
        The contents of the W register are ANDed to the contents of the
        Register-file and the results are placed in:
                   (W) if d = 0
                   (file) if d = 1

BCF   ("Bit Clear file")
Syntax: BCF   f,b
Operands: f, where 0 <= f <= 127
              And b, where 0 <= b <= 7
Operation: file<b> = 0
Status: None
Cycles: 1 Instruction Cycle
Binary Code: 01 00bb bfff ffff
Code Example: "BCF   FLAGS,3"
Operation Example: If FLAGS<3> = 1, doing "BCF FLAGS,3" will make
                    FLAGS<3> = 0.
Description:
        Bit number, b, is cleared (=0) in the register-file.

BSF   ("Bit Set file")
Syntax: BSF   f,b
Operands: f, where 0 <= f <= 127
              And b, where 0 <= b <= 7
Operation: file<b> = 1
Status: None
Cycles: 1 Instruction Cycle
Binary Code: 01 01bb bfff ffff
Code Example: "BSF   FLAGS,3"
Operation Example: If FLAGS<3> = 0, doing "BSF FLAGS,3" will make
                    FLAGS<3> = 1.
Description:
        Bit number, b, is set (=1) in the register-file.

BTFSC   ("Bit Test, Skip If Clear")
Syntax: BTFSC   f,b
Operands: f, where 0 <= f <= 127
              And b, where 0 <= b <= 7
Operation: Skip Next Instruction if file<b> = 0.
Status: None
Cycles: 1 Cycle If "NO" Skip
              2 Cycles If SKIP
Binary Code: 01 10bb bfff ffff
Code Example: "BTFSC   FLAGS,3"
Operation Example: If FLAGS<3> = 0, "BTFSC FLAGS,3" will Skip.
                    If FLAGS<3> = 1, "BTFSC FLAGS,3" will Not Skip.

Description:

    If the register-file bit, b, is zero, the next instruction will be skipped.

    If the register-file bit, b, is one, the next instruction will execute.

BTFSS   ("Bit Test, Skip If Set")

Syntax: BTFSS   f,b

Operands: f, where 0 <= f <= 127

                And b, where 0 <= b <= 7

Operation: Skip Next Instruction if file<b> = 1.

Status: None

Cycles: 1 Cycle If "NO" Skip

                2 Cycles If SKIP

Binary Code: 01 11bb bfff ffff

Code Example: "BTFSS   FLAGS,3"

Operation Example: If FLAGS<3> = 1, "BTFSS FLAGS,3" will Skip.

                  If FLAGS<3> = 0, "BTFSS FLAGS,3" will Not Skip.

Description:

    If the register-file bit, b, is one, the next instruction will be skipped.

    If the register-file bit, b, is zero, the next instruction will execute.

CALL   ("Call Subroutine")

Syntax: CALL   k

Operand: k, where 0 <= k <= 2047

Operation: (Program Counter) + 1 is moved to the Top-of-Stack

                Address = k is transferred to PC<10:0>

                (PCLATH<4,3>) is transferred to PC<12,11>

Status: None.

Cycles: 2 Instruction Cycles

Binary Code: 10 0kkk kkkk kkkk

Code Example: "CALL   START"

Operation Example: If START = 0x00C7 and "CALL START" is done,

                (PC) = 0x00C7.

Description:

    Call a subroutine. The incremented program counter (PC) is placed

    On the stack, the program counter, bits zero through 10, are loaded

    With the target address, k, and the PCLATH bits <4,3> are placed into

    The program counter bits <12,11>.

Notes:

    Doing a CALL within a 2 K block does not require any manipulation

    Of the PCLATH register. If the CALL is beyond the current 2 K block,

    The PCLATH register bits <4,3> must be set-up *before* doing the CALL.

CLRF   ("Clear file")
Syntax: CLRF   f
Operand: f, where 0 <= f <= 127
Operation: (f) = 0.
Status: Z
Cycles: 1 Instruction Cycle
Binary Code: 00 0001 1fff   ffff
Code Example: "CLRF   FLAGS"
Operation Example: If FLAGS = 0x20, doing "CLRF FLAGS" will make
                        (0x20) = 0x00.
Description:
          The register-file is reset. Zeros are written to this RAM byte.

CLRW   ("Clear W")
Syntax: CLRW
Operand: None.
Operation: (W) = 0.
Status: Z
Cycles: 1 Instruction Cycle
Binary Code: 00 0001 0xxx xxxx
Code Example: "CLRW"
Operation Example: Doing "CLRW" makes (W) = 0x00.
Description:
          Zeros are written to the W register.

CLRWDT   ("Clear Watch-Dog Timer")
Syntax: CLRWDT
Operand: None
Operation: (WDT) = 0x00. (WDT Prescaler) = 0x00.
Status: STATUS Flags: /TO = 1 and /PD = 1.
Cycles: 1 Instruction Cycle
Binary Code: 00 0000 0110 0100
Code Example: "CLRWDT"
Operation Example: Doing "CLRWDT" resets the Watch-Dog Timer.
Description:
          CLRWDT resets the Watch-Dog Timer and its prescaler.

COMF   ("Complement   file")
Syntax: COMF   f,d
Operands: f, where 0 <= f <= 127
                And d, where d = {0,1}
Operation: (W) = /(f) if d = 0
                (f) = /(f) if d = 1
Status: Z
Cycles: 1 Instruction Cycle
Binary Code: 00   1001   dfff   ffff
Code Examples: "COMF   FLAGS,W"   for d = 0.
                    "COMF   FLAGS,F"   for d = 1
Operation Examples:
        If (W) = 0xE3 and (FLAGS) = 0xF1,
          Doing "COMF FLAGS,W" gives (W) = 0x0E and (FLAGS) = 0xF1.
          Doing "COMF FLAGS,F"   gives (W) = 0xE3 and (FLAGS) = 0x0E.
Description:
        Take the one's complement of the register-file and place the results:
                In (W) if d = 0
                In (file) if d = 1.

DECF   ("Decrement file")
Syntax: DECF   f,d
Operands: f, where 0 <= f <= 127
                And d, where d = {0,1}
Operation: (W) = (f) – 1   if d = 0
                (f) = (f) – 1   if   d = 1
Status: Z
Cycles: 1 Instruction Cycle
Binary Code: 00 0011 dfff   ffff
Code Examples: "DECF   COUNT,W"   for   d = 0
                    "DECF   COUNT,F"   for   d = 1
Operation Examples:
        If (W) = 0x3A and (COUNT) = 0x12,
          Doing "DECF   COUNT,W" gives (W) = 0x11 and (COUNT) = 0x3A.
          Doing "DECF COUNT,F" gives (W) = 0x3A and (COUNT) = 0x11.
Description:
        The contents of the register-file are decremented by one and the results
        Are placed as:
                In (W) if d = 0
                In (file) if d = 1.

DECFSZ   ("Decrement file, Skip If Zero")
Syntax: DECFSZ   f,d
Operands: f, where $0 <= f <= 127$
                And d, where d = {0,1}
Operations: (W) = (f) – 1   if d = 0
                (f) = (f) – 1   if d = 1
                and Skip the next instruction if the result was zero.
Status: None.
Cycles: 1 Instruction Cycle if non-zero result (No Skip)
                2 Instruction Cycles if zero result (Skip)
Binary Code: 00 1011 dfff ffff
Code Example: "DECFSZ   COUNT,W" if d = 0
                  "DECFSZ   COUNT,F"   if d = 1
Operation Example:
        If (W) = 0x10 and (COUNT) = 0x01,
          "DECFSZ   COUNT,W" gives (W) = 0x00 and (COUNT) = 0x01
          "DECFSZ   COUNT,F" gives (W) = 0x10 and (COUNT) =0x00
                In Both Cases: The Next Instruction Will Be Skipped.
        If (W) = 0x10 and (COUNT) = 0x23,
          "DECFSZ   COUNT,W" gives (W) = 0x22 and (COUNT) = 0x23
          "DECFSZ   COUNT,F"   gives (W) = 0x10 and (COUNT) = 0x22
                In Both Case: The Next Instruction Will Be Executed.
Description:
        The contents of the register-file is decremented and the results are
        Placed in:
                        (W) if d = 0
                        (file) if d = 1
        And the next instruction will be skipped if the result was zero.
        Otherwise, the next instruction will be executed.

GOTO   ("Unconditional Branch")
Syntax: GOTO   k
Operand: k, where   0 <= k <=   2047
Operation: Address = k is transferred to PC<10:0>
                (PCLATH<4,3>) is transferred to PC<12,11>
Status: None.
Cycles: 2 Instruction Cycles
Binary Code: 10 1kkk   kkkk   kkkk
Code Example: "GOTO   START"
Operation Example: If START = 0x00C7 and "GOTO   START" is done,
                    (PC) = 0x00C7.
Description:
        Jump to a new address.
        The program counter, bits zero through 10, are loaded
        With the target address, k, and the PCLATH bits <4,3> are placed into
        The program counter bits <12,11>.
Notes:
        Doing a GOTO within a 2 K block does not require any manipulation
        Of the PCLATH register. If the GOTO is beyond the current 2 K block,
        The PCLATH register bits <4,3> must be set-up *before* doing the GOTO.

INCF   ("Increment file")
Syntax: INCF   f,d
Operands: f, where 0 <= f <= 127
                And d, where d = {0,1}
Operation: (W) = (f) + 1   if d = 0
                (f) = (f) + 1   if d = 1
Status: Z
Cycles: 1 Instruction Cycle
Binary Code: 00 1010 dfff   ffff
Code Examples: "INCF   COUNT,W"   for d = 0
                 "INCF   COUNT,F"   for d = 1
Operation Examples:
        If (W) = 0x3A and (COUNT) = 0x12,
          Doing "INCF   COUNT,W" gives (W) = 0x13 and (COUNT) = 0x3A.
          Doing "INCF COUNT,F" gives (W) = 0x3A and (COUNT) = 0x13.
Description:
        The contents of the register-file are incremented by one and the results
        Are placed as:
                In (W) if d = 0
                In (file) if d = 1.

INCFSZ   ("Increment file, Skip If Zero")
Syntax: INCFSZ   f,d
Operands: f, where 0 <= f <= 127
                 And d, where d = {0,1}
Operations: (W) = (f) + 1   if d = 0
                 (f) = (f) + 1   if d = 1
                 and Skip the next instruction if the result was zero.
Status: None.
Cycles: 1 Instruction Cycle if non-zero result (No Skip)
                 2 Instruction Cycles if zero result (Skip)
Binary Code: 00 1111 dfff ffff
Code Example: "INCFSZ   COUNT,W" if d = 0
                   "INCFSZ   COUNT,F"   if d = 1
Operation Example:
        If (W) = 0x10 and (COUNT) = 0xFF,
          "INCFSZ   COUNT,W" gives (W) = 0x00 and (COUNT) = 0xFF
          "INCFSZ   COUNT,F" gives (W) = 0x10 and (COUNT) =0x00
                In Both Cases: The Next Instruction Will Be Skipped.
        If (W) = 0x10 and (COUNT) = 0x23,
          "INCFSZ   COUNT,W" gives (W) = 0x24 and (COUNT) = 0x23
          "INCFSZ   COUNT,F"   gives (W) = 0x10 and (COUNT) = 0x24
                In Both Case: The Next Instruction Will Be Executed.
Description:
        The contents of the register-file is incremented and the results are
        Placed in:
                        (W) if d = 0
                        (file) if d = 1
        And the next instruction will be skipped if the result was zero.
        Otherwise, the next instruction will be executed.

IORLW   ("Inclusive Or Literal and W")
Syntax: IORLW   k
Operand: k, where 0 <= k <= 255
Operation: (W) = (W) | k
Status: Z
Cycles: 1 Instruction Cycle
Binary Code: 11 1111 kkkk kkkk
Code Example: "IORLW   0x07"
Operation Example: If (W) = 0xE6 and "IORLW   0x07" is executed,
                      (W) = 0xE7.

Description:

The contents of the W register are ORed with the constant, k, and the
Results are placed in the W register.

IORWF   ("Inclusive Or W and file")
Syntax: IORWF   f,d
Operands: f, where 0 <= f <= 127
                  And d, where d = {0,1}
Operation: (W) = (W) | (file)   if d = 0
                  (file) = (W) | (file)   if d = 1
Status: Z
Cycles: 1 Instruction Cycle
Binary Code: 00   0100 dfff   ffff
Code Examples: "IORWF   SUM,W" for d = 0
                     "IORWF   SUM,F"   for d = 1
Operation Examples:
                  If (W) = 0xC7 and (SUM) = 0xE5,
                  "IORWF   SUM,W" gives (W) = 0xE7 and (SUM) = 0xE5
                  "IORWF   SUM,F"   gives (W) = 0xC7 and (SUM) = 0xE7
Description:

The contents of the W register are ORed to the contents of the
Register-file and the results are placed in:
                  (W) if d = 0
                  (file) if d = 1

MOVF   ("Move file")
Syntax: MOVF   f,d
Operands: f, where 0 <= f <= 127
                  And d, where d = {0,1}
Operations: (W) = (f) if d = 0
                  (f) = (f) if d = 1
Status: Z
Cycles: 1 Instruction Cycle
Binary Code: 00 1000 dfff   ffff
Code Examples: "MOVF   TEMP,W"   for d = 0
                     "MOVF   TEMP,F"   for d = 1
Operation Examples:
        If (W) = 0x25 and (TEMP) = 0xC7,
            "MOVF   TEMP,W" gives (W) = 0xC7 and (TEMP) = 0xC7
            "MOVF   TEMP,F"   gives (W) = 0x25 and (TEMP) = 0xC7

Description:

        If d = 0, the register-file contents are placed in W, with the register
        File unchanged. If d = 1, there is no change in the contents of either
        W or the register-file. In both cases the Z flag is set if the data is zero
        And it is reset otherwise.

MOVLW   ("Move Literal to W")

Syntax: MOVLW   k
Operand: k, where 0 <= k <= 255
Operation: (W) = k
Status: None
Cycles: 1 Instruction Cycle
Binary Code: 11 00xx kkkk kkkk
Code Example: "MOVLW   0xB5"
Operation Example:
        If (W) = 0x03, Doing "MOVLW 0xB5" gives (W) = 0xB5.
Description:
        The literal value, k, is moved into the W register.

MOVWF   ("Move W to file")

Syntax: MOVWF   f
Operand: f, where 0 <= f <= 127
Operation: (f) = (W)
Status: None
Cycles: 1 Instruction Cycle
Binary Code: 00 0000 1fff ffff
Code Example: "MOVWF   TEMP"
Operation Example:
        If (W) = 0xA4 and (TEMP) = 0x3F,
         "MOVWF   TEMP" gives (W) = 0xA4 and (TEMP) = 0xA4.
Description:
        The contents of W are moved to the register-file.

NOP   ("No Operation")

Syntax: NOP
Operand: None
Operation: None
Status: None
Cycles: 1 Instruction Cycle
Binary Code: 00 0000 0000 0000
Code Example: "NOP"

Operation Example: Doing "NOP" does nothing.
Description:
      This instruction does nothing but does take one instruction cycle
      To execute.

RETFIE   ("Return from Interrupt")
Syntax: RETFIE
Operand: None
Operation: (PC) = Top-of-Stack and INTCON\<GIE\> = 1
Status: None
Cycles: 2 Instruction Cycles
Binary Code: 00 0000 0000 1001
Code Example: "RETFIE"
Operation Example:
      If (TOS) = 0x0017 and (PC) = 0x108F,
          "RETFIE" gives (PC) = 0x0017.
Description:
      The program counter is filled with the address at the top of the stack.
      Also, the INTCON bit "GIE" is set (=1).

RETLW   ("Return from Subroutine with Literal in W")
Syntax: RETLW   k
Operand: k, where 0 <= k <= 255
Operation: (PC) = Top-of-Stack   and (W) = k
Status: None
Cycles: 2 Instruction Cycles
Binary Code: 11 01xx kkkk kkkk
Code Example: "RETLW   0x34"
Operation Example:
      If (TOS) = 0x0017 and (PC) = 0x108F and (W) = 0xFC,
        "RETLW   0x34" gives (PC) = 0x0017 and (W) = 0x34.
Description:
      The program counter is filled with the address at the top of the stack.
      Also, the W register is filled with the literal value, k.

RETURN   ("Return from Subroutine")
Syntax: RETURN
Operand: None
Operation: (PC) = Top-of-Stack
Status: None

Cycles: 2 Instruction Cycles
Binary Code: 00 0000 0000 1000
Code Example: "RETURN"
Operation Example:
      If (TOS) = 0x0017 and (PC) = 0x108F,
         "RETURN" gives (PC) = 0x0017.
Description:
      The program counter is filled with the address at the top of the stack.

RLF   ("Rotate Left file Through Carry")
Syntax: RLF   f,d
Operands: f, where $0 <= f <= 127$
           And d, where d = {0,1}
Operations: See Figure 5-7 in Chapter 5.
           (W) = result if d = 0
           (f) = result if d = 1
Status: C
Cycles: 1 Instruction Cycle
Binary Code: 00 1101 dfff ffff
Code Example: "RLF   POSITION,W" for d = 0
              "RLF   POSITION,F"   for d = 1
Operation Example:
  If (POSITION) = 0x8C and (W) = 0x02 and (C-flag) = 0,
  "RLF   POSITION,W" gives (W) = 0x18 and (C-flag) = 1 and (POSITION) =
                                         0x8C
  "RLF   POSITION,F" gives (W) = 0x02 and (C-flag) = 1 and (POSITION) =
                                         0x18
Description:
      See attached Figure. The results are placed in:
           (W) if d = 0
           (file) if d = 1

RRF   ("Rotate Right file Through Carry")
Syntax: RRF   f,d
Operands: f, where $0 <= f <= 127$
           And d, where d = {0,1}
Operations: See Figure 5-7 in Chapter 5.
           (W) = result if d = 0
           (f) = result if d = 1
Status: C
Cycles: 1 Instruction Cycle
Binary Code: 00 1100 dfff ffff
Code Example: "RRF   POSITION,W" for d = 0
              "RRF   POSITION,F"   for d = 1
Operation Example:

If (POSITION) = 0x8D and (W) = 0x02 and (C-flag) = 0,
"RRF   POSITION,W" gives (W) = 0x46 and (C-flag) = 1 and (POSITION) =
0x8D
"RRF   POSITION,F" gives (W) = 0x02 and (C-flag) = 1 and (POSITION) =
0x46

Description:
See attached Figure. The results are placed in:
(W) if d = 0
(file) if d = 1

Diagram of RLF and RRF Instructions
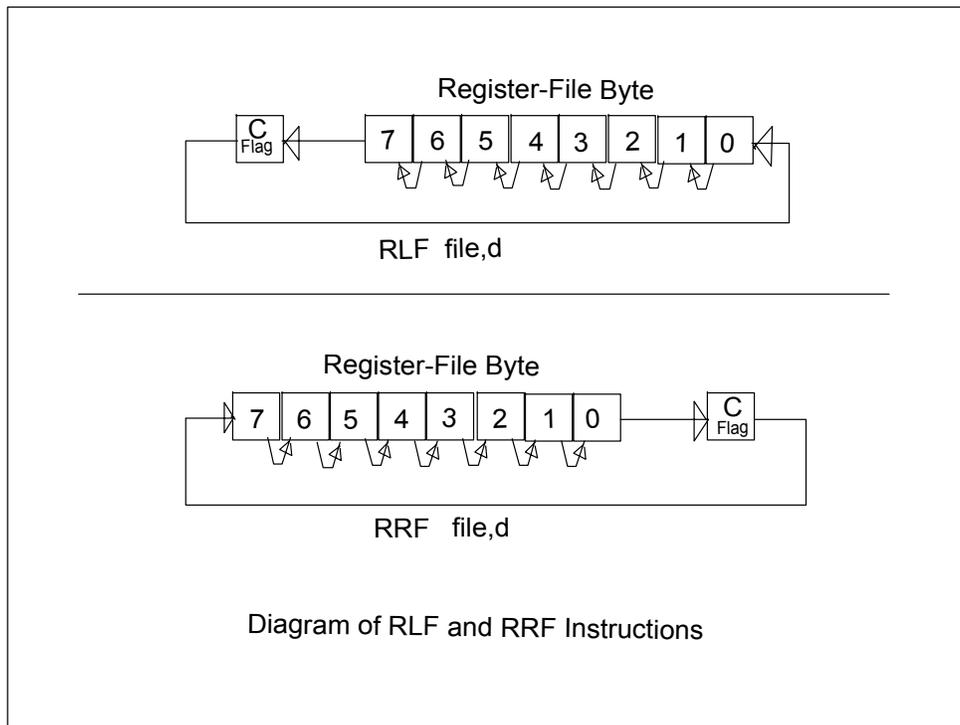
SLEEP   ("Enter Sleep Mode")
Syntax: SLEEP
Operand: None
Operation: (WDT) = 0x00, (WDT Prescaler) = 0x00,
                 STATUS flags: /TO = 1 and /PD = 0.
Status: None
Cycles: 1 Instruction Cycle
Binary Code: 00 0000 0110 0011
Code Example: "SLEEP"
Operation Example:

Doing "SLEEP" sends the CPU into sleep mode.
Description:
    Doing this instruction sends the CPU into sleep mode.

SUBLW   ("Subtract W from Literal")
Syntax: SUBLW   k
Operand: k, where 0 <= k <= 255
Operation: (W) = k – (W)
Status: C, DC, Z
Cycles: 1 Instruction Cycle
Binary Code: 11 110x kkkk kkkk
Code Example: "SUBLW 7"
Operation Example: If (W) = 3 and "ADDLW   7" is executed, (W) = 4.
Description:
        The contents of the W register are subtracted from the constant, k,
        And the results are placed in the W register.

SUBWF   ("Subtract W from file")
Syntax: SUBWF   f,d
Operands: f, where 0 <= f <= 127
                 And d, where d = {0,1}
Operation: (W) = (file) – (W)   if d = 0
                 (file) = (file) – (W) if d = 1
Status: C, DC, Z
Cycles: 1 Instruction Cycle
Binary Code: 00   0010 dfff   ffff
Code Examples: "SUBWF   SUM,W"   for   d = 0
   "SUBWF   SUM,F"   for   d = 1
Operation Examples:
            If (W) = 10 and (SUM) = 25,
                   "SUBWF   SUM,W" gives (W) = 15 and (SUM) = 25
                   "SUBWF   SUM,F"   gives (W) = 10 and (SUM) = 15
Description:
        The contents of the W register are subtracted from the contents of the
        Register-file and the results are placed in:
                (W) if d = 0
                (file) if d = 1

SWAPF   ("Swap Nibbles in file")
Syntax: SWAPF   f,d
Operands: f, where 0 <= f <= 127
                   And d, where d = {0,1}
Operations: (W) = (f<3:0>,f<7:4>) if d = 0
                   (f) = (f<3:0>,f<7:4>) if d = 1
Status: None
Cycles: 1 Instruction Cycle
Binary Code: 00 1110 dfff   ffff
Code Examples: "SWAPF   TEMP,W"   for d = 0
                      "SWAPF   TEMP,F"   for d = 1
Operation Examples:
      If (W) = 0xC7 and (TEMP) = 0xA9,
        "SWAPF   TEMP,W" gives (W) = 0x9A and (TEMP) = 0xA9
        "SWAPF   TEMP,F" gives (W) = 0xC7 and (TEMP) = 0x9A
Description:
      This instruction swaps the high and low nibbles of the register-file
      And places the results in:
             (W) if d = 0
             (file) if d = 1

XORLW   ("Exclusive Or Literal and W")
Syntax: XORLW   k
Operand: k, where 0 <= k <= 255
Operation: (W) = (W) .XOR. k
Status: Z
Cycles: 1 Instruction Cycle
Binary Code: 11 1010 kkkk kkkk
Code Example: "XORLW   0x07"
Operation Example: If (W) = 0xE6 and "XORLW   0x07" is executed,
                   (W) = 0xE3.
Description:
      The contents of the W register are XORed with the constant, k, and the
      Results are placed in the W register.

XORWF   ("Exclusive Or W and file")
Syntax: XORWF   f,d
Operands: f, where 0 <= f <= 127
                And d, where d = {0,1}
Operation: (W) = (W) .XOR. (file)   if d = 0
                (file) = (W) .XOR. (file)   if d = 1
Status: Z
Cycles: 1 Instruction Cycle
Binary Code: 00   0110 dfff   ffff
Code Examples: "XORWF   SUM,W"   for   d = 0
                   "XORWF   SUM,F"   for   d = 1
Operation Examples:
                If (W) = 0xC7 and (SUM) = 0xE5,
                "XORWF   SUM,W" gives (W) = 0x22 and (SUM) = 0xE5
                "XORWF   SUM,F"   gives (W) = 0xC7 and (SUM) = 0x22
Description:
        The contents of the W register are XORed to the contents of the
        Register-file and the results are placed in:
                (W) if d = 0
                (file) if d = 1

Appendix B --- Useful C++ Programs for Developing PIC ASM Applications

1) PERMS.CPP --- Permutation Hash Tables

2) SINEH.CPP --- Sine / Cosine Tables

3) OPENLOOP.CPP --- "Open-Loop" Time-Measurement

```
//============================================================
// Program:  PERMS.CPP  == Generate Permutation Tables
//                          in PIC16F877 Assembly Language.
//
// Author:   Timothy D. Green
// Date:     26 NOV 2005
//
// Compiler: Turbo C++ for DOS, Rev. 3.0
//
// This Program May Be Copied Freely
// It is in the Public Domain.
//
// The User of this Program does so at his or her own risk
// and bears the full responsibility thereof.
// The author, Timothy D. Green, assumes no liability
// for the use of this program.
//
//============================================================

#include<stdio.h>
int main(void)
{
  unsigned int    k,Map[8],Power[8],N_Bits,Test,Value,N_Table,j,m;
  FILE            *out;

  printf("Enter Number of Bits = ?  ");
  scanf("%d",&N_Bits);

  for(k=0,m=1;k < 8;k++,m=2*m) Power[k] = m;

  N_Table = 1;
  k = N_Bits;
  while(k > 0){
    N_Table = 2 * N_Table;
    k--;
  }

  printf("\n\n\n");
  for(k=0;k < N_Bits;k++){
    printf("For Input Bit = %d: What is the Output Bit Number = ? ",k);
    scanf("%d",&m);
    Map[k] = m;
    printf("\n");
  }

  out = fopen("PICPERM.ASM","w");

  for(k=0;k < N_Table;k++){
    Value = 0;
    for(m=0;m < N_Bits;m++){
      Test = Power[m];
      Test = Test & k;
      if(Test > 0) Value = Value + Power[Map[m]];
    }
    Value = Value & 0x00ff;
```

```
      fprintf(out,"        RETLW        %d\n",Value);
   }

   fclose(out);
   return 0;
}




//============================================================
// Program:  SINEH.CPP  == Generate Sine or Cosine Tables
//                         in PIC16F877 Assembly Language.
//
// Author:   Timothy D. Green
// Date:     26 NOV 2005
//
// Compiler: Turbo C++ for DOS, Rev. 3.0
//
// This Program May Be Copied Freely
// It is in the Public Domain.
//
// The User of this Program does so at his or her own risk
// and bears the full responsibility thereof.
// The author, Timothy D. Green, assumes no liability
// for the use of this program.
//
//============================================================

#include<stdio.h>
#include<math.h>

int main(void)
{
   char    H,L,Hex[16];
   char              S;
   int               k,N,j;
   double            Rk,Pi,Peak,RN,Angle,Sine;
   FILE              *out;

//=================

   Peak = 127.5;
   N = 128;

//=================

   RN = (double) N;
   Pi = 3.1415926535;
   for(k=0;k < 10;k++) Hex[k] = '0' + k;
   for(k=10,j=0;k < 16;k++,j++) Hex[k] = 'A' + j;

//=================

   out = fopen("PICSINE.ASM","w");

   for(k=0;k < N;k++){
```

```
    Rk = (double) k;
    Angle = (2.0 * Pi * Rk) / RN;
    Sine = Peak * sin(Angle);
    S = (char) Sine;
    L = S & 0x0f;
    H = ((S & 0xf0) >> 4) & 0x0f;
    L = Hex[L];
    H = Hex[H];
    fprintf(out,"        RETLW    0x%c%c\n",H,L);
  }

  fclose(out);

  return 0;
}




//===========================================================
// Program:  OPENLOOP.CPP  == Generate "Open-Loop" Code
//                            in PIC16F877 Assembly Language.
//
// Author:   Timothy D. Green
// Date:     26 NOV 2005
//
// Compiler: Turbo C++ for DOS, Rev. 3.0
//
// This Program May Be Copied Freely
// It is in the Public Domain.
//
// The User of this Program does so at his or her own risk
// and bears the full responsibility thereof.
// The author, Timothy D. Green, assumes no liability
// for the use of this program.
//
//===========================================================

#include<stdio.h>


int main(void)
{
  int               Start,Stop,k;
  FILE              *out;

//=================

  Start = 0;
  Stop = 100;

//=================

  out = fopen("PICLOOP.ASM","w");
```

```
    for(k=Start;k <= Stop;k++){
      fprintf(out,"        BTFSS    PORTC,TEST_BIT\n");
      fprintf(out,"        RETLW    %d\n",k);
    }

    fclose(out);

    return 0;
}
```

Appendix C --- Special Function Registers
(RAM Addresses & Bits)

STATUS (Address 0x03, In All Banks)
7   IRP   Indirect Addressing Bank Select [0 = Banks(0,1), 1 = Banks(2,3)]
6   RP1   Bank Select: 11=Bank(3), 10=Bank(2), 01=Bank(1), 00=Bank(0)
5   RP0   Bank Select
4   /TO   Time-Out Bit 1=(Power-up,CLRWDT,SLEEP) 0=WDT Overflow
3   /PD   Power-Down Bit 1=(Power-up,CLRWDT) 0=SLEEP
2   Z   Zero Bit   1=Result was Zero   0=Result was NOT Zero
1   DC   Digit Carry Bit   =Carry-State Out of 4th Low Bit of Result
0   C   Carry Bit   =Carry-State Out of Result Most Significant Bit

PCON (Address 0x8E, Bank 2 Only) (Also, Uses only Bits 1 & 0)
1   /POR   Power-On Reset Status: 1= No Reset   0=Reset
0   /BOR   Brown-Out Reset Status: 1=No Reset   0=Reset

INTCON (Address 0x0B, In All Banks)
7   GIE   Global Interrupt Enable Bit
6   PEIE   Peripheral Interrupt Enable Bit
5   T0IE   Timer0 Interrupt Enable Bit
4   INTE   RB0/INT External Interrupt Enable Bit
3   RBIE   PortB-Change Interrupt Enable Bit
2   T0IF   Timer0 Interrupt Flag
1   INTF   RB0/INT External Interrupt Flag
0   RBIF   RB0/INT External Interrupt Flag

PIR1 (Address 0x0C, Bank 0 Only)
7   PSPIF   Parallel Slave Port Interrupt Flag (Set on Read/Write)
6   ADIF   ADC Interrupt Flag (Set when Conversion is Done)
5   RCIF   USART Receive Interrupt Flag
4   TXIF   USART Transmit Interrupt Flag
3   SSPIF   Synchronous Serial Port (SSP) Interrupt Flag
2   CCP1IF   CCP1 Interrupt Flag (Capture Mode or Compare Mode)
1   TMR2IF TMR2-to-PR2 Match Interrupt Flag
0   TMR1IF   TMR1 Overflow Interrupt Flag

PIR2 (Address 0x0D, Bank 0 Only)
6   Reserved --- Never Write to This Bit
4   EEIF   EEPROM Write-Operation Interrupt Flag
3   BCLIF   Bus Collision Interrupt Flag
0   CCP2IF   CCP2 Interrupt Flag (Capture Mode or Compare Mode)

PIE1 (Address 0x8C, Bank 1 Only)
7   PSPIE   Parallel Slave Port Interrupt Enable Bit
6   ADIE   ADC Interrupt Enable Bit
5   RCIE   USART Receive Interrupt Enable Bit
4   TXIE   USART Transmit Interrupt Enable Bit
3   SSPIE   Synchronous Serial Port (SSP) Interrupt Enable Bit
2   CCP1IE CCP1 Interrupt Enable Bit
1   TMR2IE   TMR2 Interrupt Enable Bit
0   TMR1IE   TMR1 Interrupt Enable Bit

PIE2 (Address 0x8D, Bank 1 Only)
6   Reserved --- Never Write to This Bit
4   EEIE   EEPROM Write-Operation Interrupt Enable Bit
3   BCLIE   Bus Collision Interrupt Enable Bit
0   CCP2IE CCP2 Interrupt Enable Bit

OPTION-REG (Address 0x81, Banks 1 and 3)
7   /RBPU   PortB Pull-Up Enable Bit 1=Disable   0=Enable
6   INTEDG   Interrupt Edge Select Bit   1=Rising   0=Falling
5   T0CS   TMR0 Clock Source Select 1=RA4/T0CKI   0=Internal
4   T0SE   TMR0 Source Edge Select   1=Falling   0=Rising
3   PSA   Prescaler Assignment Bit   1=WDT   0=TMR0
2   PS2   (PS2:PS0)= See Chapter 8, Timer0 or WDT, Prescaler
1   PS1
0   PS0

ADCON0 (Address 0x1F, Bank 0 Only)
7-6   ADCS1:ADCS0   ADC Clock Select Bits
                00 = Fosc / 2
                01 = Fosc / 8
                10 = Fosc / 32
                11 = Internal RC-OSC for ADC
5-3   CHS2:CHS0 Analog Channel Select Bits
                (0,0,0) = Channel 0
                (0,0,1) = Channel 1
                (0,1,0) = Channel 2
                (0,1,1) = Channel 3
                (1,0,0) = Channel 4
                (1,0,1) = Channel 5
                (1,1,0) = Channel 6
                (1,1,1) = Channel 7
2   GO   ADC Start Conversion Bit (Set this to Start a Conversion)
0   ADON   ADC Activation Bit (Set this to turn the ADC unit ON)

ADCON1 (Address 0x9F, Bank 1 Only)
7   ADFM   ADC Result Format Bit   1=Right-Justified 0=Left-Justified
3-0   PCFG3:PCFG0 ADC Port Configuration Bits (See Chapter 8)

T1CON (Address 0x10, Bank 0 Only)
5-4   T1CKPS1:T1CKPS0   Timer1 Input Clock Prescale Select Bits
                (1,1) = 1:8 Prescale Value
                (1,0) = 1:4 Prescale Value
                (0,1) = 1:2 Prescale Value
                (0,0) = 1:1 Prescale Value
3   T1OSCEN   Timer1 Oscillator Enable Control Bit (1=Enable, 0=Disable)
2   /T1SYNC   Timer1 External Clock Sync Bit (1=No Sync, 0=Sync)
1   TMR1CS   Timer1 Clock Source Select (1=External Clock, 0=Internal)
                   (External Clock on RC0/T1OSO/T1CKI)
0   TMR1ON   Timer1 Activation Bit (1=Turn TMR1 On, 0=Turn Off)

T2CON (Address 0x12, Bank 0 Only)
6-3   TOUTPS3:TOUTPS0 Timer2 Output Postscale Select (See Chapter 8)
2   TMR2ON   Timer2 Activation Bit (1=Turn TMR2 On, 0=Turn Off)
1-0   T2CKPS1:T2CKPS0 Timer2 Clock Prescale Select (See Chapter 8)

CCP1CON (Address 0x17, Bank 0 Only)
5-4   CCP1X:CCP1Y   PWM LSBs (In Order: MSB-LSB)
3-0   CCP1M3:CCP1M0 CCP1 Mode Select Bits (See Chapter 8)

CCP2CON (Address 0x1D, Bank 0 Only)
5-4   CCP2X:CCP2Y   PWM LSBs (In Order: MSB-LSB)
3-0   CCP2M3:CCP2M0 CCP2 Mode Select Bits (See Chapter 8)

TRISE (Address 0x89, Bank 1 Only)
7   IBF   Parallel Slave Port, Input Buffer Full (See Chapter 8)
6   OBF   Parallel Slave Port, Output Buffer Full (See Chapter 8)
5   IBOV PSP, Input Buffer Overflow Detect Bit (See Chapter 8)
4   PSPMODE   (1=PSP Mode, 0= General I/O)
2   RE2 Direction Bit (1=Input, 0=Output)
1   RE1 Direction Bit (1=Input, 0= Output)
0   RE0 Direction Bit (1=Input, 0=Output)

EECON1 (Address 0x8C, Bank 3 Only)
7   EEPGD   Program/Data Memory Select Bit (1=FLASH, 0=EEPROM)
3   WRERR EEPROM Error Flag Bit (1=Write-Op Prematurely Terminated)
                              (0=Write-Op OK)
2   WREN   EEPROM Write-Enable Bit (1=Enable, 0=Disable)
1   WR   Start Write Process (Automatically Cleared by Hardware)
0   RD   Start Read Process (Auto Cleared by Hardware)

TXSTA (Address 0x98, Bank 1 Only)
7   CSRC   Clock Source Select Bit (See Chapter 9)
6   TX9   9-Bit Transmit Enable Bit (See Chapter 9)
5   TXEN   Transmit Enable Bit (1=Enabled, 0=Disabled)
4   SYNC   USART Mode Select (1=Synchronous, 0=Asynchronous)
2   BRGH   High Baud Rate Select Bit (1=High Speed, 0=Low Speed)
1   TRMT   Transmit Shift Register Status Bit (1=TSR Empty, 0=TSR Full)
0   TX9D 9th Bit of Transmitted Data

RCSTA (Address 0x18, Bank 0 Only)
7   SPEN   Serial Port Enable Bit (1=Enable USART, 0=Disable USART)
6   RX9   9-Bit Receive Enable Bit (1=Enable, 0=Disable)
5   SREN   Single Receive Enable Bit (See Chapter 9)
4   CREN   Continuous Receive Enable Bit (See Chapter 9)
3   ADDEN Address Detect Enable Bit (See Chapter 9)
2   FERR   Frame Error Status Bit (1=Error, 0= No Error) (See Chapter 9)
1   OERR   Overrun Error Bit (1=Error, 0=No Error) (See Chapter 9)
0   RX9D 9th Received Data Bit

SSPCON (Address 0x14, Bank 0 Only)
7   WCOL   Write Collision Detect Bit (See Chapter 9)
6   SSPOV   Receive Overflow Indicator Bit (See Chapter 9)
5   SSPEN   Synchronous Serial Port Enable Bit (1=Enable, 0=Disable)
4   CKP   Clock Polarity Select Bit (See Chapter 9)
3-0   SSPM3:SSPM0 Synchronous Serial Port Mode Select Bits
                          (See Chapter 9)

SSPCON2 (Address 0x91, Bank 1 Only)
7   GCEN   General Call Enable Bit (See Chapter 9)
6   ACKSTAT   ACKN Status Bit (See Chapter 9)
5   ACKDT   ACKN Data Bit (See Chapter 9)
4   ACKEN   ACKN Sequence Enable Bit (See Chapter 9)
3   RCEN   Receive Enable Bit (See Chapter 9)
2   PEN   "STOP" Condition Enable Bit (See Chapter 9)
1   RSEN   Repeated Start Enable Bit (See Chapter 9)
0   SEN   "START" Condition Enable Bit (See Chapter 9)

SSPSTAT (Address 0x94, Bank 1 Only)
7   SMP   Sample Bit (See Chapter 9)
6   CKE   SPI Clock Edge Select Bit (See Chapter 9)
5   DA   Data/Address Bit (See Chapter 9)
4   P   "STOP" Bit (See Chapter 9)
3   S   "START" Bit (See Chapter 9)
2   RW   Read/Write Bit Information (See Chapter 9)
1   UA   Update Address (See Chapter 9)
0   BF   Buffer Full (Auto Cleared When SSPBUF is Read) (See Chapter 9)

Appendix D --- PIC16F877 Register File Map

ADDR	Bank 0	Bank 1	Bank 2	Bank3
0x00	INDF	INDF	INDF	INDF
0x01	TMR0	OPTION_REG	TMR0	OPTION_REG
0x02	PCL	PCL	PCL	PCL
0x03	STATUS	STATUS	STATUS	STATUS
0x04	FSR	FSR	FSR	FSR
0x05	PORTA	TRISA	====	====
0x06	PORTB	TRISB	PORTB	TRISB
0x07	PORTC	TRISC	====	====
0x08	PORTD	TRISD	====	====
0x09	PORTE	TRISE	====	====
0x0A	PCLATH	PCLATH	PCLATH	PCLATH
0x0B	INTCON	INTCON	INTCON	INTCON
0x0C	PIR1	PIE1	EEDATA	EECON1
0x0D	PIR2	PIE2	EEADR	EECON2
0x0E	TMR1L	PCON	EEDATH	====
0x0F	TMR1H	====	EEADRH	====
0x10	T1CON	====		
0x11	TMR2	SSPCON2		
0x12	T2CON	PR2		
0x13	SSPBUF	SSPADD		
0x14	SSPCON	SSPSTAT		
0x15	CCPR1L	====		
0x16	CCPR1H	====		
0x17	CCP1CON	====		
0x18	RCSTA	TXSTA		
0x19	TXREG	SPBRG		
0x1A	RCREG	====		
0x1B	CCPR2L	====		
0x1C	CCPR2H	====		
0x1D	CCP2CON	====		
0x1E	ADRESH	ADRESL		
0x1F	ADCON0	ADCON1		

Notes:
1) Entries marked "====" are reserved – do not use them.
2) Blank Entries are Usable as RAM
3) RAM Addresses from 0x20-Through-0x7F Are Freely Available
4) RAM from 0x70-through-0x7F Do Not Need To Switch Banks
5) Indirect Addresses Must Add 0x80 For Bank 1 & Bank 3

Appendix E --- PIC16F877 Pin Function Map

PIN #	Functions	PIN #	Functions
1	/MCLR, Vpp	40	RB7, PGD
2	RA0, AN0	39	RB6, PGC
3	RA1, AN1	38	RB5
4	RA2, AN2, Vref-	37	RB4
5	RA3, AN3, Vref+	36	RB3, PGM
6	RA4, T0CKI	35	RB2
7	RA5, AN4, /SS	34	RB1
8	RE0, AN5, /RD	33	RB0, INT
9	RE1, AN6, /WR	32	Vdd (+5 Volts)
10	RE2, AN7, /CS	31	Vss (Ground)
11	Vdd (+5 Volts)	30	RD7, PSP7
12	Vss (Ground)	29	RD6, PSP6
13	OSC1, CLKIN	28	RD5, PSP5
14	OSC2, CLKOUT	27	RD4, PSP4
15	RC0, T1OSO, T1CKI	26	RC7, RX, DT
16	RC1, T1OSI, CCP2	25	RC6, TX, CK
17	RC2, CCP1	24	RC5, SDO
18	RC3, SCK, SCL	23	RC4, SDI, SDA
19	RD0, PSP0	22	RD3, PSP3
20	RD1, PSP1	21	RD2, PSP2

Appendix F --- Save Register / Restore Registers on Interrupt

The code to save the W and STATUS registers at the start of an interrupt is as follows:

```
W_TEMP:    EQU        0x70        ; Storage for W Register
STATUS_T:  EQU        0x71        ; Storage for STATUS Register

           MOVWF      W_TEMP     ; Save W
           SWAPF      STATUS,W   ; Get STATUS into W
           MOVWF      STATUS_T   ; Save STATUS
```

The code to restore the same is as follows:

```
           SWAPF      STATUS_T,W   ; Get Saved STATUS
           MOVWF      STATUS       ; Restore STATUS
           SWAPF      W_TEMP,F      ; Restore W
           SWAPF      W_TEMP,W
```

Note: The process of saving and restoring the registers can be simplified by using assembly language MACROs. These are called "PUSH" and "POP" and are defined as follows:

```
PUSH:      MACRO
           MOVWF      W_TEMP     ; Save W
           SWAPF      STATUS,W   ; Get STATUS into W
           MOVWF      STATUS_T   ; Save STATUS
           ENDM


POP:       MACRO
           SWAPF      STATUS_T,W   ; Get Saved STATUS
           MOVWF      STATUS       ; Restore STATUS
           SWAPF      W_TEMP,F      ; Restore W
           SWAPF      W_TEMP,W
           ENDM
```

To use these MACROs, do as follows:

```
INT_SERVICE:
            PUSH            ; Inserts the "PUSH" MACRO's Code Here
            ---- Do The Service ------
            POP             ; Inserts the "POP" MACRO's Code Here
            RETFIE
```

The MACRO-code is translated verbatim into these places where they are called. They are *NOT* subroutines! The code is just repeated in each place where the MACRO is called.

References

1) All material covering DSP and the Median filter:
    Digital Signal Processing: Theory, Applications, and Hardware
    By Richard A. Haddad & Thomas W. Parsons
    Copyright 1991 by W. H. Freeman and Company

2) All material & facts about the PIC16F877:
    PIC16F87X Data Book
    MPLAB Manual
    MPASM Manual
    Copyright 2001 by Microchip Technology, Inc.

3) Material on FSK, ADPCM, PWM, and Manchester:
    Data and Computer Communications
    By William Stallings
    Copyright 1994 by Macmillian Publishing Company

    And

    Speech Coding: A Computer Laboratory Textbook
    By Barnwell, Nayebi, and Richardson
    Copyright 1996 by John Wiley & Sons, Inc.

4) Material on microprocessor & microcontroller interfacing:
    Interfacing
    By Stephen E. Derenzo
    Copyright 1990 by Prentice-Hall, Inc.

5) Material on Dithering (Chapter 7):
    FAB
    By Neil Gershenfeld
    Copyright 2005 by Basic Books, A Member of the Perseus Books Group

6) Material on Direct Digital Synthesis (Chapter 7):
    ARRL Handbook (of 1986)
    Copyright 1985 by the American Radio Relay League

7) Material on Speech Compression (Chapter 10):
    Time-Compressed Speech, Volumes I, II, & III
    By Sam Duker
    Copyright 1974 by Sam Duker