INTRODUCTION TO PROGRAMMING

Disclaimer: THIS SOFTWARE IS PROVIDED "AS IS" WITHOUT ANY EXPRESS OR IMPLIED WARRANTY OF ANY KIND. MICROCHIP DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT OF INTELLECTUAL PROPERTY OR OTHER VIOLATION OF RIGHTS. MICROCHIP AND ITS SUPPLIERS DO NOT WARRANT OR MAKE ANY REPRESENTATIONS REGARDING THE ACCURACY, COMPLETENESS OR RELIABILITY THE SOFTWARE ON THIS CD-ROM, INCLUDING THE INFORMATION, TEXT, GRAPHICS, LINKS OR OTHER ITEMS CONTAINED WITHIN THE MATERIALS. MICROCHIP MAY MAKE CHANGES TO THE SOFTWARE AT ANY TIME WITHOUT NOTICE. MICROCHIP HAS NO OBLIGATION TO UPDATE THE SOFTWARE.

Limitation of Liability: IN NO EVENT, INCLUDING, BUT NOT LIMITED TO NEGLIGENCE, SHALL MICROCHIP OR ITS SUPPLIERS BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, INCLUDING, WITHOUT LIMITATION, BUSINESS INTERRUPTION OR LOSS OF DATA OR PROFIT, ARISING OUT OF THE USE OR INABILITY TO USE, THIS SOFTWARE, EVEN IF MICROCHIP HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

CONTENTS AT A GLANCE

Numbering Systems

Data Types

Programming Basics

High-Level Languages and Structured Programming
THE BASIC LANGUAGE
THE C LANGUAGE

Assembly Language Programming

Interrupts

Fuzzy Logic

Event-Driven Programming

State Machines

The best programming language is solder."

Remark attributed to Steve Ciarcia.

Despite Steve Ciarcia's assertion that applications are best designed with a hot piece of metal and molten lead, microcontroller applications and most other modern electronics applications require you to write some software for them.

I have found that many people starting out in electronics are reluctant to include device programming. I believe that the reasons for this reticence is caused by the approach with which most people are taught programming and its disassociation with hardware. Most people learn programming on PCs or workstations running Microsoft Windows, UNIX (Linux), or other sophisticated operating systems. The student is generally told not to worry about what kind of code is generated by the compiler and how it works in the system's processor.

This appendix goes through the basics of programming, describing both *High-Level Languages (HLLs)* and assembly-language programming from the perspective of the processor. I believe this makes it easier to visualize how an application is executing.

Along with this, I present interrupt handling and event-driven programming from a processor-independent perspective to show how these techniques work together and suggest how they can be used for microcontroller application programming.

Numbering Systems

In any kind of programming, you will be inundated with different numbering systems in different formats. This can make understanding what is happening in your application difficult or the data that you are seeing is confusing. Having a good understanding of the different numbering systems and how data is converted between them is crucial to being able to develop a working application and debugging one that isn't.

Some of the information presented here is a repeat of what I described in "Introduction to Electronics." This section focuses on the information that I feel is important for understanding programming and interfacing to hardware devices. After going through both this section and "Introduction to Electronics" section, you should have a very good understanding in the basic mathematics needed to develop microcontroller applications both from the hardware and software perspectives.

All of our lives, we have been taught to work in the Base 10. *Base 10* means that numbers are represented by the 10 digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. A number greater than nine needs to be written down and multiple digits are used with each digit representing the power of 10 of the digit. For example, 123 is one hundreds (10 to the power 2), two tens (10 to the power 1) and three ones (10 to the power 0). It could be written out as:

```
123 = (1 × 100) + (2 × 10) + (3 × 1)
    = [1 × (10^2)] + [2 × (10^1)] + (3 × (10^0)]
```

Each digit is a number is given a position starting from the rightmost digit (working left and numbered starting with 0). The digit position is actually the exponent. The base is multiplied by to get the value for the digit. When you learned Base-10 mathematics, each digit was referred to as *ones*, *tens*, *hundreds*, etc., instead of using the exponent to base 10 (which these values actually are).

When you work with computers and want to change specific digits, you will find it easier to work with numbers in terms of the base's exponent, rather than the actual value. I am making this point because numbers in a computer are represented as binary values with each digit two times (in base) greater than the digit to its right.

In a computer, numbers are represented by a series of bits, where a bit (digit) can have the value *0* or *1*. Binary values are represented as a series of ones or zeros with each digit representing a power of two. Using this system, B'101100' can be converted to decimal using the binary exponent of each digit.

```
B'101100' = [1 × (2^5)] + [0 × (2^4)] + (1 × (2^3)] + [1 × (2^2)]
            [0 × (2^1)] + [0 × (2^1)] (Decimal)
          = 32 + 8 + 4 (Decimal)
          = 44 (Decimal)
```

For convenience, binary numbers are normally written as hexadecimal digits instead of individual bits. Hexadecimal digits combine four bits into a single value. The numbers *0* to *9* and *A* to *F* are used to represent each hex digit. For multidigit hex numbers, each digit is multiplied by 16 to the appropriate power:

```
0x0123 (hex) = [1 × (16^2)] + [2 × (16^1)] + [3 × (16^0)] (Decimal)
             = 256 + 32 + 3 (Decimal)
             = 291 (Decimal)
```

When working with the different numbering systems in this book, I use three different formats to let you know which numbering system is being used. When the number is enclosed in single quotes (') and prefixed by a *B* or prefixed by the string *0b0*, then the number is a binary value. When *0x0* is in front of the number, then the number is in hexadecimal format. If the number isn't modified in any way, then it is decimal (Table 1).

These formats are similar to what are used when programming the PICmicro® MCU in Microchip assembler. The only prefix that isn't available in the PICmicro® MCU's assembler is *0b0* for binary. This book is consistent with these formats to avoid confusion over what is the actual value of a decimal string like *101*. Unless one of the prefixes is specified, then the number is decimal.

As an aside, as you probably know, *hex* is the prefix for six, not 16. The correct prefix for 16 is *sex* and in the early days of programming, base 16 numbers were known as *sexadecimal*. When IBM released the System 360, all the documentation referred to base 16 num-

TABLE 1 Number Format and Types Used in this Book

NUMBER FORMAT	NUMBER TYPE
0b0 . . .	Binary
B' . . .' or 0b0 . . .	Binary
0x0 . . .	Hexadecimal
No Prefix	Decimal

bers as *hexadecimal*. This change was made because IBM felt that the System 360 would be used by a large group of different users, some of whom would object to the word "sex" as part of the basic programming operation of the computer. To be fair, I'm sure many of the early programmers snickered at the term *sexadecimal,* leading to the feeling that a reputable computer company would not sell a machine which was programmed in *sex*.

Along with decimal, binary, and hex numbers, there are also a base-eight numbering system known as *octal*. Octal numbers were popularized by DEC (in their VAX line of mini-computers) and the C language. Octal numbers are awkward to work with because each digit requires three bits. A byte cannot be represented evenly with octal numbers: three digits are required for a byte. For the top digit, only two bits are available. Most assemblers and high-level languages support octal, but it should really be avoided simply because it does not fit in very well with binary and hexadecimal.

To help make creating binary numbers with specific bits set easier, might I suggest shifting the value "1" to the left by a specified number of bits instead of converting the bit number to hexadecimal or decimal values. This is done by using the "<<" (shift left) operator:

```
1 << Bit#
```

rather than trying to figure out which bit is which value. I find it easier to shift one up by the bit digit number. Thus to create a byte constant where bit 4 is set, I could use the expression:

```
1 << 4
```

This is a very useful trick for PICmicro® MCU assembly-language programming, where individual bits are set and reset for hardware control. Constants with specific bit settings can be created much more quickly and accurately using this trick than creating a binary number from the bits you want to set and converting to hexadecimal or decimal.

To reset bits, the same formula can be used to reset bits in a byte, but the resulting bits of the constant has to be inverted. The expression to do this is:

```
0x0FF ^ (1 << Bit#)
```

For example, if a byte with bit 3 reset were required, the expression would become:

```
0x0FF ^ (1 << 3) = 0x0FF ^ 8
                 = 0x0F7
```

Despite little tricks like this, you will still have to convert numbers from one base to another as you are programming. To aid in this operation, I recommend that you buy a calcu-

Figure 1

lator that has base conversion, as well as bitwise operations (AND, OR, XOR, and NOT), capabilities. I have owned two HP-16C calculators (Fig. 1) for more than 15 years (one for home and one for work) and they are two of the best investments I have ever made.

Some modern calculators with numbering system conversion (and bitwise operations) are:

```
Texas Instruments: TI-85, TI-86
Sharp: EL-546L, EL-506L
```

As I was proofreading this book, I discovered that my HP-20S could convert decimal values to hex and binary numbers. I only noticed the Hex, Bin, Dec, and Oct key selections (along with A, B, C, D, E, and F) when my eyes were wandering when I was trying to come up with a better way of saying something as I was writing this section. The numbering conversions tend to be an undocumented feature of the calculator and not included in the instructions. For many of these calculators, there doesn't seem to be any way to do bitwise operations (AND, OR, and XOR), which is unfortunate because these operations would allow you to "simulate" the operations of the processor.

When writing application software for the PICmicro® MCU, you will have to be familiar with how mathematical operations work. For addition and multiplication for integer operations are quite simple to understand. When subtracting numbers, if a digit is greater than the one above it, the number base is "borrowed" from the digit to the left of it.

To subtract 15 from 22, when subtracting the ones, five is greater than 2, and 10 is borrowed from the tens:

```
 22        10 + 12
-15  =>  -(10 +  5)
           0 +  7
```

If the magnitude of the number subtracted is greater than the value it is subtracted from, then a negative number results:

```
 22
-25
---
 -3
```

In a computer processor, there is no such thing as a negative number, instead when a value is subtracted from a small value, the borrowed values results in a two's complement negative number. This can be shown in the previous example, by converting the two decimal numbers into binary:

```
 22       00010110
-25 =>  -00011001
```

For the right-most digit (bit 0), *1* cannot be taken away from *0,* so the digit to the left (bit 1) is borrowed from. In this case, *2* is borrowed from bit 1 and subtracted from bit 1. The result of one subtracted from two is placed as the result.

For bit 3, the *1* in the number being subtracted has to borrow from bit 4, leaving bit 4 without having anything subtracted from it. Because the higher bits are all zero, this is not possible, so the bits are checked to the end of the byte (bit 7) until a bit is found to borrow from. In most computers (like the PICmicro® MCU), if no other bits are available to be borrowed from, the hardware behaves as if the first bit that is larger than the most-significant bit of the byte is set. So, in actuality, the subtraction is:

```
 1 00010110        1 00010110
-0 00011001 =>  - 0 00011001
                    11111101
```

The value *B'11111101'* is the *two's complement* representation of −3. This value, when added to a positive value or another two's complement negative number, will give a valid result.

For example, when −3 is added to 5, the result is 2. Using the two's complement representation of −3, this can be shown with binary numbers:

```
-3       11111101
+5 =>  + 00000101
 2     1 00000010
```

Like the virtual ninth bit made available for the borrow, the ninth bit produced is ignored (or used as the carry in the processor). With the ninth bit ignored, the result is B'00000010' or 2 (Decimal).

Two's complement negative numbers can be generated from positive numbers using the formula:

```
Two's complement negative = (Positive ^ 0x0FF) + 1
```

for −3, this formula can be used to generated the bit pattern:

```
-3 = (3 ^ 0x0FF) + 1
   = (B'00000011' ^ B'11111111) + 1
   = B'11111100' + 1
   = B'11111101'
```

which is the value calculated previously.

With two's complement negative numbers, the most-significant bit (bit 7 in a byte) is often referred to as the *sign bit*. If the sign bit is set, the number is negative. If it is reset, the number is positive. The remaining seven bits are the number's magnitude. Two's complement bytes can be in the range of −128 to +127.

Now that I've shown that computer processors can handle positive and negative two's complement numbers, I have to point out that most processors (including the PICmicro® MCU's) *cannot* handle negative numbers "natively" (or without any special instructions or software). Two's complement negative numbers are a representation of a negative number—they are not actually "negative numbers" to the processor. The difference is subtle, but important. The computer processor only works with bit patterns.

Another popular numbering system is the *Binary Coded Decimal (BCD)* format. In this numbering system, each base-10 digit of a number is represented as a nybble. Using BCD, one byte can represent the decimal numbers *00* to *99*. BCD is quite popular for people working with "mainframe" processors, where an essentially unlimited memory is available and processor support is built in to handle the numbers natively. The PICmicro® MCU, like most other small microcontrollers, does not have this capability built into the processor and the limited memory makes this format not recommended for use in applications.

Representing a number in BCD is very straightforward. For example, the number 23 decimal is simply stored in a byte as 0x023. As noted, this means that 100 numbers can be represented in a byte. Compared to the possible 256 numbers using all eight bits of a byte without restriction, BCD cannot represent 60% of the values that binary numbers can. For larger numbers, this inefficiency becomes even more acute. For example, two bytes (which have four nybbles) can represent the 10,000 BCD values (0000 to 9999), where the two bytes could represent 65,536 values if the data is stored as straight binary. In the two byte case, the efficiency of BCD in representing all possible values has dropped to just over 15% compared to binary. For a limited-memory device like the PICmicro® MCU, this can be a serious problem.

Handling BCD numbers can be very difficult as well. Although in a later section of this appendix, I show how a two-bit BCD addition can be processed reasonably easily, I feel this effort should not be expended at all. Although addition is quite simple, subtraction is more complex (especially with handling negative numbers). Additional complexity comes into place if more than two digits per number are used or if multiplication or division operations are required on the BCD numbers.

In this book, instead of looking toward BCD data processing, I want to point you toward working only with bytes and use the binary-to-decimal conversion routines provided to handle input and display decimal values in your applications.

Data Types

Computer processors are really designed to manipulate "data." Data can be a flag, a sensor value, a name, or even your driver's license number. For each kind of data, different data formats, known as *types,* are optimal in terms of data-storage requirements. Before starting any computer code application, you should understand the types of data that are used in the application and how they will be saved.

The most basic unit of data is the bit. A *bit* is a single two-state memory cell. Two state means that a bit can either be on or off, yes or no, high or low, or any other pair of oppo-

site states that you can think of. The two-state memory bit is often referred to as a binary memory bit. The two-state memory bit is the basis for all computer memories and is combined into larger groups of data.

To make binary data easier to work with, every four bits are combined and written as a single character. These four bits are sometimes being referred to as a *nybble*. The nybble uses the characters *0* to *9* and *A* to *F* to represent the 16 different possible states stored in the four bits. Table 2 shows the characters represented for each of the different bit states.

Normally, nybbles are written as part of a hexadecimal (hex) number. For example, the eight bits:

```
0b011000110
```

could be written in the nybble/hex format as:

```
0x0C6
```

In this book (and most others), the hex/nybble format is the preferred method of listing binary data.

From this table of values, it is useful to note that the first number in any binary sequence

TABLE 2 Binary Number to Hexadecimal Digit Table

BITS	NYBBLE CHARACTER
(3-0)	
0000	"0"
0001	"1"
0010	"2"
0011	"3"
0100	"4"
0101	"5"
0110	"6"
0111	"7"
1000	"8"
1001	"9"
1010	"A"
1011	"B"
1100	"C"
1101	"D"
1110	"E"
1111	"F"

is always zero. This makes electrical engineers and computer scientists different animals from the rest of humanity. Whereas most people think the first number is *1,* we tend to think of it as *0* because that is the first valid state in a binary data sequence.

Eight bits are combined to form a byte, the basic unit of storage in most computer systems. The eight bits can store up to 256 (two to the power eight) different values. These values can either be numeric or characters.

Numeric values can be positive integers in the range of 0 to 255 (which is 0b011111111 or 0x0FF) or positive and negative integers in the range of −128 to +127.

When a byte is used for positive and negative values, the most significant bit (known as *bit 7*) is used as the *sign bit*. The remaining seven bits are the *magnitude bits*.

As an aside, bits in a byte (or other data type) are referred to by the power of two they bring to the number. For example, bit 0 is the *1*s bit, bit 1 is the *2*s bit, etc. (Table 3).

From this table, you should be able to see how the bits are combined to form a byte. For this example (0x0C6), to make up the number, the bits 1, 2, 5, and 6 of the byte are set.

Negative values are normally stored in two's complement format. To generate a two's complement negative number, the positive value is NOTted (inverted or complemented) and incremented.

For example, to calculate the hex value −47, the following process is used:

```
−47 = NOT (47) + 1
    = NOT (0b000101111) + 1
    = 0b011010000 + 1
    = 0b011010001
    = 0x0D1
```

The number NOT operation is the same as XORing the number with 0x0FF (as shown previously).

The advantage of using two's complement format is that it is automatically produced when a number is subtracted from a smaller value. Also, it can be added to a positive or negative two's complement number and the result will be correct (if it is positive or negative).

TABLE 3 Bit to Power of Two's for Bit to Hex Value Cross-Reference

BIT	TWO'S POWER OF BIT	"HEX" VALUE
0	1	0x001
1	2	0x002
2	4	0x004
3	8	0x008
4	16	0x010
5	32	0x020
6	64	0x040
7	128	0x080

For larger pieces of data, bytes are normally “concatenated” together to form the new data type. As I go through the book, I will explain that I prefer working with two bytes to make a number that is 16 bits in size. Sixteen-bit numbers have a greater range of two’s complement values (from −32,768 to +32,677) than a single eight-bit byte that has the numeric range −128 to +127.

Four bytes (32 bits) can be combined to provide even larger data values and is very common for PCs and workstations (although as I write this, 64 bits is becoming the norm).

Fractional or “real” data values can also be represented with binary memory. To store real data values, a format that is very analogous to the scientific notation you learned in high school, is used. This type of data could also be known as *floating point* or *real*.

For the microprocessor used in your PC, floating-point data is stored in IEEE Standard 754 format, which requires four or eight bits to store a floating-point number. The IEEE 754 double precision floating-point number is defined as:

$$s\ mmmm \times 2^{e-bias}$$

where *s* is the sign of the number, *mmmm* is known as the *mantissa* or *precision* of the value, *e* is the two’s complement exponent that denotes the order of magnitude of the mantissa. The exponent has a bias value subtracted from it that is used to compensate for the number of mantissa bits in the number.

This is probably very confusing, so let’s look at an actual example.

The number *6.5* could be represented as:

$$6.5 \times (10^0)$$

or without the decimal point as:

$$65 \times (10^{-1})$$

In the computer itself, these values would be represented as binary values instead of base 10. *6.5* could be represented as:

$$110.1 \times (2^0)$$

or without the decimal point as:

$$1101 \times (2^{-1})$$

To eliminate the decimal point in the data, I simply shift the mantissa to the left (and decrement the exponent) until the mantissa is entirely positive.

I chose 6.5 because the binary value and fraction are relatively easy to understand. This conversion is nontrivial for many other examples (such as 1,022.73).

The double precision data format is shown in Table 4. Before the value can be stored, the exponent has to be calculated with respect to the bias value. For the double precision numbers, this value is 1023. To find the correct exponent, the formula:

$$Required\ exponent = Saved\ exponent - Bias\ value$$

TABLE 4 IEEE 745 Double Precision Floating Point Data Format

BITS	PURPOSE
63	Sign Bit
62–52	Exponent
51–0	Mantissa

which, in this case, is:

```
−1 = Saved exponent − 1023
```

Moving the 1023 to the other side of the equation, saved exponent is 1022 (0x03FE). This data is then stored as the following eight bytes (16 nybbles):

```
0x03FE000000000000D
```

From this example, you should come to the conclusion that floating-point numbers are awfully hard to work with and can use a lot of memory (both in terms of storage as well as processing routines). Small eight-bit processors, like the one in the PICmicro® MCU, simply cannot process floating-point data efficiently. For these reasons, I don't present any applications that use floating-point numbers. Instead I present methods of converting binary data in such a way that floating-point numbers, like this, are completely avoided.

If you insist on working with floating-point numbers in the PICmicro® MCU, I suggest that you only work with high-level languages that have a floating-point data type supported as part of the language and library routines for processing the floating-point data built in. Life is too short to be playing around with floating-point numbers in an eight-bit processor programmed in assembler.

Bytes of characters can be used together to store character strings. These strings can be names, messages, lists, or whatever human-readable information that you require to save. This data is normally stored in such a way as to make it easily readable in a simulator or emulator display.

To simplify my applications, I save all my string data in ASCIIZ format. This is a string of bytes that are ended by the Null (0x000) character. When processing these strings, I simply read through them to the first 0x000. Lists of ASCIIZ strings can be put together with the end of the list being indicated by a final Null character after the last ASCIIZ string.

ASCIIZ strings are often simpler to work with than character strings with explicit lengths. This is especially true when creating string data for use in the source code. Changes in an ASCIIZ string do not require any other changes (so long as the ending Null character isn't lost). Other types of strings will require that the length indicators will have to be updated after the string. This is something that, historically, I'm not good at remembering to do. By using an ASCIIZ string, I have eliminated the need to remember to update the string length all together.

Data types can also be extended to include multiple data types built into a single "structure." For example, your Social Security record probably looks something like the structure shown in Table 5.

TABLE 5 Example Data Structure

SOCIALSECURITY_STRUCTURE
Bit32 Number
ASCIIZ "Your Name"
ASCIIZ "Your Address"
Float Income_1999
Float Tax_1999

Structures are useful in database programs and other applications that process large amounts of data. Structures are not very useful in microcontroller applications that do not typically process much data and do not have much memory to do it with.

This section throws a lot of information at you, with a number of various data types. I did this to give you an idea of what is possible in a computer application. All of these data types are used in a variety of systems that you interact with on a daily basis.

In microcontrollers, which have limited program memory and data memory, fancy and extensive data types are inappropriate. Throughout this book, only the simplest of the data types are presented. I suggest that you follow this example in your own application development.

This might require a mind shift on your part if you are already an experienced programmer, but keeping to the simplest data types possible will make learning to use the PICmicro® MCU easier and will make your application development easier as well.

Programming Basics

Many people have learned how to program in a variety of different ways. I, personally, first was taught on an assembly-language simulator using punch cards in high school. Like many introductory courses, the teacher spent the class trying to explain how programming was done while showing how the concepts are implemented on the training tool. This made it very hard for the students who are trying to learn two new concepts (programming and working with the computer) at the same time.

The basic programming concepts are very simple. In fact, I would say that there are only four of them. Once you understand these basics, you can then begin to use them in high-level languages followed by assembly-language programming. In this section, before describing high-level languages, I want to introduce these four concepts as well as two additional concepts that are applied to virtually all programming environments.

Before getting into the four programming concepts, what is programming?

As best I can define it:

Programming is the process of developing a series of symbols that will be converted, by a compiler or assembler, into a set of instructions for a computer processor.

I use this definition because it can be used to illustrate the steps used to develop an application. When you are programming, you are developing a dialog that is meaningless,

except in the context to which it is being used. This dialog is then treated as a series of text strings that are meaningless, except as input to develop instructions that will run on the final processor. The final step is the execution of the instructions without regard as to what the final application is doing.

From this perspective, *programming* is writing a text file for a computer application that will convert it into a format that will be used to control another computer. The second computer simply executes the instructions without any regard for what the programmer expects that it should be doing.

As you start programming, you are going to go through some extremely frustrating times. I've been programming for more than 20 years and I still get frustrated and angry when something doesn't work as I expect it to. The only thing all this experience has given me is that I have nothing and nobody to be angry with except myself.

When you try to compile or assemble your first PICmicro® MCU programs, chances are that the PC running MPLAB will output syntax errors. The error messages it will spit out at you probably won't make any sense and the lines that the errors point to will look perfect to you. The important thing to remember is: keep calm and try to develop different techniques to find the problems. This book explains many techniques that I use to find problems and pitfalls I've had along the way. But it still won't be easy.

Probably more frustrating will be the problems you encounter when the application source code compiles and assembles cleanly (without problems), but the application doesn't work. Like the source code compiler/assembler, the PICmicro® MCU processor is only executing what is has been given.

In university, we used to have the joke that went like:

First Student: How's your computer science project going?
Second Student: It works exactly according to how I wrote it.
First Student: So you're saying that it doesn't work at all.

It is a hard lesson to learn, but *you* are really the weakest link in the application-development process. It's easy to blame a problem on MPLAB, the particular PICmicro® MCU that you are using, or unusually intense sun spots, but when it comes right down to it (999 times out of 1,000), you are the source of the problem.

Once you can accept that the problem is of your own doing; you should be able to calmly work through the problem and find its cause.

The four basic concepts that you must understand to successfully develop your own applications are:

1. Array variables
2. Data assignment
3. Mathematical operators
4. Conditional execution

Once you understand these four concepts, the work in learning new programming languages and environments (processors) will be greatly simplified and you will become productive with them much faster.

Array variables probably is an overly complicated way to describe memory locations that are used to store application parameters, but it does explain the concept very well. In all computers, memory is arranged in an addressed block of bytes that can be read from or written to.

A visual model is often used is a set of *cubbies* into which sorted mail is stored. In the

Rows ↓ \ Cols ->	0	1	2	3
0	Byte 0	Byte 1	Byte 2	Byte 3
1	Byte 4	Byte 5	Byte 6	Byte 7
2	Byte 8	Byte 9	Byte 10	Byte 11
3	Byte 12	Byte 13	Byte 14	Byte 15

Figure 2 **Memory cubby model**

computer memory model, each cubby, which has its own unique address, is a byte and the contents can be changed. This changeable aspect of these cubbies is where the name *variable* comes from.

In Fig. 2, the four-by-four array of cubbies have been given addresses based on the row and column that are in. The address is defined by the formula:

```
Address = (Row × 4) + Column
```

Each of the four rows and columns can be addressed using two bits. Two bits have four different states, which is the number of rows and columns in the block of cubbies. This is exactly how all memory is addressed in your PC.

To keep with the computer analogy, I have created the address not using the formula above, but by combining the two row bits as the most-significant two bits of the address and making the columns being the least-significant two bits of the address. These four bits (a nybble) produce an address that is equivalent to the result of this formula. In a computer memory, data can be read from or stored in a memory location at a specific address or using a computed address, similar to this formula.

In most computers, the size of each memory address is one byte (eight bits) and is said to be *eight bits wide*. If the processor is designed to read or write (usually combined into the term *access*) eight bits at a time, then it is described as an *eight-bit processor*.

As shown earlier in the book, the PICmicro® MCU instructions are larger than eight bits, but can only be accessed eight bits at a time by the processor. Even though the instructions are larger than eight bits, the processor is still called an *eight-bit processor*.

The data in the memory locations can be accessed one of two ways. *Direct memory ac-*

Rows ↓ \ Cols ->	0	1	2	3
0	i	j	Byte 2	k
1	"H"	"e"	"l"	"l"
2	"o"	" "	"M"	"y"
3	"k"	"e"	Byte 14	Byte 15

Figure 3 **Variables in cubbies**

cess uses a known address for the variable. This address is calculated when the application code is compiled or assembled and cannot be changed during the program's execution.

The second type of memory addressing is to algorithmically produce the memory address to access during program execution. This allows variables of arbitrary sizes to be created. The general term for data stored this way is *array variables*.

Variables and array variables can be stored in the same memory array. Figure 3 shows how the variables *i*, *j*, and *k* can co-exist with the ASCII string "Hello Myke." This ASCII string is actually an array variable.

In this example, *i*, *j*, and *k* are placed in byte locations at addresses 0, 1, and 3, respectively. The ASCII string is stored in memory locations 0x004 through 0x00C (decimal 12).

The starting address of the ASCII string is known as the array *offset*. To access a specific element in the string (such as the second *e*), the address is generated using the formula:

```
Address = Offset + Element
```

For the second *e*, which is the tenth element of the "Hello Myke" string, the calculation would be:

```
Address = Offset + Element
        = 4 + 9
        = 12
```

Elements in arrays are normally zero based and are pointed to by index registers (which is why the tenth element has the value of *9*). *Zero based* means that the first element in the array is given the element number zero. This makes the address calculation for array addresses much simpler.

The index register is a hardware register that can drive an arbitrary address for accessing an array element. In some processors, the index register can have the array offset separate from the element number (and combine them automatically when the access takes place). In the PICmicro® MCU, you will have to combine the offset and element values manually before they can be loaded into the index register to access an array variable's element.

Along with being character strings, array variables can also be a collection of bytes that can be arbitrarily accessed. For example, you might be recording the weights put on a series of scales (Fig. 4).

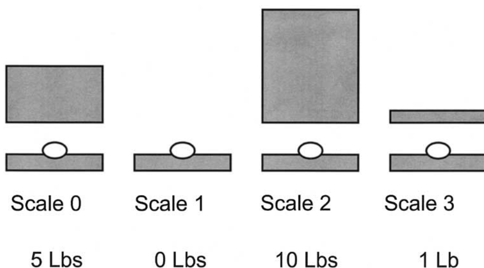

Figure 4 Scale values stored as an array

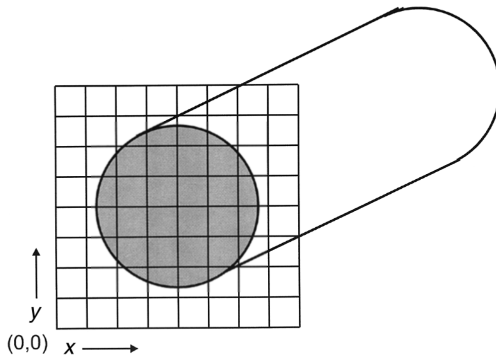

Figure 5 **Two-dimensional array around a pipe**

An array of bytes can be set up in memory with the value stored for each scale. For example, the first array variable (Scale 0) would have *5,* the second (Scale 1) would have *0,* the third (Scale 2) would have *10,* and the last one (Scale 3) would have *1*. Any one of these elements can be accessed using the same code by simply calculating the address using the element number of the scale. Using the array allows an application to access individual data points at random with no time or code penalty regarding which one was being accessed.

Multidimensional array variables can also be used in applications. For example, if you wanted to observe heat transfer from a pipe into the surrounding soil, you could create an array that covers the area you are interested in (Fig. 5).

In this example, the two dimensions (*x* and *y*) can be used to keep track of which array element is being accessed. To calculate the address of the desired element, the same formula as presented previously is used:

```
Address = Offset + Element
```

But, in this case, the element number must be calculated using the *x* and *y* address of the element. When I set up a two-dimensional arrays, I use a single-dimensional array that is broken up into pieces as wide as the array width. Each piece corresponds to one row in the array.

For the pipeline example in which a four-by-four array is used to measure the heat flow (Fig. 5), an eight-by-eight array is shown), the memory array can be represented as shown in Fig. 6.

Figure 6 is really a table that shows the array element for a specific *x* and *y* coordinate. The actual element could be looked up from this table or calculated using the formula:

y	0				1				2				3			
x	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
Element	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Figure 6 **Two-dimensional array memory implementation**

```
Address = Offset + Element
        = Offset + (y × x_width) + x
```

This can be expanded into three or more dimensions. For example, if we wanted to simulate heat transfer along the length of the pipe, we could call the length of the pipe the z dimension, with each unit along the length the location for an x/y array slice. The same single-dimensional array in memory could be used, with each slice at an offset governed by z, multiplied by the size of each slice.

To calculate an element address, in this case, the formula would be modified to:

```
Address = Offset + (z × x_width × y_height) + (y × x_width) + x
```

This method can be extended for as many dimensions as you require. Notice that the memory required will increase geometrically for each dimension that you add.

Creating multidimensional arrays can be difficult and confusing in assembly language. For this reason, I recommend that if more than two dimensions are required for an application array, the code should be written in a high-level language, which will perform the offset calculations for you.

If you have a two-dimensional array and you want to use assembly language for the code, then I recommend that you make the array width equal to a power of two. This will allow you to simply shift the y coordinate to the left the appropriate number of bits (which can be expressed as the logarithm, base 2, of the width).

For an eight-by-eight array, the y coordinate would have to be shifted three (which is the base-two logarithm of eight) bits to multiply it by the width of the array.

This can be expressed as:

```
Address = Offset + Element
        = Offset + (y × width) + x
        = Offset + (y << 3) + x
```

When array elements are accessed, the conventional (C) way of writing them is in the format:

```
Variable[ Index ]
```

Where *Variable* is the start of the array and *Index* is the element being accessed.

Multidimensional arrays access can be represented using the C data format:

```
Variable[ x_Index ][ y_Index ]
```

In the compiler, the *x_Index* and *y_Index* parameters will be combined to form the actual element value.

Arrays must be placed in consecutive memory positions (known as *contiguous memory*) for most processors (including the PICmicro® MCU). If an array is broken up (placed in *noncontiguous memory*), a much more complex algorithm than what has been shown here is needed to access the array variable data. Although this can be done, I recommend that only contiguous memory should be used to avoid any potential problems or unneeded complexities in the code.

So far, when describing array variables, I have only been mentioning the memory word size (a byte for the examples here and the PICmicro® MCU). Multiples of the word size can be combined to produce different data types. In many of the experiments, projects, and tools presented in this book, I will use 16- and 32-bit data types (although 32-bit data types are much less common).

To create these data types, bytes are concatenated together. Thus, to produce a 32-bit variable, four consecutive bytes are used to produce the 32-bit data type.

Figure 7 shows the variables *i* and *k* as eight-bit data types, *j* as a 16-bit variable, and the "Hello Myke" ASCII string array stored in memory to show how the different data types can co-exist in the same memory space.

In most languages, variables are declared before they are used. Declaration statements are used to notify the compiler/assembler that space is needed for the variable and, optionally, the variable is set to an initial value.

Variables that are not declared are known as *dynamic variables* and are normally only available in interpreters, instead of full compilers or assemblers. To implement dynamic variables, the system (which could be the code produced by the compiler for running in a microcontroller must maintain the memory resource, allocating and freeing memory as required by the application.

This capability can take a lot of system resources and is not what you would expect in a small microcontroller like the PICmicro® MCU. In fact, I would recommend staying away from application-development tools that provide this capability because chances are that you will have problems with the software running out of variable memory and crashing or running erroneously because the requested variables could not be made available.

I realize that I have gone on a lot about variables. Much of the information I have presented here is for completeness and will not be required for PICmicro® MCU application programming. I can summarize what you have to know for variables in PICmicro® MCU programming with the three points:

1. Variable memory is stored as an array of bytes.
2. Bytes can be combined to make larger variables or different variable types.
3. Variable arrays are best created using a contiguous portion of the total variable memory.

Assignments are data movements within the computer. Normally, they are written in the familiar equation format:

```
Destination = Source
```

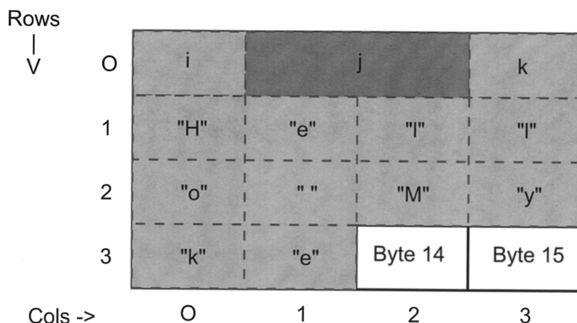

Figure 7 16-bit *j* in two cubbies

Notice that the *Destination* (which can also be referred to as the *result*) is on the left side of the equation. I tend to think of the assignment as:

```
Destination <- Source
```

With the arrow replacing the equals sign to indicate in which direction data is flowing.

The *Source* can be a constant value, a register, a variable, an array element, or a mathematical expression. The *Destination* can only be a register, a variable, or an array element. It should be obvious that the *Destination* cannot be a constant.

Previously, I described a variable and an array element. A *register* is a byte location that is available to the processor to pass hardware information back and forth. An example of a register would be an I/O port. By writing to the port, data is output to peripheral devices. Reading from the I/O port, the state of a peripheral device is returned.

The PICmicro® MCU has no explicit memory locations. instead registers and memory are in the same address spaces. To differentiate between them, memory bytes are known as *file registers*. This can be confusing for new users. If you have read the previous descriptions of the PICmicro® MCU architecture, you could be wondering where memory is located in the PICmicro® MCU.

With the concepts explained so far, a computer is only able to load from and save data to memory and registers. To make the device more useful, it must be able to carry out some kind of mathematical operations.

As I was writing this, I was reminded of Arthur C. Clarke's short story *Into the Comet* in which the computer on a deep-space mission loses the ability to do mathematical calculations (but can still retrieve information accurately). Using this capability, the crew is able to calculate a return orbit back to Earth using abacuses.

Hopefully, you realize that this is impossible because for a computer to retrieve information and assign it as output, it must be able to calculate memory addresses from which to access it from. This calculator, known as the *Arithmetic Logic Unit (ALU)* of a computer, not only provides the ability to calculate addresses in the computer's memory, but also to arithmetically process data as part of the application's execution.

When creating computer applications, you can assume that the computer can do basic arithmetic operations on numeric data. The basic operations are shown in Tables 6 and 7 might also be available, but in small processors (like most of the PICmicro® MCUs) these functions usually have to be created using a series of programmed functions.

The book explains how the multiplication, division, and *modulus* (which returns the "remainder" from a division operation) operations can be developed for the PICmicro® MCU.

Normally, these functions only work on integers (whole positive and negative numbers) and not real numbers. Some processors (generally PCs, workstations, and servers) can

TABLE 6 Basic Integer Mathematical Operations used in Programming

SYMBOL	FUNCTION
+	Addition
−	Subtraction/Negation

TABLE 7 Advanced Integer Mathematical Operations used in Programming

SYMBOL	FUNCTION
*	Multiplication
/	Division
%	Modulus

work with these numbers "natively," but most (including the PICmicro® MCU) must create software routines to handle them. These routines are normally provided as libraries to the executing applications.

Along with the basic arithmetic operations, logarithmic and trigonometric operations are often available in high-level languages and some processors (Table 8).

For these operations, real numbers are normally required. In some computers (such as the Parallax Basic Stamp), the trigonometric functions are provided for a circle radius 127. The trigonometric values returned are then in the range of 0 to 127, instead of 0 to 1, as you are probably most familiar with. For the circle of radius 127, notice that two's complement negative values can be returned for negative sine and cosine values.

Along with arithmetic functions, processors also include "bitwise" operations to allow the processor to manipulate individual data bits. These operations (Table 9) are always built into computer processors.

The last type of operators are known as *logical operators* and is used to compare values and execute according to the results. Logical operators only work on single *true* or *false* values. These *true*/*false* values are used to determine whether or not a change in conditional execution is made. The typical logical operations, using C formatting for the first six, are shown in Table 10.

TABLE 8 Floating Point Mathematical Operations used in Programming

ABS	- Return the Absolute (Positive) value of a number
INT	- Return the data that is greater than or equal to One
FRAC	- Return only the data that is less than Zero
SIN	- Return the trigonometric "Sine" of an angle
COS	- Return the trigonometric "Cosine" of an angle
ARCSIN	- Return the trigonometric "Sine" of a value
ARCCOS	- Return the trigonometric "Cosine" of a value
EXP	- Return the value of an exponent to some base (usually 2)
LOG	- Return the value of a Logarithm to some base (usually 2)

TABLE 9 Bit Operations used in Programming

Operator	Description
&	- AND two numbers together (return a “1” for a bit in a byte if the corresponding bits of both inputs are set to “1”)
|	- OR two numbers together (return a “1” for a bit in a byte if one or both of the corresponding bits in the inputs are set to “1”)
^	- XOR two numbers together (return a “1” for a bit in a byte in only one bit corresponding bit is set to “1” in the two inputs)
!	- NOT the byte (“complement” or “invert” each bit. This operation is the same as XORing the byte with 0x0FF to change the state of each bit to its complement)
<<	- Shift the byte one bit to the left
>>	- Shift the byte one bit to the right

The results of these logical operations are normally not zero (*1* or −*1*) for *true* and *0* for *false*. Applying a numeric value to the result allows them to be used like arithmetic or bit values.

When using arithmetic functions, the assignment format is used with the source now referred to as an *Expression*. Expressions consist of the arithmetic and bitwise operations listed previously.

TABLE 10 Logical Operations used in Programming

Operator	Description
==	- Compare two values for being equal and return “true” if they are
!=	- Compare two values for being not equal and return “true” if they are not equal to one another
>	- Compare two values for the first (leftmost) being greater than the other and return “true” if it is
>=	- Compare two values for the first (leftmost) being greater than or equal to the other and return “true” if it is
<	- Compare two values for the first (leftmost) being less than the other and return “true” if it is
<=	- Compare two values for the first (leftmost) being less than or equal to the other and return “true” if it is
NOT (“!”)	- Complement the Logical value (if it is “true” then make it “false” and visa-versa)
AND (“&&”)	- Return “true” if two logical values are both “true”
OR (“||”)	- Return “true” if either one of two logical values is “true”
XOR	- Return “true” if only one of two logical values is “true”

For example, the assignment with expression (usually known as an *assignment statement*) takes the form:

```
A = B + C
```

Where variable *A* is assigned the sum of *B* and *C*.

Assignment statements can become quite complex. For example, going back to the two-dimensional pipe heat-transfer example, to calculate a new value for one of the array elements, the average of the surrounding elements is calculated and assigned to it. This could be calculated using the assignment statement:

```
element(x, y) = (element(x - 1, y) + element(x - 1, y - 1) +
 element(x, y - 1) + element(x + 1, y - 1) + element(x + 1, y) +
 element(x + 1, y + 1) + element(x, y + 1) + element(x - 1, y + 1)) / 8
```

In this example, the summation of the eight surrounding elements takes place before the divide by eight to get the average. The summation is enclosed by brackets (also known as *parenthesis*) to indicate to the compiler/assembler that it has to be computed before the division can occur.

If the brackets are left out:

```
element(x, y) = element(x - 1, y) + element(x - 1, y - 1) +
 element(x, y - 1) + element(x + 1, y - 1) + element(x + 1, y) +
 element(x + 1, y + 1) + element(x, y + 1) + element(x - 1, y + 1) / 8
```

The computer will evaluate the expression according to its internal order of operations. In most cases, multiplication and division are at a higher priority over addition and subtraction.

Thus, in the example, *element(x, y)* will be loaded with the sum of the seven surrounding elements plus one eighth of the last element.

To avoid problems like this, I always surround what I want to be the highest-priority operations with parenthesis to explicitly force the compiler/assembler to evaluate the portions of the expression in the lowest level parenthesis first. I find that doing this also makes the statements easier to read (either by others or by yourself when you are looking at the code several years later).

Following what I've presented so far, you have enough information to understand how to program a processor that has memory that can be read from and written to, as well as the contents arithmetically modified. Except for one important piece, this is all there is to the programming basics.

The missing piece is the ability to change the execution path of the application code. This can be based on the status of some parameter or done unconditionally. In traditional high-level programming languages, conditional changes are accomplished by the *if* statement. I use this convention when explaining how execution change is implemented in programming.

For nonconditional changes in execution, the *goto Label* statement is used. This statement is used to load the processor's program counter with a new address and start reading instructions from there.

For example, a "loop" can be created in an application by executing a *goto label* statement to a location in the code that is "above" the *goto label*:

```
  - Loop Initialization code
LoopLabel:                    // "Loop" back Here
  - Instructions executed inside the Loop
  goto LoopLabel              // Execute Instructions inside Loop Again
```

The *LoopLabel* is a text string (known as a *label*) that is used as a destination for the *goto* statement. The colon (*:*) at the end of the label is a common indicator to compilers and assemblers that the text string is a label.

Although a physical address can be explicitly stated in virtually all programming environments, I highly recommend that you let the compiler or assembler generate the addresses for you unless a specific hardware address is required. The only examples in the PICmicro® MCU where addresses have to be explicitly specified are the reset address, the interrupt vector, and the page boundaries. All these addresses are explained elsewhere in the book as to how they relate to the PICmicro® MCU.

The *goto* statement by itself isn't that useful (except for creating endless loops, as the one shown) unless it is coupled with a conditional execution statement. *Conditional execution* is code that is only executed if specific parameters are met. The *if* statement is normally used with the *goto* to provide "intelligence" to an application.

The conventional format of the *if* statement is:

```
If (Condition) then goto Label
```

where *Condition* is a logical mathematical operation that tests parameters to meet a specific condition.

For example, you can test two variables. If they are equal, then execution changes to a new location.

```
  If (A == B) then goto NewLocation
  - Instructions Executed if A is not equal to B
NewLocation:
```

In this example code, if *A* does not equal *B*, then the *Instructions Executed if A is not equal B* are executed. Otherwise, when *A* does equal *B*, then instructions are skipped over.

The *if (Condition) then goto Label* conditional execution statement is the primary method used in all conditional execution statements in all general-purpose high-level languages. This might seem hard to believe, and if you have any experience with structured programming, this doesn't seem true. The next two sections show how the *if (Condition) then goto Label* statement is used as the basis for structured conditional statements, assembly-language conditional execution, and table jumps.

Understanding how to use the *if (Condition) then goto Label* statement will be one of the hardest and most difficult things that you will have to master as you learn how to program. This concept will take a bit of work to see it in action in PICmicro® MCU assembly-language code, but as you work through applications and start to develop your own, seeing them will become second nature.

Although the first four concepts are basic to programming and I urge you to keep them

clear in you mind, two others will make programming easier and help you to understand how the PICmicro® MCU is operating.

Macros and subroutines are useful programming techniques. Both are designed to do the same thing, eliminate the need for you to type in redundant code. Instead of repeatedly entering in the same code, a subroutine or macro is created and then called or invoked, respectively, each time access to it is required. From your point of view, both are very similar, although there are some differences that you should be comfortable with.

When a subroutine's *call* statement is encountered, the location *after* the *call* statement is saved (usually on the program counter stack) and execution jumps to the start of the subroutine. When the subroutine is finished, the *return* statement/instruction is executed and execution returns to the saved address (the first instruction after the call). This is shown in Fig. 8.

When *Subroutine* is called, the main execution registers (known as the *Context Registers*) might be saved before the subroutine's code executes (but after the *call Subroutine* instruction is executed). In this case, after the subroutine has finished executing, but before the *Return* statement is about to return execution to the caller, the context registers are be restored. When the program resumes execution after the subroutine, the registers will be in the same state as if the subroutine wasn't executed at all. This context saving is not often required in simple processors like the PICmicro® MCU, but for complex processors and applications, it usually is.

Data passed to and from a subroutine from the calling code are known as *parameters*. These parameters can be used to specify data for the subroutine to process, or information needed by the caller. It is important that parameters are supplied at *run time,* which is in contrast to macros, which execute the parameters at *compile/assembly time*. This is an important point and one that often trips up new programmers. The "Assembly-Language Programming" section of this chapter describes parameters in more detail.

Macros are code that are inserted directly into the calling (or, in this case, "invoking") code, replacing the macro statement and inserting the executable code from within the macro (Fig. 9).

Inserting the code directly into the invoking code, eliminates the need for the *call* and *return* statements, which allows the code to execute slightly faster. As indicated, macros are customized for the application during compile/assembly time, instead of at run time. In

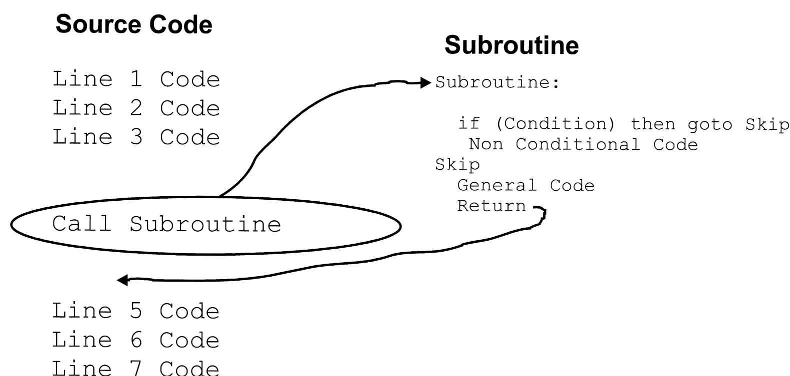

Figure 8 **Subroutine operation**

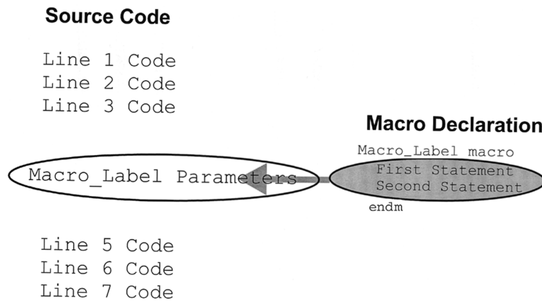

Figure 9 Macro invocation operation

many ways, this makes macros much more flexible and better able to add considerable functionality to your applications.

Directives are the last concept presented in this section. Directives are instructions passed from the software to the compiler or assembler to instruct how the operation is to take place. The parameters passed via directives are the kind of processor model it should create instructions for, the device configuration information and information used to customize the application and macros.

When you first start programming (or first start programming a new device or using a new language), directives will seem like "the straw that broke the camel's back." You end up ignoring them and hoping the application will work without understanding them. Although I can relate to this feeling (and have been there myself), when starting with a new language, I urge you to take a look at the directives to at least have an idea of what kind of operations and functions are made possible by them.

High-Level Languages and Structured Programming

A *high-level language* is a specification for a series of natural language-like statements, which are used to develop a computer program. As I go through assembly-language programming, you'll discover that many of the details of the concepts presented in the previous section are obscured by the individual processor instructions and the methods in which they are used to carry out specific tasks. In high-level languages, these concepts are very obvious and are easily seen built into the source code.

This section is really an extension to the previous one. It covers the four basic concepts as how they relate to actual computer programs. Although this information is more specific than that given in the previous section, it is still pretty general. Actual high-level languages will implement the concepts presented here in different ways according to the language and the implementation.

The basis for most high-level languages is the *statement*. A statement is usually in the conventional formats shown in Table 11.

These statements are available in virtually all programming languages. Although the syntax (format) of the different languages changes the formats slightly, these statement types are available for your use.

A constant theme in most of the statements is the *expression.* An expression is another

TABLE 11 List of Basic High Level Language Statements

destination = expression	// Assignment statement
goto Label	// Execution change statement
if (expression) then goto Label	// Conditional execution change // statement
call subroutine (expression, . . .)	// Subroutine call statement

term for the *arithmetic operation* presented in the previous section. An expression combines different functions together to produce a mathematical result.

For example, the expression:

```
A + (B × C)
```

will add *A* to the product of *B* and *C*.

In the assignment statement, the expression is used to create the value to be stored into the destination. So, for the assignment statement:

```
A = 32
```

The variable *A* is loaded with the decimal value *32*. Although this is quite simple (and obvious), the expression can become very complex and optionally use array variables as well:

```
A[ 47 ] = B + (C[ 32 ] * 67)
```

As I go through high-level languages, you are going to discover that many of the concepts presented are *recursive,* meaning that they can be used within themselves. A good example of this is for array variables, in which the index (selected element) is an expression itself.

Instead of hard coding the array indices, as shown in the previous assignment statement, they can be arithmetically derived as:

```
A[ 37 + D ] = B + (C[ 22 + D ] * 67)
```

Most operators in an expression have two parameters (inputs). For example, the addition operator requires two and is defined as:

```
sub_expression + sub_expression
```

where *sub_expression* can be a constant, variable, array variable, or another expression (this is where the recursive property of programming comes in).

There are also single parameter (expression) operators, with the most common being − (negate) and *!* (bitwise invert) and return a single value. *Functions,* which return a value, can also require a single input parameter.

You are probably most familiar with the minus symbol ("−") as a two-parameter subtraction operator, but when used with only one parameter, it returns the negative of its parameter to the expression. This means the unlikely expression:

```
A + -B
```

is actually valid, It is literally adding the negative of *B* to *A*. This statement can be simplified to:

```
A - B
```

Negation and subtraction have some special problems as they relate to the PICmicro® MCU architecture that is carefully covered in this book.

Functions are subroutine calls that return a single value. Many high-level languages have basic functions built into them, which use the format:

```
Return_Value Function_Name(Input_Parameter)
```

Creating and using functions are explained later in this section.

As I go through programming in more detail and specifics, I should explain how programming constructs are defined. Normally, constructs are specific programming objects specific to the language that are defined using formulas in the format shown in Table 12.

Using this convention, a simple arithmetic "expression" can be defined as:

```
expression := {-|!} Constant | Variable | Array_Variable[ expression ]
                 { + | - | * | / | % expression }
```

Notice that within this *expression* definition, *expression* itself is part of the definition. This is because the expression can be expanded using two input-parameter operations (+, −, *, /, and % in this example) or using an array variable, which has an expression to define its index.

I use this format throughout this book to explain how operations work.

The *goto Label* statement carries out a nonconditional execution change. When it is used with the conditional execution change statement, *if (expression) then goto Label,* a variety of "structured programming" can be derived, which will make application programming much easier.

TABLE 12 High Level Language Statement Construct Definition

Construct_Type := Label (Required Parameters {Optional_Operators | . . . }	
Where:	
Construct_Type	- The programming object (or "construct") being defined.
:=	- Equate Symbol
Parameter/Label/Operators	- Inputs/Labels/Variables/Constants used in building the construct
{ }	- Parenthesis to indicate optional parameters
|	- Indicate that one or another Parameter is used
. . .	- Indicate "And So On"

In high-level languages, the expression in the *if* statement is evaluated, but *not* saved. The evaluated expression is used to determine whether or not the *goto Label* portion of the statement is to be executed or skipped over. In most languages, if the expression is evaluated to zero, then the *goto Label* is skipped. The expression evaluated to zero case can be called *false* in many languages. If the expression is not zero (true), then the *goto Label* portion of the statement is executed.

Thus, arithmetic only expressions can be used along with comparisons in most high-level languages. For example:

```
if ((A + B) != 0) then goto Label
```

could be simplified to:

```
if (A + B) then goto Label
```

I am of a mixed mind whether or not to recommend programming like this. Although it is quite efficient, it can be somewhat confusing for new programmers to understand. In this book, I will always have a comparison in an *if* statement wherever possible.

Be sure that you use the correct comparison operator in your expression. In C, the comparison operator is == (two equals signs) instead of the expected one (which is an assignment statement). One of the biggest mistakes made by C programmers is that they write the comparison statement:

```
if (A = B)
```

when the proper form is:

```
if (A == B)
```

In the *if (A = B)* case, the variable *A* is assigned the value of *B* and executes the conditional code if the contents of *B* are not equal to zero. This common mistake is very difficult for new programmers to find.

To minimize the opportunity of this error, many C programmers will only put the constant values of a comparison on the left side of the comparison operator. So, instead of:

```
if (A == 7)
```

the *if* statement is written as:

```
if (7 == A)
```

If the second equals sign (=) is forgotten, then the compiler will return an error stating that the language can't assign a value to a constant. This error is detected before the application begins executing, which makes it very simple to find and fix.

Structured programming is the term for conditional code that is based on, but doesn't use the *if (expression) then goto Label* and *goto Label* concepts. If you haven't been exposed to structured programming before, the idea of "gotoless" (to coin a word) programming might seem quite radical and difficult to work with. But it actually makes application development easier and less error prone.

I have indicated that the structured programming constructions are based on the *if (expression) then goto Label* and *goto Label* constructs. As I go through the different structured programming constructs, I show how these two basic constructs are used to produce them. To make this description clearer, I will refer to the expression evaluated for the structured programming constructs to determine how to execute as the "condition."

The most basic structured programming construct is the *if/else/endif,* which is used to conditionally execute two different paths based on a set of conditions. The *if/else/endif* construct is written as:

```
if (condition)
  -Code to Execute if "condition" is "true"
else
  -Code to Execute if "condition" is "false"
endif
```

In this construct, if the condition is *true* then the code following the *if* statement is executed to the *else* statement and the jumps to after the *endif* statement. If the condition is *false,* the execution jumps to the instruction after the *else* statement and resumes execution from there.

The *if/else/endif* construct is created using the *if (expression) then goto Label* and *goto Label* basic constructs as:

```
  if NOT(condition) then goto ifelseLabel       // "if" Statement
    -Code to execute if "condition" is "true"
  goto ifendLabel                               // "else" Statement
ifelseLabel:                                    // "else" Statement
    -Code to execute if "condition" is "false"
ifendLabel:                                     // "endif" Statement
```

For this function, I have used the logical NOT function to complement (invert) the results of the evaluation of the *condition* expression. The logical NOT returns *false* when the input parameter (*condition,* in this case) is *true* and visa-versa. Thus, the *goto Label* at the end of the statement only executes if the condition is false.

Also notice that the *else* statement is really just a label and a *goto Label* statement and *endif* is just a label.

This simple implementation is true for most structured programming. Although structured programming simplifies the application from the programmer's point of view, it does not significantly increase the complexity of the code used in the application. This is important for the PICmicro® MCU that has limited execution and variable register space.

The *else* statement is normally optional in the *if/else/endif* construct, allowing code to be inserted that is only executed if the condition is *true:*

```
if (condition)
  - Code to Execute if "condition" is "true"
endif
```

the *if (condition)* then *goto Label/goto Label* analog is:

```
  if NOT(condition) then goto ifendLabel        // "if" Statement
    - Code to execute if "condition" is "true"
ifendLabel:                                     // "endif" Statement
```

In most languages, you can put Null statements after the *if* or *else*. For example, if you had code that only executed if a condition was not true, then the following code could be written:

```
if (condition)
else
  - Code to Execute if "condition" is "false"
endif
```

I would recommend that you never use this code format in your application for two reasons. First, the code is confusing to read and begs the question of where is the line is that appears to be missing between "if" and "else". This code format uses what I call "negative logic" and I recommend that you avoid using it in your application code because it makes the application more difficult to read and understand.

The second reason to avoid using a null statement followed by an *else* in a *if* conditional statement is that the compiler might insert extra instructions (which are unneeded to implement it. The *if (condition) then goto Label/goto Label* model you expect to see is:

```
  if (condition) then goto ifendLabel           // "if" Statement
    - Code to execute if "condition" is "false"
ifendLabel:                                     // "endif" Statement
```

but, chances are you will get:

```
  if NOT(condition) then goto ifelseLabel       // "if" Statement
  goto ifendLabel                               // "else" Statement
ifelseLabel:                                    // "else" Statement
    - Code to execute if "condition" is "false"
ifendLabel:                                     // "endif" Statement
```

which takes up a few extra instructions of program memory and a few extra cycles to execute. This will lower the performance of your application in terms of total speed and size.

The *if*/*else*/*endif* constructs, like other structured programming constructs, can be nested within other constructs. *Nesting* means that conditional statements are placed inside of other conditional statements. For example:

```
if (condition_1)
  if (condition_2)
    - Code to Execute if both conditions are true
  else
    - Code Executes if "condition_1" is true/"condition_2" is false
  endif
else
    - Code to Execute if "condition_2" is false
endif
```

Notice that I have indented the nested statements in this example. Indenting nested code helps to make the different nesting levels more obvious and easier to follow through.

The *if/else/endif* construct will allow conditional execution of an application, but it cannot loop or return to the top of the application. To perform this function, the *while/wend* and *do/until* structured programming constructs are normally used. The *while/wend* constructs look like:

```
while (condition)

  - Execute while “condition” is true

wend
```

This construct will repeat the code within it until the condition is no longer true. Many applications use a flag for the *while* condition and change its state when it is time to exit the loop. The *if* (*condition*) *then goto Label/goto Label* form of this construct is:

```
whileloopLabel:                                       // “while” Statement
  if NOT(Condition) then goto whileendLabel           // “while” Statement

    - Execute while “condition” is true

  goto whileloopLabel                                 // “wend” Statement
whileendLabel:                                        // “wend” Statement
```

In most microcontroller applications, execution has to continually loop around to execute the application. In this case, the condition used in the *while* statement can never become false and the application loops forever.

This is known as an *endless loop,* which can be implemented with the *while/wend* constructs as:

```
while (1 == 1)

  - Execute application code forever

wend
```

Ideally, the compiler that processes this source code will recognize that this is an endless loop and generate the code:

```
whileloopLabel:                                       // “while” Statement

    - Execute while “condition” is true

  goto whileloopLabel                                 // “wend” Statement
```

which saves a few instructions and cycles over continually testing the case that one is equal to itself.

For the endless *while* loop, I use the condition that the code should execute while the basic truth that unity is always equal to itself is true. If this ever changes, then I think execution should stop. Other programmers will declare a constant zero using the label *Doomsday* and then loop *while NOT(Doomsday)* to be funny. Of course, you can always just put a *1* as the *while* condition and loop exactly the same way as these two examples.

Along with the *while/wend* looping constructs is the *do/until*. This construct is the inverse of the *while/wend* and only exits the loop when the condition becomes true.

```
do
  - Execute until "condition" is true
until (Condition is True)
```

This can be modeled as:

```
doLabel:                                         // "do" Statement
    - Execute until "condition" is true
  if NOT(condition is True) then goto doLabel    // "until" Statement
```

The advantage of this structured programming construct is that a test of the condition is not done before the first execution of the loop code. Personally, I find this construct confusing and difficult to think through because it requires logic that is negative to the *if* and *while* constructs that execute based on a condition being true. As indicated, I always try to avoid negative logic because it seems to take away from the readability of the application source code.

Along with *while*/*wend* and *do*/*until,* the *for*/*next* structured programming construct can be used to execute some code for a set number of times. The format of the *for*/*next* constructs in code is:

```
for Variable = Initial to Final {step = stepvalue}
  - Code to execute while "Variable" does not equal "Final"
next
```

Variable is a counter that is set to the value of the *Initial* expression and then incremented each time through the loop until it equals the *Final* expression. The *for*/*next* construct can be represented as:

```
  Variable = Initial                             // "for" Statement
  ifdef step                                     // "for" Statement
    StepValue = stepvalue                        // "for" Statement
  else                                           // "for" Statement
    StepValue = 1                                // "for" Statement
  endif                                          // "for" Statement
forloopLabel:                                    // "for" Statement
  if (Variable == Final) then goto forendLabel   // "for" Statement

    - Code to execute while "Variable" does not equal "Final"

  Variable = Variable + StepValue                // "Next" Statement
  goto forloopLabel                              // "Next" Statement
forendLabel:                                     // "Next" Statement
```

In this example, you will see the statement *ifdef step,* which I haven't explained yet (and will explain in greater detail in the book). Quickly, the *ifdef step* statement is a directive that is processed during the compilation (and assembly) step and tests whether or not the *step* parameter is present. If the parameter is present, then the *stepvalue* specified will be added to *Variable* in each loop. If the *step* parameter is not present, then one will simply be added to *Variable* in each loop.

Normally to execute a set number of times, the initial value is set to one and the final

value is set to the total number of times through the loop. For example, to execute the code within the *for/next* statements five times:

```
for i = 1 to 5
  - Code to execute five times
next
```

The *for* loop is very straightforward, although there is some opportunity for problems. First is the value specified for the *step* increment. Take care to avoid the opportunity to specify a value that will result in the loop will never being ended. An example of this is:

```
for i = 1 to 4 step 2
  - Code to execute twice
next
```

In the *for/next* loop, *i* will be 1, 3, 5, and so on, but never 2. This might seem obvious, but I think everybody gets caught with this one, one time or another (especially if variables are used for the initial and final counter values).

I do not recommend modifying the *for* counter from inside the loop. Although this is possible in most high-level languages, writing to the counter could cause it to be given an invalid value. Like the previous point to watch out for, you could end up in an endless loop or exiting the loop before you want to.

The last structured language construct to describe is the *switch*. This construct is designed for situations where an expression is going to be checked for multiple values. Probably the most obvious way to implement this is as a series of *if* statements like:

```
if (expression == Case1)
  - Execute if expression is equal to “Case1”
else if (expression == Case2)
  - Execute if expression is equal to “Case2”
else if (expression == Case3)
  :
else
  - Execute if expression doesn’t equal any of the cases
endif
endif
  :
endif
```

This method is useful for certain situations, but is somewhat clumsy (notice that the number of *endif*s has to match the number of *if*s in this code). As well, the expression has to be evaluated repeated for each *if* case.

In terms of source code formatting, notice that for this application, the subsequent *if* statements are not indented relative to the previous ones. Instead, in this form it is assumed

that the same expression is being repeated, which is best displayed as a repeating condition that does not have its indentations changed.

A *repeating condition* is a good way of describing the *switch* statement, which has the form:

```
switch( expression )
  Case1:

    - Execute if expression == Case1

  Case2:

    - Execute if expression == Case2

  Case3:

    :

  else

    - Execute if expression does not equal any of the cases

endswitch
```

This construct can be produced using the *if (condition) then goto Label* and *goto Label* constructs as:

```
  Temp = expression                                    // "Switch" Statement
  If (Temp == Case1) then goto switchCase1Label   // "Case1" Statement
  If (Temp == Case2) then goto switchCase2Label   // "Case2" Statement
  If (Temp == Case3) then goto switchCase3Label   // "Case3" Statement
   :
  goto switchelseLabel                                 // "else" Statement

switchCase1Label:                                      // "Case1" Statement

     - Execute if expression == Case 1

  goto switchendLabel

switchCase2Label:                                      // "Case2" Statement

     - Execute if expression == Case 2

  goto switchendLabel

switchCase2Label:                                      // "Case3" Statement

  :

switchelseLabel:                                       // "else" Statement

     - Execute if expression does not equal any of the cases

switchendLabel:
```

If the cases are in sequential order, then the code could be further enhanced by jumping from an address table. An *address table* is a variable array that contains addresses that execution can jump to or call instead of retesting each condition. For this example, the code could be simplified (assuming that the cases are in sequential order) to:

```
  Table[ 0 ] = switchCase1Label                        // "Case1" Statement
  Table[ 1 ] = switchCase2Label                        // "Case2" Statement
  Table[ 2 ] = switchCase3Label                        // "Case3" Statement
```

```
   :
  Temp = expression                              // "Switch" Statement
  if ((Temp < Case1) OR (Temp > Case#)) then goto switchelseLabel
  goto Temp[ Temp - Case1 ]                      // "Switch" Statement
switchCase1Label:                                // "Case1" Statement
     - Execute if expression == Case 1
  goto switchendLabel
switchCase2Label:                                // "Case2" Statement
     - Execute if expression == Case 2
  goto switchendLabel
switchCase2Label:                                // "Case3" Statement

  :

switchelseLabel:                                 // "else" Statement
     - Execute if expression does not equal any of the cases
switchendLabel:
```

Using the table jumps allows the code to be implicitly tested as part of finding the address. The advantages of using the table of addresses are that the code required for the switch table is vastly reduced. The time to access the code specific to different cases is the same for all the different cases, no matter how large the number of cases.

The ability to jump to addresses from a table is an important feature in the PICmicro® MCU and one that I will be explaining to you and exploiting in the experiments, projects, and tools presented in this book.

In the switch examples, I have shown leaving the case conditionally executing code as happening automatically when the next *case* statement or the *else* statement is encountered. This is not always true. In some languages (C, in particular) an explicit *break* statement is used to jump out of the *switch* statement's *case*. The programmer-defined break gives additional control over how the *switch* statement executes.

The *break* statement is also often made available for the other structured constructs listed. For example, the break statement could be used for exiting a *while* loop without executing the *while* condition test statement.

Personally, I do not like using *break* for anything other than the *switch* statement because it is (forgive me for using this term) an "unstructured" approach to program execution control. I prefer writing applications that only exit through the standard condition test statements, instead of forcing the code to execute in a specific manner.

Most modern high-level languages are procedural languages that have a method of declaring subroutines and functions that causes them to be isolated from the mainline and each other. The biggest issue with regard to procedural languages is how to handle the variables and parameters passed to them.

In a typical assembly-language program, variables are accessible from anywhere in the code. Subroutines can have parameters passed to them or they can access them from common variables. This aspect of assembly-language programming can make the subroutines hard to read and understand exactly how they work. Also, they can be difficult to reuse in different applications.

I state this elsewhere in the book, but the goal of successful application software devel-

opment is to create routines that can be used in other applications. To do this, prior planning of the design of the routine must be made to avoid having application-specific information hard coded into it.

The easiest way to create routines that can be used in other applications is to avoid accessing global variables from within them. Instead, following the rules for procedural languages presented here, will allow you to write high-level language and assembly-language routines that can be transferred to other applications without modification.

Procedural routines are defined like:

```
subroutine Label( type parameter { , ... } )
  - Subroutine code
endsub
```

where the calling statement passes a number of parameters to the routine:

```
call Label( parameter { , ... } )
```

The parameters are passed to the subroutine by copying in their values into temporary variables. These variables can be read from and written to in the subroutine without affecting the original variable values. A few points on this follow.

The first is on the different type of variables. So far in my description of software development, I have only noted one category of variable, the types that are accessible throughout the application software. Procedural high-level languages have two categories. First is the *global variable*, which is the standard variable that is accessible throughout the application. The second type is the *local* or *automatic variable* that is only available within a subroutine. Memory for the automatic variables are defined when the subroutine starts executing and is freed up when the subroutine's execution has completed.

The memory used for automatic variables is usually taken from the application's data stack or heap. Both of these memory resources allow applications to quickly request memory for temporary purposes and then return them when the application is done with them.

To design your routines to be as "portable" to other applications as possible, they should have minimal access to global variables. Instead, any application specific information should be passed to the routine via the input parameters. Passing all the required information to a subroutine as input parameters does use up more memory than using global variables. In fact, recursive subroutines can use an alarming amount of memory for creating the automatic variables as execution moves through the called routines (this is another reason to avoid recursion in the PICmicro® MCU).

Along with the passed parameters, variables specific to the routine can be declared within the routine. These are automatic variables as well. The memory used by them will automatically be returned to the data stack or heap when the routine is finished.

Sometimes when routines are called, the programmer wants data to be returned from the calling routine. Two methods are normally used.

The first is the function type of subroutine. In this type of subroutine, a data value is passed back to the caller. The routine takes the form:

```
type function Label[ type parameter{ , ... } ]
  - Function code
  Label = expression              // Save Return Value
endfunc
```

With the calling statement being part of an expression like:

```
A + (Label[ parameter { , ... } ] * 47)
```

Functions can be used to enhance a high-level language compiler. For example, if you wanted to have a factorial function (which returns the factorial of a number), you could write it as:

```
int function Factorial( int Number )
  if (Number > 1)
    Factorial = Number * Factorial(Number – 1)
  else
    Factorial = 1
endfunc
```

This recursive function will return the factorial of a given integer as the product of the number times the factorial of the number less one. Using a function for the routine makes it very easy to develop.

The second method of returning parameters from a routine is to pass a *pointer* to the data to be modified. If you have taken some programming courses before, you are probably groaning at the thought of pointers. You probably remember them as being difficult to work with, but they are necessary in some situations and they can make your application development easier. The next few paragraphs provide everything I think you need to know about pointers for microcontroller programming.

First, pointers are type defined and are used to provide a direct pointer into data memory of an application. Where I defined the accessing an array variable element as:

```
Address = Offset + Element
```

When using a pointer, the *Address* value calculated can be loaded in as the pointer's address and the array accessed from there. Data pointed to can be very easily accessed as a single-dimensional array of bytes in C. For example, if you wanted to find the third byte in a string and print it out, the following C code is required:

```
char far * StringPtr;
char String[ 45 ] = "So, So you think you can tell...";
  StringPtr = &String[ 0 ];
  printf( "The Third Character of \"String\" is: \"%c\"\n",
    StringPtr[ 2 ]);
```

The first line defines a pointer named *StringPtr* to type *char*. The next line creates a pointer (*String*) to an single-dimensional array 45 bytes long that has been initialized to a line from an old song. In the third line, the pointer to the initialized array is passed to *StringPtr*. The last line is used to output the third character of the initialized array (the comma *,*).

In C, the & character is used to define the physical location of a variable in memory. The * character is used to define, return, or set the current value of a pointer.

In this code, a pointer is used both to point to data as well as provide a label into an array element. After seeing this code, you're probably wondering what is the difference between a string, an array or characters, and a pointer to a string. There really is no difference.

This is important because it means that you can store data from a file into a buffer, point to the buffer, and index into the buffer using the same pointer that was set up when the buffer was initially created and the memory for the buffer was allocated.

Being able to understand and recreate the code above is all I think anybody should be capable of when they develop a C application. This level of knowledge will allow you to access buffers (as single-dimensional arrays), as well as keep track of where the buffers are in memory.

I have not presented a very comprehensive tutorial on pointers because they are not really appropriate to use with the PICmicro® MCU. Pointers are best used in applications and systems that can access large amounts of memory and require a large amount of data to process. As well, pointers are also best suited to processors that can have a large amount of physical memory. The mid-range PICmicro® MCUs can have up to file register 392 bytes that are not stored in a contiguous fashion, which really eliminates much of the need for pointers.

THE BASIC LANGUAGE

The *BASIC (Beginner's All-purpose Symbolic Instruction Code)* language has been used by new computer programmers for more than 30 years. It really became famous when it was included with the Apple][computer in the mid-1970s. There are a number of BASICs are available for the PICmicro® MCU, one of which is described in detail in the next section.

In its original form, the BASIC language is somewhat unstructured, although Microsoft has done a lot to enhance the language and make it easier to work with. This section introduces the BASIC language, with Microsoft's extensions. The next two sections introduce PicBasic (which is provided by meLabs), as well as Visual Basic, Microsoft's outstanding tool for creating Windows applications.

BASIC variables do not have to be declared, except in specialized cases. The variable name itself follows normal conventions of a letter or _ character as the first character, followed by alphanumeric characters and _ for variable names. Variable (and address label) names might be case sensitive, depending on the version.

TABLE 13 Basic Language Data Type Specification

SUFFIX	FUNCTION
$	String Data
%	Integer
&	Long Integer (32 Bits) - Microsoft BASIC Extension
!	Single Precision (32 Bits) - Microsoft BASIC Extension
#	Double Precision (64 Bits) - Microsoft BASIC Extension

To specify data types, a suffix character is added to the end of the variable name (Table 13).

In Microsoft BASICs, the *DIM* statement can be used to specify a variable type:

```
DIM Variable AS INTEGER
```

without using the suffixes.

To declare arrays, the DIM statement is used like:

```
DIM Variable( [Low TO] High[, [Low TO] High...] ) [AS Type]
```

A number of built-in statements are used to provide specific language functions to applications (Table 14).

TABLE 14 Basic Language Built In Function Statements

STATEMENT	FUNCTION
BASE	Starting Array Element
DATA	Data Block Header
DIM	Dimension Array Declaration
OPTION	Starting Array Element
LET	Assignment Statement (Not Mandatory)
RANDOMIZE	Reset Random Number "Seed"
INPUT [Prompt ,] Variables	Get Terminal Input
PRINT	Output to a Terminal
?	Output to a Terminal
READ	Get "Data" Information
GOTO	Jump to Line Number/Label
GOSUB	Call Subroutine at Line Number/Label
RETURN	Return to Caller from Subroutine
IF Condition [THEN] Statement	Conditionally Execute the "Statement"
FOR Variable = Init TO Last [STEP Inc] . . . NEXT [Variable]	Loop Specified Number of Times
ON Event GOTO	On an Event, Jump to Line Number/Label
RESTORE	Restore the "DATA" Pointer
STOP	Stop Program Execution
END	End Program Execution
'	Comment - Everything to the Right is Ignored

(continued)

TABLE 14 Basic Language Built In Function Statements *(Continued)*

STATEMENT	FUNCTION
REM	Comment - Everything to the Right is Ignored
ABS	Get Absolute Value of a Number
SGN	Return the Sign of a Number
COS	Return Cosine of an Angle (input usually in Radians)
SIN	Return Sine of an Angle (input usually in Radians)
SIN	Return Tangent of an Angle (input usually in Radians)
ATN	Return the Arc Tangent of a Ratio
INT	Convert Real Number to Integer
SQR	Return the Square Root of a Number
EXP	Return the Power of e for the input
LOG	Return the Natural Logarithm for the Input
RND	Return a Random Number
TAB	Set Tab Columns on Printer

For assignment and *if* statements, the operators shown in Table 15 are available in BASIC.

BASIC's order of Operations is quite standard for programming languages (Table 16).

The functions shown in Table 17 are available in Microsoft versions of BASIC for the PC, as well as some BASICs for the PICmicro® MCU.

PicBasic microEngineering Labs, Inc.'s (meLabs) PicBasic is an excellent tool for learning about the PICmicro® MCU, before taking the big plunge into assembly-language programming. The source code required by the compiler is similar to the Parallax Basic Stamp BS2's PBASIC, with many improvements and changes to make the language easier to work with and support different PICmicro® MCUs.

This section describes PicBasic Pro, which is the enhanced version of the compiler and can support the widest range of PICmicro® MCU part numbers. PicBasic Pro is a very complete application-development system that is able to convert your single-file applications into PICmicro® MCU statements very efficiently.

PicBasic does not currently have the ability to link together multiple source files, which means that multiple source files must be "included" in the overall source. Assembly-language statements are inserted in line to the application. PicBasic produces either assembler source files or completed .HEX files. It does not create object files for linking modules together.

For additional information and the latest device libraries, look at the microEngineering Labs, Inc. web page at **http://www.melabs.com/mel/home.htm.**

PicBasic Pro is an MS-DOS command-line application that is invoked using the statement:

```
PBP [options...] source
```

TABLE 15 Basic Language Built In Operators

OPERATOR	OPERATION
+	Addition
−	Subtraction
*	Multiplication
/	Division
^	Exponentiation
“	Start/End of Text String
’	Separator
;	Print Concatenation
$	String Variable Identifier
=	Assignment/Equals To Test
<	Less than
<=	Less than or Equals To
>	Greater than
>=	Greater than or Equals To
<>	Not Equals

It has also been compiled to run from the Microsoft Windows Protect mode, using a full 4.3-GB flat memory model to avoid out-of-memory errors that are possible with the MS-DOS version. To invoke the Protect mode version, open a MS-DOS prompt window in Microsoft Windows 95/98/NT/2000 and enter in the command:

```
PBPW [options...] source
```

TABLE 16 Basic Language Operator Order Of Operations

OPERATORS	PRIORITY	TYPE
Functions		Expression Evaluation
= <> < <= > >=	Highest	Conditional Tests
^		Exponentiation
* /		Multiplication/Division
+ −	Lowest	Addition/Subtraction

TABLE 17 Microsoft Basic Language Enhancement Functions

FUNCTION	OPERATION
AND	AND Logical Results
OR	OR Logical Results
XOR	XOR Logical Results
EQV	Test Equivalence of Logical Results
IMP	Test Implication of Logical Results
MOD	Get the Modulus (remainder) of an Integer Division
FIX	Convert a Floating Point Number to Integer
DEFSTR Variable	Define the Variable as a String (instead of the "DIM" Statement)
DEFINT Variable	Define the Variable as an Integer (instead of the "DIM" Statement)
DEFLNG Variable	Define the Variable as a "long" Integer (instead of the "DIM" Statement)
DEFSNG Variable	Define the Variable as a Single Precision Floating Point Number (instead of the "DIM" Statement)
DEFDBL Variable	Define the Variable as a Double Precision Floating Point Number (without using the "DIM" Statement)
REDIM Variable ([low TO]	Redefine a Variable High [, [low TO] High . . .]) [As Type]
ERASE	Erase an Array Variable from Memory
LBOUND	Return the First Index of an Array Variable
UBOUND	Return the Last Index of an Array Variable
CONST Variable = Value	Define a Constant Value
DECLARE Function | Subroutine	Declare a Subroutine/Function Prototype at Program Start
DEF FNFunction (Arg [, Arg . . .])	Define a Function ("FNFunction") that returns a Value. If a Single Line, then "END DEF" is not required
END DEF	End the Function Definition
FUNCTION Function (Arg [, Arg . . .])	Define a Function. Same Operation, Different Syntax as "DEF FNFunction"

TABLE 17 Microsoft Basic Language Enhancement Functions *(Continued)*

FUNCTION	OPERATION
END FUNCTION	End a Function Declaration
SUB Subroutine (Arg[, Arg . . .])	Define a "Subroutine" which does not returns a Value. If a Single Line, then "END DEF" is not required
END SUB	End the Subroutine Definition
DATA Value [, Value . . .]	Specify File Data
READ Variable [, Variable . . .]	Read from the "Data" File Data
IF Condition THEN Statements ELSE Statements END IF	Perform a Structured If/Else/Endif
ELSEIF	Perform a Condition Test/Structured If/Elseif/Endif instead of simply "Else"
ON ERROR GOTO Label	On Error Condition, Jump to Handler
RESUME [Label]	Executed at the End of an Error Handler. Can either return to current location, 0 (start of Application) or a specific label
ERR	Return the Current Error Number
ERL	Return the Line the Error Occurred at
ERROR #	Execute an Application-Specific Error (Number "#")
DO WHILE Condition Statements LOOP	Execute "Statements" while "Condition" is True
DO Statements LOOP WHILE Condition	Execute "Statements" while "Condition" is True
DO Statements LOOP UNTIL Condition	Execute "Statements" until "Condition" is True
Exit	Exit Executing "FOR", "WHILE" and "UNTIL" Loops without executing Check
SELECT Variable	Execute based on "Value" "CASE" Statements used to Test the Value and Execute Conditionally
CASE Value	Execute within a "SELECT" Statement if the "Variable" Equals "Value". "CASE ELSE" is the Default Case
END SELECT	End the "SELECT" Statement
LINE INPUT	Get Formatted Input from the User

(continued)

TABLE 17 Microsoft Basic Language Enhancement Functions *(Continued)*

FUNCTION	OPERATION
INPUT$ (#)	Get the Specified Number ("#") of Characters from the User
INKEY$	Check Keyboard and Return Pending Characters or Zero
ASC	Convert the Character into an Integer ASCII
Code	
CHR$	Convert the Integer ASCII Code into a Character
VAR	Convert the String into an into an Integer Number
STR$	Convert the Integer Number into a String
LEFT$ (String, #)	Return the Specified Number ("#") of Left Most Characters in "String"
RIGHT$ (String, #)	Return the Specified Number ("#") of Right Most Characters in "String"
MID$ (String, Start, #)	Return/Overwrite the Specified Number ("#") of Characters at Position "Start" in "String"
SPACE$ (#)	Returns a String of the Specified Number ("#") of ASCII Blanks
LTRIM$	Remove the Leading Blanks from a String
RTRIM$	Remove the Trailing Blanks from a String
INSTR (String, SubString)	Return the Position of "SubString" in "String"
UCASE$	Convert all the Lower Case Characters in a String to Upper Case
LCASE$	Convert all the Upper Case Characters in a String to Upper Case
LEN	Return the Length of a String
CLS	Clear the Screen
CSRLIN	Return the Current Line that the Cursor is On
POS	Return the Current Column that the Cursor is On

TABLE 17 Microsoft Basic Language Enhancement Functions *(Continued)*

FUNCTION	OPERATION
LOCATE X, Y	Specify the Row/Column of the Cursor (Top Left is 1, 1)
SPC	Move the Display the Specified Number of Spaces
PRINT USING "Format"	Print the Value in the Specified Format. "+", "#", ".", "^" Characters are used for number formats
SCREEN mode [, [Color] [, [Page] [, Visual]	Set the Screen Mode. "Color" is 0 to display on a "Color" display, 1 to display on a "Monochrome". "Page" is the Page that receives I/O and "Visual" is the Page that is currently active.
COLOR [foreground] [, [background] [,border]]	Specify the Currently Active Colors
PALETTE [attribute, color]	Change Color Assignments.
VIEW [[SCREEN] (x1, y1) − (x2, y2) [, [color]] [, border]]]	Create a small Graphics Window known as a "Viewport"
WINDOW [[SCREEN] (x1, y1) − (x2, y2)]	Specify the Viewport's logical location on the Display
PSET (x, y) [, color]	Put a Point on the Display
PRESET (x, y)	Return the Point to the Background Color
LINE (x1, y1) − (x2, y2) [, Color] [, [B | BF] [, style]]]	Draw a Line between the two specified points. If "B" or "BF" specified, Draw a Box ("BF" is "Filled)
CIRCLE (x, y), radius [, [color] [, [start] [, end] [, aspect]]]]	Draw the Circle at center location and with the specified "radius". "start" and "end" are starting and ending angles (in radians). "aspect" is the circle's aspect for drawing ellipses
DRAW CommandString	Draw an arbitrary Graphics Figure. There should be spaces between the commands Commands:
	U# - Moves Cursor up # Pixels
	D# - Moves Cursor down # Pixels
	E# - Moves Cursor up and to the right # Pixels

(continued)

TABLE 17 Microsoft Basic Language Enhancement Functions *(Continued)*

FUNCTION	OPERATION
	F# - Moves Cursor down and to the right # Pixels
	G# - Moves Cursor down and to the Left # Pixels
	H# - Moves Cursor up and to the left # Pixels
	L# - Moves Cursor Left # Pixels
	R# - Moves Cursor Right # Pixels
	Mxy - Move the Cursor to the Specified x, y Position
	B - Turn Off Pixel Drawing
	N - Turns On Cursor and Move to Original Position
	A# - Rotate Shape in 90 Degree Increments
	C# - Set the Drawing Color
	P# Color#Border - Set the Shape Fill and Border Colors
	S# - Set the Drawing Scale
	T# - Rotates # Degrees
LPRINT	Send Output to the Printer
BEEP	“Beep” the Speaker
SOUND Frequency, Duration	Make the Specified Sound on the PC’s Speaker
PLAY NoteString	Output the Specified String of “Notes” to the PC’s Speaker
DATE$	Return the Current Date
TIME$	Return the Current Time
TIMER	Return the Number of Seconds since Midnight
NAME FileName AS NewFileName	Change the Name of a File
KILL FileName	Delete the File
FILES [FileName.Ext]	List the File (MS-DOS “dir”). “FileName.Ext” can contain “Wild Cards”

TABLE 17 Microsoft Basic Language Enhancement Functions *(Continued)*

FUNCTION	OPERATION
OPEN FileName [FOR Access] AS # Handle	Open the File as the Specified Handle (Starting with the "#" Character). Access: I - Open for Text Input O - Open for Text Output A - Open to Append Text B - File is Opened to Access Single Bytes R - Open to Read and Write Structured Variables
CLOSE #Handle	Close the Specified Files
RESET	Close all Open Files
EOF	Returns "True" if at the End of a File
READ #Handle, Variable	Read Data from the File
GET #Handle, Variable	Read a Variable from the File
INPUT #Handle, Variable	Read Formatted Data from the File using
"INPUT",	"INPUT USING" and "INPUT$" Formats
WRITE #Handle, Variable	Write Data to the File
PUT #Handle, Variable	Write a Variable to a File
PRINT #Handle, Output	Write Data to the File using the "PRINT" and "PRINT USING" Formats
SEEK #Handle, Offset	Move the File Pointer to the Specified Offset within the File

The source is the MS-DOS (maximum eight character) application code source file name with the .BAS extension. *Options* are compiler execution options (Table 18).

I have not included a list of the different PICmicro® MCUs that can have their application developed by PicBasic simply because this list changes along with the different parts that are available from Microchip. meLabs works very hard to ensure that all new PICmicro® MCU part numbers are supported as quickly as possible after they are announced. Information on what part numbers the latest version of PicBasic supports can be found at meLabs' web site.

PicBasic does assume a constant set of configuration values. For most PICmicro® MCUs, the configuration fuses are set as shown in Table 19.

TABLE 18 PicBasic Command Line Options

OPTION	FUNCTION
-h/-?	Display the help screen. The help screen is also displayed if no options or source file name is specified
-ampasm	Use the MPASM Assembler and not the PicBasic Assembler
-c	Insert Comments into PicBasic Compiler produced Assembler Source File. Using this option is recommended if you are going to produce MPASM Assembler Source from PicBasic
-iPath	Specify a new directory "Path" to use for include files in PicBasic
-lLibrary	Specify a different library to use when compiling. Device specific libraries are provided by PicBasic when the processor is specified
-od	Generate a listing, symbol table and map files
-ol	Generate a listing file
-pPICmicro® MCU	Specify the "PICmicro® MCU" that the source is to be compiled into. If this parameter is not specified, then a PIC16F84 is used as the processor. "PICmicro® MCU" is in the format: 16F84, where the "PIC" at the start of the Microchip part number is not specified.
-s	Do not assemble the compiled code
-v	Turn on "Verbose Mode" which provides additional information when the application is compiled

When you program the PICmicro® MCU, you should be comfortable with the oscillator and PWRTE value. The watchdog timer should be left enabled because the *NAP* and *SLEEP* instructions use it.

Along with using the MS-DOS command line, PicBasic can be used with MPLAB. This is the preferred way of working with PicBasic because it works seamlessly with the Microchip tools. Chapter 16 includes two projects that used PicBasic Pro for the application's source code to demonstrate how PicBasic works with MPLAB.

To use PicBasic with MPLAB, after installing PicBasic, select *Install Language Tool* under MPLAB's *Project* pull down and select *microEngineering Labs, Inc.* for the *Lan-*

TABLE 19 PicBasic Default Configuration Fuse Settings

FEATURE	PICBASIC SETTING
Code Protect	Off
Oscillator	XT - or Internal RC if 12Cxxx
WDT	On
PWRTE	Off

guage Suite, followed by the PicBasic that you are using. Figure 10 shows the *Install Language Tool* dialog box with the tool selection being made. After *PicBasic* or *PicBasic Pro* has been selected, click on *Browse* to find PBC.EXE or PBPW.EXE. Finally, be sure that the *Command Line* radio button is selected and click on *OK*. After doing this, you will be able to create PicBasic projects as if they were assembler projects.

The only issue I have with using the PicBasic compiler is that the project and source files must reside in the same subdirectory as the PicBasic compiler and its associated files. This breaks up my normal strategy of keeping all the project files in a directory that is different from the tool directory.

When you develop your PicBasic applications, I recommend that you copy them into a separate subdirectory when you are finished and delete them from the PicBasic execution directory. This avoids ending up with large, confusing PicBasic subdirectories and no idea what is in them.

The source code used with PicBasic should not be surprising for you if you've ever worked with a high-level language before. The four different types of programming constructs that I introduced earlier in this appendix are well represented in the language. The language is not procedural based, like C, but it does allow multiple subroutines and does have many built-in instructions that you can use to your advantage.

The starting point of the language is the *label*. Labels are used to specify execution addresses, variables, hardware registers, and constant strings. Like most other languages, PicBasic labels are case sensitive and can include the characters *A* to *Z*, *0* to *9*, and _. A numeric cannot be used as the first character of the label.

Labels in the application code are used to provide *jump to* and *call* addresses. They are indicated by written into the application code, starting at the left-most column of the source file and terminated with a colon (*:*). To access the label, the label name is used in a *goto, call,* or *branch* statement. For example, to implement an endless loop, you would use the label declaration and *goto* statement:

```
Loop:                  ' Return here after running through the Code
   :                   ' Code to Execute inside the loop
 goto Loop             ' Jump back to "Loop" and Start Executing
                       ' again
```

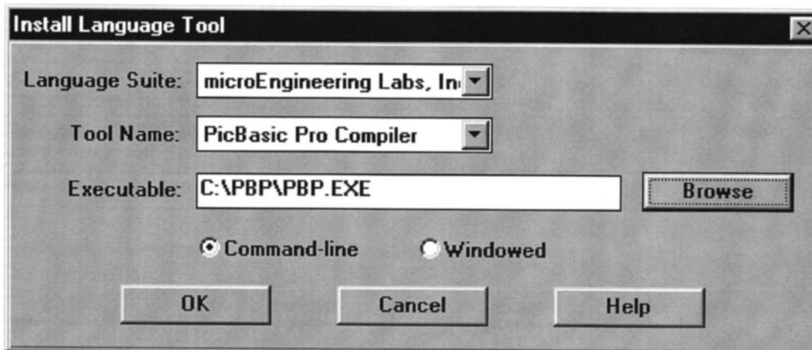

Figure 10 **PicBasic MPLAB installation tool**

Absolute addresses are not available within PicBasic, the labels should always be defined and referenced. The interrupt vector (address 0x004) in the mid-range PICmicro® MCUs is described in the following paragraphs.

In this example, I have placed a number of comments after the single quote (′) character. The compiler will ignore everything to the right of the single quote character, just as it is for the comment character (;) in assembly-language programming.

Variables in PicBasic can be defined one of two different ways. Each way reflects a method that was used by the two different Basic Stamps. In the BS1, variables are predefined in the system and broken up either into 16-bit words (which are given the identifiers, *w0*, *w1*, *w2*, etc.), eight-bit "bytes" (with the identifiers *b0*, *b1®MDRV¯*...), and bits (with the identifiers *Bit#* or just the bit number). The maximum number of variables in the BS1 system is defined by the processor used (in the BS1, which uses the PIC16C54, only 14 bytes are available in memory).

Each byte occurs in one of the words. For example, b4 is the least-significant byte of w2. The 16-bit variables are defined as being a part of the 16 bits taken up by w0 (b0 and b1). This method works well, but care has to be taken to ensure that the overlapping variables are kept track of and not used incorrectly. The most common problem for new Basic Stamp developers is defining a variable on b0 and w0 and having problems when a write to one variable overwrites the other.

To provide these variables to the PicBasic application, the Basic Stamp variable declaration files are defined in the following two include files that are shown within *include* statements below. Only one of these statements can be used in an application.

```
include "bs1defs.bas"
include "bs2defs.bas"
```

A much better way to declare variables is to use the *var* directive to select the different variables at the start of the application and let the PicBasic compiler determine where the variables belong and how they are accessed (i.e., put in different variable pages). Along with the *var* directive, the *word, byte,* and *bit* directives are used to specify the size of the variable. Some example variable declarations are:

```
WordVariable var word       ' Declare a 16 Bit Variable
ByteVariable var byte       ' Declare an 8 Bit Variable
BitVariable var bit         ' Declare a single byte Variable
```

Initial values for the variables cannot be made at the variable declarations.

Along with defining variables, the *var* directive can be used to define variable labels built out of previously defined variables to specify specific data. Using these variables, I can break *WordVariable* up into a top half and bottom half and *ByteVariable* into specific bytes with the statements:

```
WordVariableTop       var WordVariable.byte1
WordVariableBottom    var WordVariable.byte0

BitVariableMSB        var BitVariable.bit7
BitVariableLSB        var BitVariable.0
```

Variables can also be defined over registers. When the PicBasic libraries are merged with the source code, the standard PICmicro® MCU register names are available within

the application. Using this capability, labels within the application can be made to help make the application easier to work with. For example, to define the bits needed for an LCD, the following declarations could be used:

```
LCDData   var PORTB          ' PORTB as the 8 Bits of Data
LCDE      var PORTA.0        ' RA0 is "E" Strobe
LCDRS     var PORTA.1        ' RA1 is Data/Instruction Select
LCDRW     var PORTA.2        ' RA2 is the Read/Write Select Bit
```

When variables are defined using the *var* and *system* directives, specific addresses can be made in the application. For example, the statement:

```
int_w var byte $0C system
```

will define the variable *_w* at address 0x00C in the system. This reserves address 0x00C and does not allow its use by any other variables.

The bank of a variable can be specified using the *system* directive, like:

```
int_status var byte bank0 system
```

These two options to the *var* directive are useful when defining variables for interrupt handlers.

Along with redefining variables with the *var* statement, PicBasic also has the *symbol* directive. The *symbol* directive provides the same capabilities as the *var* statement and it is provided simply for compatibility with the BS1. If you are only developing PicBasic applications, I would recommend only using *var* and avoiding the *symbol* directive.

Single-dimensional arrays can be defined within PicBasic for each of the three data types when the variable is declared.

```
WordArray var word[10]       ' Ten Word Array
ByteArray var byte[11]       ' Eleven Byte Array
BitArray var bit[12]         ' Twelve Bit Array
```

Note that bits can be handled as an array element, which is a really nice feature to the language. Depending on the PICmicro® MCU part number, the maximum array sizes are shown in Table 20.

As part of the "bit" definition, I/O port pins are predefined within PicBasic. Up to 16 pins (addressed using the *Pin#* format, where # is the pin number) can be accessed, although how they are accessed changes according to the PICmicro® MCU part number that the application is designed for. The pins for different parts are defined in Table 21.

TABLE 20 PicBasic Array Restrictions

VARIABLE TYPE	MAXIMUM NUMBER OF ELEMENTS
Word	32
Byte	64
Bit	128

TABLE 21 PicBasic Pin To PICmicro® MCU Specifications

NUMBER OF PINS	PINS 0 – 7	PINS 8 – 15
8	GPIO	Mapped onto Pins 0 – 5
18	PORTB	PORTA
28 - Not PIC14C000	PORTB	PORTC
14C000	PORTC	PORTD
40 Pin	PORTB	PORTC

Notice that not all the ports that have all eight pins specified. For example, accessing *pin6* in an 18-pin device (which does not have an RA6 bit) will not do anything.

Constants are declared in a similar manner to constants but using the *con* directive with a constant parameter:

```
SampleConstant con 3 + 7      ' Define a Sample Constant
```

Constant values can be in four different formats. Table 22 lists the different formats and the modifiers to indicate to the PicBasic compiler which data type is being specified (Table 22).

In Table 22, notice that only an ASCII byte can be passed within double quotes. Some instructions (following) can be defined with strings of characters that are enclosed within double quotes.

The *define* statement is used to change constants given defaults within the PICmicro® MCU when a PicBasic-compiled application is running. The format is:

```
DEFINE Label NewValue
```

The labels, their default values and their values are listed in Table 23.

The OSC define should be specified if serial I/O is going to be implemented in the PICmicro® MCU. This value is used by the compiler to calculate the time delays necessary for advanced operations.

TABLE 22 PicBasic Data Types

DATA TYPE	MODIFIER	COMMENTS
Decimal	None	PicBasic Default is Decimal
Hex	$	"$" is placed before the Number
Binary	%	"%" is placed before the Number
ASCII	"	Double Quotes placed around a Single Character

TABLE 23 PicBasic Internal Operating Parameters

DEFINE	DEFAULT	OPTIONAL VALUES	COMMENTS
BUTTON_PAUSE	10	Any Positive Int	Button Debounce Delay in msecs
CHAR_PACING	1000	Any Positive Int	Time between SerOut Characters
DEBUG_BAUD	2400	Any	Specified Data Rate of Debug information
DEBUG_BIT	0	0 – 7	Output Pin for Debug Serial Output
DEBUG_MODE	1	0, 1	Polarity of Debug NRZ Output data. “0” Positive, “1” Inverted
DEBUG_PACING	1000	Any Positive Int	Time between Output Characters for DEBUG Statements
DEBUG_REG	PORTB	Any PORT	Port Debug Output Pin is Connected to
DEBUGIN_BIT	0	0 – 7	Input Pin for Debug Serial Output
DEBUGIN_MODE	1	0, 1	Polarity of Debug NRZ Input data. “0” Positive, “1” Inverted
DEBUGIN_REG	PORTB	Any PORT	Port Debug Input Pin is Connected to
HSER_BAUD	2400	Any	Hardware Serial Port’s Data Rate
HSER_SPBRG	25	0 – 0x0FF	Hardware Serial Port’s SPBRG Register Value
HSER_RCSTA	0x090	0 – 0x0FF	Hardware Serial Port’s Initialization value for “RCSTA” register. Default set for Asynchronous Communications
HSER_TXSTA	0x020	0 – 0x0FF	Hardware Serial Port’s Initialization value for “TXSTA” register. Default set for Asynchronous Communications
HSER_EVEN	1	0, 1	Hardware Serial Port’s Parity Select Values. Only used if Parity checking is desired
HSER_ODD	1	0, 1	Hardware Serial Port’s Parity Select Values. Only used if Parity checking is desired
I2C_HOLD	1	0, 1	Stop I2C transmission while the SCL line is held low
I2C_INTERNAL	1	0, 1	Set to use Internal EEPROM in the 12Cexx PICmicro® MCUs

(continued)

TABLE 23 PicBasic Internal Operating Parameters (*Continued*)

DEFINE	DEFAULT	OPTIONAL VALUES	COMMENTS
I2C_SCLOUT	1	0, 1	Use a Bipolar Driver instead of an Open-Drain I2C Clock Driver
I2C_SLOW	1	0, 1	Run the I2C at no more than 100 kbps data rate
LCD_BITS	4	4, 8	Number of Bits for LCD Interface
LCD_DBIT	0	0, 4	Specify the Data bit for LCD Data
LCD_DREG	PORTA	Any PORT	Select the LCD Data Port
LCD_EBIT	3	0 – 7	Specify the Bit for the LCD Clock
LCD_EREG	PORTB	Any PORT	Specify the Port for the LCD “E” Clock Bit
LCD_LINES	2	1, 2	Specify the Number of Lines on the LCD. Check Information on LCDs for how the multiple line specification is used in some single line LCDs.
LCD_RSBIT	4	Any PORT	LCD RS Bit Selection
LCD_RSREG	PORTA	Any PORT	LCD RS Bit Select Register
OSC	4	3, 4, 8, 10, 12, 16, 20	Specify PICmicro® MCU Operating Speed in MHz. Note “3” is actually 3.58
OSCCAL_1K	1	0, 1	Set OSCCAL for PIC12C672
OSCCAL_2K	1	0, 1	Set OSCCAL for PIC12C672
SER2_BITS	8	4 – 8	Specify Number of bits sent with “SERIN2” and “SEROUT2” instructions

Assembly language can be inserted at any point within a PicBasic application. Single instructions can be inserted by simply starting the line with an @ character:

```
@ bcf INTCON, T0IF    ; Reset T0IF Flag
```

Multiple lines of assembly language are prefaced by the *asm* statement and finished with the *endasm*. An example of this is:

```
asm
  movlw 8                   ; Loop 8x
Loop
  bsf PORTA, 0              ; Pulse the Bit
  bcf PORTA, 0
  addlw $0FF                ; Subtract 1 from "w"
  btfss STATUS, Z           ; Do 8x
   goto Loop
endasm
```

Notice that labels inside the assembler statements do not have a colon at the end of the string and that traditional assembly-language comment indicators (the semi-colon *;*) is used.

Implementing interrupt handlers in PicBasic can be done one of two ways. The simplest way of implementing it is using the *ON INTERRUPT GOTO Label* statement. Using this statement, any time an interrupt request is received, the label specified in the *ON INTERRUPT* statement will be executed until there is a resume instruction which returns from an interrupt. Using this type of interrupt handler, straight PicBasic statements can be used and assembly-language statements avoided.

The basic operation looks like:

```
   :
  ON INTERRUPT GOTO IntHandler
   :
IntHandler:
  disable                        ' Turn off interrupt and debug requests
   :                             ' Process Interrupt
enable                           ' Enable other Interrupts and debug
                                 ' requests
resume                           ' Return to the executing code
```

The problem with this method is the interrupt handler is executed once the current instruction has completed. If a very long instruction is being executed (for example, a string serial send), then the interrupt will not be serviced in a timely manner.

The best way of handling an interrupt is to add the interrupt handler as an assembly-language routine. To reference the interrupt handler, the *define INTHAND Label* instruction is used to identify the label where the assembly-language code is listed. The interrupt handler will be moved to start at address 0x004 in the mid-range devices.

A code template for generic mid-range PICmicro® MCU interrupt handlers is:

```
  int_w var byte 0x020 system      ' Define the Context Save Variables
  int_status var byte bank0 system
  int_fsr var byte bank0 system
  int_pclath byte bank0 system
```

```
   :
  define INTHAND IntHandler        ' Specify what the Interrupt
                                   ' Handler is
   :
' Interrupt Handler — to be relocated to 0x00004
asm
IntHandler
  movwf int_w                      ; Save the Context Registers
  movf STATUS, w
  bcf STATUS, RP0                  ; Move to bank 0
  bcf STATUS, RP1
  movwf int_status
  movf FSR, w
  movwf int_fsr
  movf PCLATH, w
  movwf int_pclath
;  #### - Execute Interrupt Handler Code Here
  movf int_pclath, w               ; Finished, restore the Context
  movwf PCLATH                     ; Registers
  movf int_fsr, w
  movwf FSR
  movf int_status, w
  movwf STATUS
  swapf int_w, f
  swapf int_w, w
  retfie
endasm
```

If you are reading through this appendix to try and understand programming better, you might be confused by my use of assembly language for the PICmicro® MCU here. The reason why I included the two assembly-language examples was for a later reference for you. As you read through the text of the book, the two assembly-language snippets will make a lot more sense to you.

In the interrupt template, notice that I am working through a "worst case" condition. I am assuming that execution is taking place in something other than *bank 0,* the FSR has been given a specific value along with the PICmicro® MCU having more than one page of program memory. If you use this template in your own applications, I do not recommend that this code be changed. The only way you can be sure that the index register is not being used and execution will be in bank 0, along with the processor taking program memory addresses out of page 0, is to look at the code produced by the compiler. Rather than going to this trouble, you should just use this template and insert your interrupt-handler code at the comment with #### inside it.

You should be aware of one issue when you are adding an assembly-language interrupt handler and that is you should ensure that no crucially timed PicBasic are executing when the interrupt is acknowledged. If, for example, a serial transfer was occurring when the interrupt request was received and acknowledged, then you would end up changing the timing and the data that is potentially received or sent. To ensure that this doesn't happen, you will have to disable interrupts while the crucially timed operation is executing.

Mathematical operators used in assignment statements and PicBasic instructions are very straightforward in PicBasic and work conventionally. In Basic Stamp PBASIC, you must remember that the operations execute from left to right. Thus, the statement:

```
A = B + C * D
```

which you would expect to operate as:

1 Multiply *C* and *D*
2 Add the results from 1 to *B*

In PBASIC, it returns the result of:

1 Get the sum of *B* and *C*
2 Multiply the results from 1 with *D*

PicBasic does not follow the PBASIC evaluation convention and returns the expected result from complex statements like the previous one. Thus, in PicBasic, you do not have to break complex statements up into single operations, like you do in PBASIC, to avoid unexpected expression evaluation. If you are using a Basic Stamp to "prototype" PicBasic applications, then I recommend that you break up the complex statements and use the temporary values as shown elsewhere in the book.

The mathematical operators are listed in Table 24, along with their execution priority and parameters. All mathematical operators work with 16-bit values.

Along with the mathematical operators, the *if* statement provides the following test conditions. This is listed in the following table. Notice that both the BASIC standard labels as well as the C standard labels are used. *Parm1* and *Parm2* are constants, variables, or statements made up of variables statements, along with the different mathematical operators and test conditions.

When a test condition is true, a nonzero is returned, if it is false, then a zero is returned. Using this convention, single variable parameters can be tested in *if* statements, rather than performing comparisons of them to zero. The comparison operators are listed in Table 25.

The PicBasic instructions are based on the Parallax Basic Stamp PBASIC language. Although there are a lot of similarities, they are really two different languages. Table 26 lists all the PicBasic instructions and indicated any special considerations that should be made for them, with respect to being compiled in a PICmicro® MCU.

These instructions are really library routines that are called by the mainline of the application. I am mentioning this because you will notice that the size of the application changes based on the number of instructions that are used in the application. You might find that you can drastically reduce the program memory size by looking at the different instructions that are used and change the statements to assembler or explicit PicBasic statements.

When I specified the various instructions, notice that the square brackets (*[* and *]*) are used to specify data tables in some instructions. For this reason, I have specified optional values using braces (*{* and *}*), which breaks with the conventions used in the rest of the book.

Visual Basic Microsoft's Visual Basic is probably the fastest way to get into Microsoft Windows application programming. This ease of using the language and development system also makes it great as a "what if" tool and allows you to write an application quickly to try out new ideas.

To create an application, the Primary dialog box (which is known as a *form,* Fig. 11) is created first, with different features (I/O boxes, buttons, etc.). These features are known as *controls* within Visual Basic. With the Window defined, by simply clicking on the different controls, subroutine prototypes to handle "events" (such as mouse clicks over these

TABLE 24 PicBasic Mathematical and Bit Operations

<table>
<tr><th>PRIORITY</th><th>OPERATOR</th><th>OPERATION</th></tr>
<tr><td>Lowest</td><td>Parm1 + Parm2</td><td>Return the Sum of “Parm1” and “Parm2”</td></tr>
<tr><td></td><td>Parm1 − Parm2</td><td>Return the Result of “Parm2” Subtracted from “Parm1”</td></tr>
<tr><td></td><td>Parm1 * Parm2</td><td>Return the least significant sixteen bits of the product of “Parm1” and “Parm2”. This is often referred to as Bytes 0 and 1 of the result</td></tr>
<tr><td></td><td>Parm1 */ Parm2</td><td>Return the middle sixteen bits of the product of “Parm1” and “Parm2”. This is often referred to as Bytes 1 and 2 of the result</td></tr>
<tr><td></td><td>Parm1 ** Parm2</td><td>Return the most significant sixteen bits of the product of “Parm1” and “Parm2”.This is often referred to as Bytes 2 and 3 of the result</td></tr>
<tr><td></td><td>Parm1 / Parm2</td><td>Return the number of times Parm2 can be divided into Parm1 evenly</td></tr>
<tr><td></td><td>Parm1 // Parm2</td><td>Return the remainder from dividing Parm2 into Parm1. This is known as the “Modulus”.</td></tr>
<tr><td></td><td>Parm1 & Parm2</td><td>Return the bitwise value of “Parm1” AND “Parm2”</td></tr>
<tr><td></td><td>Parm1 | Parm2</td><td>Return the bitwise value of “PARM1” OR “Parm2”</td></tr>
<tr><td></td><td>Parm1 ^ Parm2</td><td>Return the bitwise value of “PARM1” XOR “Parm2”</td></tr>
<tr><td></td><td>~ Parm1</td><td>Return the inverted bitwise value of “PARM1”</td></tr>
<tr><td></td><td>Parm1 &/ Parm2</td><td>Return the inverted bitwise value of “Parm1” AND “Parm2”</td></tr>
<tr><td></td><td>Parm1 | / Parm2</td><td>Return the inverted bitwise value of “PARM1” OR “Parm2”</td></tr>
<tr><td></td><td>Parm1 ^/ Parm2</td><td>Return the inverted bitwise value of “PARM1” XOR “Parm2”</td></tr>
</table>

	Parm1 ≪ Parm2	Shift “Parm1” to the left “Parm2” bits. The new least significant bits will all be zero
	Parm1 ≫ Parm2	Shift “Parm1” to the right “Parm2” bits. The new most significant bits will all be zero
	ABS Parm1	Return the magnitude of a number. (“ABS −4” is equal to “ABS 4” and returns “4”)
	Parm1 MAX Parm2	Return the higher Parameter
	Parm1 MIN Parm2	Return the lower Parameter
	Parm1 DIG Parm2	Return Digit Number “Parm2” (Zero Based) of “Parm1”. (“123 DIG 1” returns “2”)
	DCD Parm1	Return a value with only the “Parm1” bit Set. (“DCD 4” returns “%00010000”)
	NCD Parm1	Return the Bit number of the highest set bit in “Parm1”
	Parm1 REV Parm2	Reverse the Bits in “Parm1” from zero to “Parm2”. (“%10101100 REV 4” will return “%10100011”)
	SQR Parm1	Return the Integer Square Root of “Parm1”
	SIN Parm1	Return the Trigonometric “Sine” of “Parm1”. The returned value will be based on a circle of radius 127 and 256 degrees (not the traditional 360)
Highest	COS Parm1	Return the Trigonometric “Cosine” of “Parm1”. The returned value will be based on a circle of radius 127 and 256 degrees (not the traditional 360)

TABLE 25 PicBasic Logical Operations

TEST CONDITION	DESCRIPTION
Parm1 = Parm2	Return a Non-Zero if “Parm1” equals Parm2”
Parm1 == Parm2	Return a Non-Zero if “Parm1” equals “Parm2”
Parm1 <> Parm2	Return a Non-Zero if “Parm1” does not equal “Parm2”
Parm1 ! = Parm2	Return a Non-Zero if “Parm1” does not equal “Parm2”
Parm1 < Parm2	Return a Non-Zero if “Parm1” is less than “Parm2”
Parm1 <= Parm2	Return a Non-Zero if “Parm1” is less than or equal to “Parm2”
Parm1 > Parm2	Return a Non-Zero if “Parm1” is greater than “Parm2”
Parm1 >= Parm2	Return a Non-Zero if “Parm1” is greater than or equal to “Parm2”
Parm1 AND Parm2	Return a Non-Zero if “Parm1” is Non-Zero and “Parm2” is Non-Zero
Parm1 && Parm2	Return a Non-Zero if “Parm1” is Non-Zero and “Parm2” is Non-Zero
Parm1 OR Parm2	Return a Non-Zero if “Parm1” is Non-Zero or “Parm2” is Non-Zero
Parm1 || Parm2	Return a Non-Zero if “Parm1” is Non-Zero or “Parm2” is Non-Zero
Parm1 XOR Parm2	Return a Non-Zero if “Parm1” and “Parm2” are different logical values.
Parm1 ^^ Parm2	Return a Non-Zero if “Parm1” and “Parm2” are different logical values.
Parm1 NOT AND Parm2	Return Zero if “Parm1” is Non-Zero and “Parm2” is Non-Zero
Parm1 NOT OR Parm2	Return Zero if “Parm1” is Non-Zero or “Parm2” is Non-Zero
Parm1 NOT XOR Parm2	Return a Non-Zero if “Parm1” and “Parm2” are in the same logical state.

features) are automatically created. Additional features in Visual Basic’s source-code editor allow you to specify the control parameters (known as *properties*).

This section does not include a description of the MSComm control that can be used to provide serial interfaces from the PC to the PICmicro® MCU. I have described it in detail in the “PC Interfacing” chapter because I consider it the primary method of interfacing a PICmicro® MCU to the PC because of the serial port’s standard interface and isolation from the PC’s and PICmicro® MCU’s hardware.

TABLE 26 PicBasic Built In Functions

INSTRUCTION	DESCRIPTION
BRANCH Index, [Label (,Label . . . }]	Jump to the *Label* specified by the value in *Index. Index* is zero based, so an *Index* of zero will cause execution jump to the first *Label,* an *Index* of one will cause execution to jump to the second *Label,* etc. This instruction only jumps within the current page. If a PICmicro® MCU with more than one page of program memory is used, then the *BRANCHL* instruction is recommend.
BRANCHL Index, [Label {,Label . . . }]	Jump to the *Label* specified by the value in *Index. Index* is zero based, so an *Index* of zero will cause execution jump to the first *Label,* an *Index* of one will cause execution to jump to the second Label, etc. This instruction can jump anywhere in PICmicro® MCU program memory.
BUTTON Pin, Down, Delay, Rate, Bvar, Action, Label	Jump to *Label* when the button has been pressed for the specified number of *BUTTON* instruction invocations. *Down* is the pin state for the switch to be assumed to be closed. *Delay* is the number of invocations the *Down* state has to be true before the *Action* is taken. *Rate* is how many invocations after the first *BUTTON* jump is true that an autorepeat happens. *Bvar* is a byte-sized variable only used in this function. *Action* is whether or not you want the jump to occur when the key is pressed (1) or released (0)
CALL Label	Execute the assembly-language *call* instructions.
CLEAR	Load all the variables with zero.
COUNT Pin, Period, Variable	Count the number of pulses on *Pin* that occur in *Period* ms.
DATA @Location, Constant {,Constant . . . }	Store constants in data EEPROM starting at *Location* when the PICmicro® MCU is programmed. For data at different addresses, use multiple *DATA* statements.
DEBUG Value {,Value . . . }	Define the *DEBUG* pin as output with the serial output parameters used in the *DEBUG* defines at reset. When this instruction is executed, pass the parameter data. If an ASCII # (0x023) is sent before a *Value,* the decimal numeric is sent, rather than the ASCII byte. This instruction (and *DEBUGIN*) can be used for serial I/O because they take up less space than the *SERIN* and *SEROUT* instructions

(continued)

TABLE 26 PicBasic Built In Functions *(Continued)*

INSTRUCTION	DESCRIPTION
DEBUGIN {TimeOut, Label,} [Variable {,Variable . . . }]	Define the *DEBUGIN* pin as an input with the serial input parameters used in the *DEBUGIN* defines at reset. When this instruction is executed, wait for a data byte to come in or jump to the label if the *TimeOut* value (which is specified in ms) is reached
DISABLE	*DISABLE* interrupts and debug operations. Interrupts will still be acknowledged but *ON INTERRUPT GOTO* will not execute
DISABLE INTERRUPT	*DISABLE* interrupts and debug operations. Interrupts will still be acknowledged but *ON INTERRUPT GOTO* will not execute
DTMFOUT Pin, {On, Off,} [Tone{,Tone . . . }]	Output the TouchTone sequence on the specified pin. Tones *0* through *9* are the same as on the telephone keypad. Tone 10 is the * key and tone 11 is the *#* key. Tones 12 through 15 correspond to the extended key standards for *A* to *D*. Filtering is required on the pin output to "smooth" out the signal output
EEPROM Location, [Constant { [,Constant . . . }]	Store new values in EEPROM when the PICmicro® MCU is programmed. This instruction is the same as *DATA*.
ENABLE	Enable debug and interrupt processing that was stopped by *DISABLE*.
ENABLE DEBUG	Enable debug operations that were stopped by *DISABLE*.
ENABLE INTERRUPT	Enable interrupt operations that were stopped by the DISABLE and DISABLE INTERRUPT instructions.
END	Stop processing the application and put the PICmicro® MCU in a low-power *Sleep* mode.
FOR Variable = Start To Stop {STEP Value} : NEXT {Variable}	Execute a loop, first initializing *Variable* to the *Start* value until it reaches the *Stop* value. The increment value defaults to one if no *STEP* value is specified. When *NEXT* is encountered, *Variable* is incremented and tested against the *Stop* value.
FREQOUT Pin, On, Frequency {, Frequency}	Output the specified *Frequency* on the *Pin* for *On* ms. If a second *Frequency* is specified, output this at the same time. Filtering is required on the pin output to "smooth" out the signal output.
GOSUB Label	Call the subroutine that starts at address *Lable*. The existence of *Label* is checked at compile time.

GOTO Label	Jump to the code that starts at address *Label*.
High Pin	Make *Pin* an output and drive it high.
HSERIN {ParityLabel,} {TimeOut, Label,} [Variable {[,Variable . . . }]	Receive one or more bytes from the built in USART (if present). The “ParityLabel” will be jumped to if the parity of the incoming data is incorrect. To use “ParityLabel”, make sure the “HSER_EVEN” or “HSER_ODD” defines have been specified.
HSEROUT [Value {,Value . . . }]	Transmit one or more bytes from the built in USART (if present).
I2CREAD DataPin, ClockPin, ControlByte, {Address,} [Variable {,Variable . . . }]{,NoAckLabel}	Read a byte string from an I2C device. The *ControlByte* is used to access the device with block or device-select bits. This instruction can be used to access internal EEPROM in the PIC12CExxx devices by entering the *define I2C_INTERNAL 1* statement at the start of the application code.
I2CWRITE DataPin, ClockPin, Control, {Address,} [Value {,Value . . . }] {,NoAckLabel}	Send a byte string to an I2C device. The *ControlByte* is used to access the device with block or device select bits. This instruction can be used to access internal EEPROM in the PIC12CExxx devices by entering the *define I2C_Internal 1* statement at the start of the application code.
IF Comp THEN Label Label	Evaluate the *Comp* comparison expression. If it is not equal to zero, then jump to *Label*.
IF Comp THEN Statement : {ELSE Statement :} ENDIF	Evaluate the *Comp* comparison expression. If it is not equal to zero, then execute the *Statements* below until either an *ELSE* or an *ENDIF* statement is encountered. If an *ELSE* statement is encountered, then the code after it, to the *ENDIF* instruction, is ignored. If *Comp* evaluates to zero, then skip over the *Statements* after the *IF* statement are ignored to the *ELSE* or *ENDIF* statements, after which any statements are executed.
INCLUDE “file”	Load in FILE, BAS in the current directory and insert it at the current location in the source file.
INPUT Pin	Put the specified pin into Input mode.
{LET} Assignment	Optional instruction value for an *Assignment* statement.
LCDOUT Value {,Value . . . }	Send the specified bytes to the LCD connected to the PICmicro® MCU. The LCD’s operating parameters are set with the *LCD* defines. To send an instruction byte to the LCD, a *$0FE* byte is sent first

(continued)

TABLE 26 PicBasic Built In Functions *(Continued)*

INSTRUCTION	DESCRIPTION
LOOKDOWN offset, [Constant {,Constant . . . }], Variable	Go through a list of constants with an offset and store the constant value at the offset in the second *Variable*. If the offset is greater than the number of constants, then *Zero* is returned in *Variable. Offset* is zero based, the first constant is returned if the offset is equal to zero.
LOOKDOWN2 offset, {Test} [Constant {,Constant . . . }], Variable	Search the list and find the constant value that meets the condition *Test.* If *Test* is omitted, then the *LOOKDOWN2* instruction behaves like the *LOOKDOWN* instruction with the *Test* is assumed to be and equals sign (=).
LOOKUP Variable, [Constant {,Constant . . . }], Variable	Compare the first *Variable* value with a constant string and return the offset into the constant string in the second *Variable.* If there is no match, then the second *Variable* is not changed.
LOOKUP2 Variable, [Value {,Value . . . }], Variable	Compare the first *Variable* value with a *Value* string and return the offset into the *Value* string in the second *Variable*. If there is no match, then the second *Variable* is not changed. *LOOKUP2* differs from *LOOKUP* because the *Values* can be 16-bit variables.
LOW Pin	Make *Pin* an output pin and drive it with a high voltage.
NAP Period	Put the PICmicro® MCU to "sleep" for the period value, which is given in the table:

Period	Delay
0	18 ms
1	36 ms
2	73 ms
3	144 ms
4	288 ms
5	576 ms
6	1,152 ms
7	2,304 ms

ON DEBUG GOTO Label	When invoked, every time an instruction is about to be invoked, the Debug monitor program at *Label* is executed. Two variables, the *DEBUG_ADDRESS* word and *DEBUG_STACK* byte must be defined as bank0 system bytes. To return from the debug monitor, a *RESUME* instruction is used.
ON INTERRUPT GOTO Label	Jump to the interrupt handler starting at *Label*. When the interrupt handler is complete, execute a *RESUME* instruction.
OUTPUT Pin	Put *Pin* into Output mode.
PAUSE Period	Stop the PICmicro® MCU from executing the next instruction for *Period* milliseconds. *PAUSE* does not put the PICmicro® MCU to “sleep” like *NAP* does.
PAUSEUS Period	Stop the PICmicro® MCU from executing the next instruction for *Period* microseconds.
PEEK Address, Variable	Return the Value at the register “Address” in *Variable.*
POKE Address, Value	Write the register *Address* with the *Value*.
POT Pin, Scale, Variable	Read a potentiometer’s wiper when one of its pins is connected to a capacitor. *Scale* is a value that will change the returned value until it is in the range of 0 to 0x0FF (255).
PULSIN Pin, State, Variable	Measure an incoming pulse width of *Pin. State* indicates the state of the expected pulse. If a 4-MHz clock is used with the PICmicro® MCU, the time intervals have a granularity of 10 μs.
PULSOUT Pin, Period	Pulse the *Pin* for the *Period*. If the PICmicro® MCU is run with a 4-MHz clock, then the pulse *Period* will have a granularity of 10 μs.
PWM Pin, Duty, Cycle	Output a pulse-width modulated signal on *Pin*. Each cycle is 5 ms long for a PICmicro® MCU running at 4 MHz. *Duty* selects the fraction of the cycles (zero to 255) that the PWM is active. *Cycle* specifies the number of cycles that is output.
RANDOM Variable	Load *Variable* with a pseudo-random *Variable*.
RCTIME Pin, State, Variable	Measure the absolute time required for a signal to be delayed in a RC network. If a 4-MHz oscillator is used with the PICmicro® MCU, then the value returned will be in 10-μs increments.
READ Address, Variable	Read the byte in the built-in data EEPROM at *Address* and return its value into *Variable*. This instruction does not work with the built-in EEPROM of PIC12CExx parts.

(continued)

TABLE 26 PicBasic Built In Functions *(Continued)*

INSTRUCTION	DESCRIPTION
RESUME {Label}	Restore execution at the instruction after the *ON DEBUG* or *ON INTERRUPT* instruction handler was executed. If a *Label* is specified then the hardware is returned to its original state and execute jumps to the code after *Label.*
RETURN	Return to the instruction after the "GOSUB" instruction.
REVERSE Pin	Reverse the function of the specified *Pin.* For example, if it were in Output mode, it is changed to Input mode.
SERIN Pin, Mode, {Timeout, Label,} {Qual, . . . } [Variable {,Variable . . . }]	Receive one or more asynchronous data bytes on *Pin.* The *Pin* can be defined at run time. The *Qual* bytes are test qualifiers that only pass following bytes when the first byte of the incoming string match them. The *Timeout* value is in msecs and execution jumps to *Label* when the *Timeout* interval passes without any data being received. *Mode* is used to specify the operation of the *Pin* and is defined in the table:

Mode	Baud Rate	State
T300	300	Positive
T1200	1200	Positive
T2400	2400	Positive
T9600	9600	Positive
N300	300	Negative
N1200	1200	Negative
N2400	2400	Negative
N9600	9600	Negative

INSTRUCTION	DESCRIPTION
SERIN2 Pin {\FlowPin}, Mode, {ParityLabel,} {Timeout, Label,} [Specification]	Receive one or more asynchronous data bytes on *Pin. FlowPin* is used to control the input of data to the PICmicro® MCU to make sure there is no overrun. If *Even Parity* is selected in the *Mode* parameter, then any time an invalid byte is received, execution will jump to the *ParityLabel.* Input timeouts can be specified in 1-ms intervals with no data received in the specified period causing execution to jump to *Label. Mode* selection is made by passing a 16-bit variable to the SERIN2 instruction. The bits are defined as:

Bit	Function
15	Unused
14	Set if input data is negative
13	Set if even parity is to be used with the data
12–0	Data rate specification, found by the formula:
	Rate = (1,000,000/Baud) - 20

The *Specification is* a string of data qualifiers /modifiers and destination variables that are used to filter and process the incoming data. The qualifiers/modifiers are listed:

Modifier	Operation
Bin {1 . . . 16} Var	Receive up to 16 Binary digits and store in *Var*
DEC {1 . . . 5} Var	Receive up to 5 Decimal digits and store in *Var*
HEX {1 . . . 4} Var	Receive up to 4 Hexadecimal digits and store in *Var*
SKIP #	Skip # received characters
STR Array\n\c	Receive a string of *n* characters and store in *Array.* Optionally ended by character c.
WAIT (“String”)	Wait for the specified *String* of characters.
WAITSTR Array\n	Wait for a character string *n* characters long.

SEROUT Pin, Mode, [Value{,Value . . . }]

Send one or more asynchronous data bytes on *Pin*. The *Pin* can be defined at run time. *Mode* is used to specify the operation of the *Pin* and the output driver and is defined in the table:

Mode	Baud Rate	State	Driver
T300	300	Positive	CMOS
T1200	1200	Positive	CMOS
T2400	2400	Positive	CMOS
T9600	9600	Positive	CMOS

(continued)

TABLE 26 PicBasic Built In Functions *(Continued)*

INSTRUCTION	DESCRIPTION			
	N300	300	Negative	CMOS
	N1200	1200	Negative	CMOS
	N2400	2400	Negative	CMOS
	N9600	9600	Negative	CMOS
	OT300	300	Positive	Open-Drain
	OT1200	1200	Positive	Open-Drain
	OT2400	2400	Positive	Open-Drain
	OT9600	9600	Positive	Open-Drain
	ON300	300	Negative	Open-Drain
	ON1200	1200	Negative	Open-Drain
	ON2400	2400	Negative	Open-Drain
	ON9600	9600	Negative	Open-Drain
SEROUT2 Pin {\FlowPin}, Mode, {Pace,} {Timeout, Label,} [Specification]	Send one or more asynchronous data bytes on *Pin. FlowPin* is used to control the output of data to the PICmicro® MCU to make sure there is no overrun. Timeouts can be specified in 1-ms intervals with no *Flow* control on the receiver the specified period causing execution to jump to *Label.* The optional *Pace* parameter is used to specify the length of time (measured in μs) that the PICmicro® MCU delays before sending out the next character. *Mode* selection is made by passing a 16-bit variable to the SERIN2 instruction. The bits are defined as:			
	Bit	Function		
	15	CMOS/Open drain driver specification. If set, open drain output		
	14	Set if input data is negative		
	13	Set if even parity is to be used with the data		
	12–0	Data rate specification, found by the formula: Rate = (1,000,000/Baud) − 20		

Command	Description
	The *Specification* is a string of data qualifiers/ modifiers and source values that are used to format the outgoing data. The output format data can be specified with an *I* prefix to indicate that the data type is to be sent before the data and the *S* prefix indicates that a sign (−) indicator is sent for negative values. The qualifiers/modifiers are listed in the table:

Modifier	Operation
Bin{1 . . . 16} Var	Receive up to 16 binary digits and store in *Var*
DEC{1 . . . 5} Var	Receive up to 5 decimal digits and store in *Var*
HEX{1 . . . 4} Var	Receive up to 4 hexadecimal digits and store in Var
SKIP #	Skip *#* received characters
STR Array\n\c	Receive a string of *n* characters and store in *Array.* Optionally ended by character c
WAIT ("String")	Wait for the specified *String* of characters
WAITSTR Array\n	Wait for a character string *n* characters long

Command	Description
SHIFTIN DataPin, ClockPin, Mode, [Variable {\Bits} {,Variable . . . }]	Synchronously shift data into the PICmicro® MCU.The *Bits* parameter is used to specify the number of bits that are actually shifted in (if *Bits* is not specified, the default is 8). The *Mode* parameter is used to indicate how the data is to be transferred and the values are listed in the table.

Mode	Function
MSBPRE	Most-significant bit first, Read data before pulsing clock
LSBPRE	Least-significant bit first, read data before pulsing clock
MSBPOST	Most-significant bit first, read data after pulsing clock
LSBPOST	Least-significant bit first, read data after pulsing clock

Command	Description
SHIFTOUT DataPin, Clock Pin, Mode, [Variable {\Bits} {,Variable . . . }]	Synchronously shift data out of the PICmicro® MCU. The *Bits* parameter is used to specify how many bits are to be shifted out in each word (if not specified, the default is 8). The *Mode* parameter is used to specify how the data is to be shifted out and the values are listed in the table:

(continued)

TABLE 26 PicBasic Built In Functions *(Continued)*

INSTRUCTION	DESCRIPTION
	Mode — Function LSBFIRST — Least-Significant bit first MSBFIRST — Most-Significant bit first
SLEEP Period	Put the PICmicro® MCU into *Sleep* mode for *Period* seconds.
SOUND Pin, [Note, Duration {,Note, Duration . . . }]	Output a string of tones and durations (which can be used to create a simple tune) on the *Pin.* Note *0* is silence and Notes *128* to *255* are “white noise.” Note *1* (78.5 Hz for a 4-MHz PICmicro® MCU) is the lowest valid tone and note *127* is the highest (10 kHz in a 4-MHz PICmicro® MCU). Duration is specified in 12-ms increments
STOP	Place the PICmicro® MCU into an endless loop. The PICmicro® MCU is not put into *Sleep* mode.
SWAP Variable, Variable TOGGLE Pin	Exchange the values in the two variables. Toggle the output value of the specified pin.
WHILE Cond :WEND	Execute the code between the *WHILE* and the *WEND* statements while the Cond condition returns a nonzero value. Execution exits the loop when *Cond* is evaluated to zero.
WRITE Address, Value	Write the byte value into the built-in data EEPROM. This instruction will not work with the built-in EEPROM in the PIC12CExxx devices.
XIN DataPin, ZeroPin, {Timeout, Label,} [Variable {,Variable . . . }]	Receive data from X–10 devices. *ZeroPin* is used to detect the *Zero Crossing* of the input AC signal. Both *DataPin* and *ZeroPin* should be pulled up with 4.7-Ω resistors. The optional *Timeout* (specified in 8.33-ms intervals) will cause execution to jump to *Label* if no data is received by the specified interval. If the first *Variable* data destination is 16 bits, then both the *House Code* and the *key Code* will be saved. If the first *Variable* is eight bits in size, then only the *Key Code* will be saved.
XOUT DataPin, ZeroPin, [House Code\KeyCode {\Repeat} {,Value . . . }]	Send X–10 data to other devices. The *ZeroPin* is an input and should be pulled up with a 4.7-Ω resistor. *HouseCode* is a number between 0 and 15 and corresponds to the *House Code* set on the X–10 Modules A through P. The *KeyCode* can either be the number of a specific X–10 receiver or the function to be performed by the module.

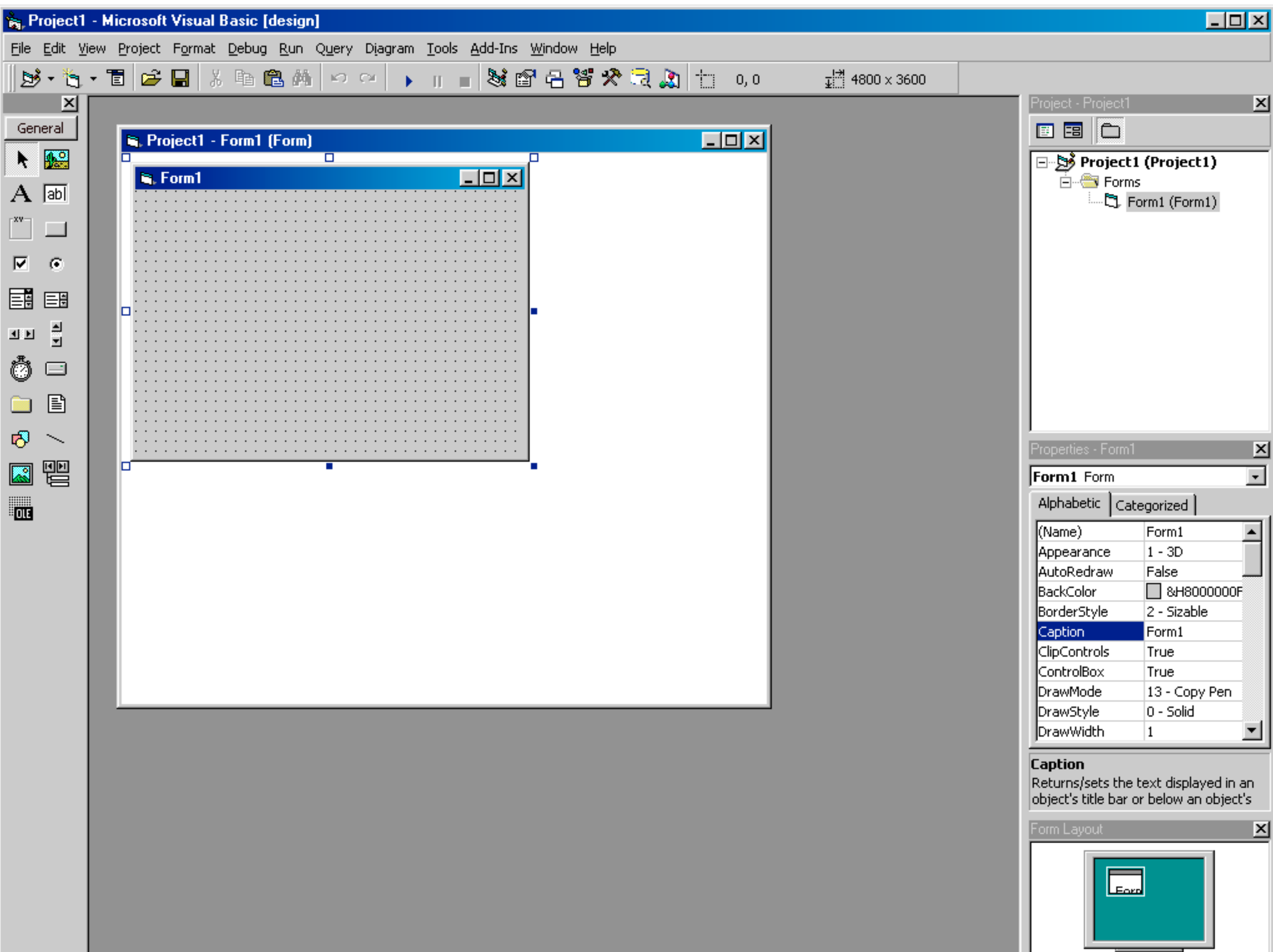

Figure 11 Visual Basic development system

Visual Basic applications are built around The Dialog Box Editor desktop. When application development is started, Visual Basic provides you with the initial dialog box of your application (Fig. 12).

From here, *Dialog Resources* are selected from the *ToolBox* and placed upon the dialog.

Once the dialog box is designed and looks the way you want it to, code development can begin. The easiest way to create the application code is by double clicking on the control in the *Dialog Box Editor* and a subroutine prototype will appear and allows code to be added. In Fig. 13, I have clicked on the *Quit* button to prompt Visual Basic to produce the:

```
Private Sub Command2_Click()
End Sub
```

code. Once this was done, I entered the comment and statement:

```
' "Quit" CommandButton

  End
```

Control attributes (also known as *properties*) can be set globally from the integrated development environment or from within the event handlers. The event handler's code is written in pretty standard Microsoft BASIC. Once the handler prototypes are created by Visual Basic, it is up to the application developer to add the response code for the application. Visual Basic provides a large number of built-in functions, including trigonometry, logarithms, and the ability to interface with the file system and dialog controls.

Variables in Visual Basic are typically "integer," which is to say they are 16 bit values in the ranges −32768 to +32767. Thirty-two bit integer variables can be specified by declaring the variable with a "Dim" statement and type "Long." One important thing about variables is that they are local to the event routine they are used in, unless they are declared globally in the General Module, which executes at the beginning of the application and is not specific to any controls.

A number of controls are basic to Visual Basic. Others are available for downloading or purchasing off the Internet. These can make your Visual Basic applications very impressive and add a lot of "pizzazz." The default controls are listed in Table 27.

A number of controls cannot be activated with a left button click and cannot be "seen" on the application's form. The one that is used the most is the *Timer*. This control causes an event after a set period of microseconds. This control can be set within the dialog editor or modified within the application itself. The *Timer* can provide many different advanced functions without requiring any interrupt interfaces.

THE C LANGUAGE

For modern systems, C is the programming language of choice because it is available for a wide range of systems and processors (including the PICmicro® MCU). This ubiquity requires anyone who is planning on developing application for processing systems to have at least a passing knowledge of the language. C is often referred to as the "universal assembly language" because it is designed in such a way that it can access a system's lowest levels efficiently.

The book presents "pseudo-code" in C format to help explain the operation of the hardware without getting bogged down in the details of assembly language.

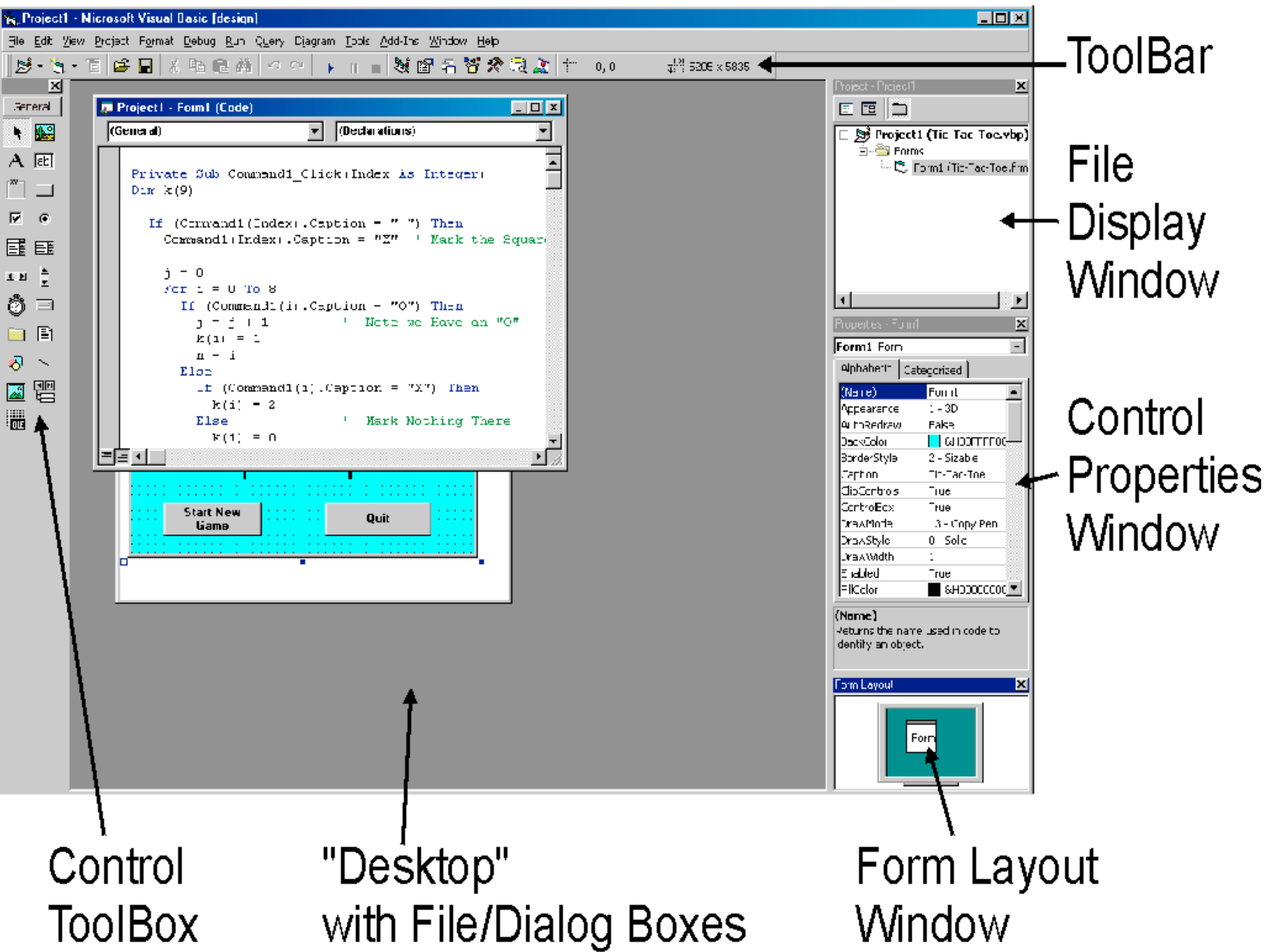

Figure 12 **Visual Basic desktop**

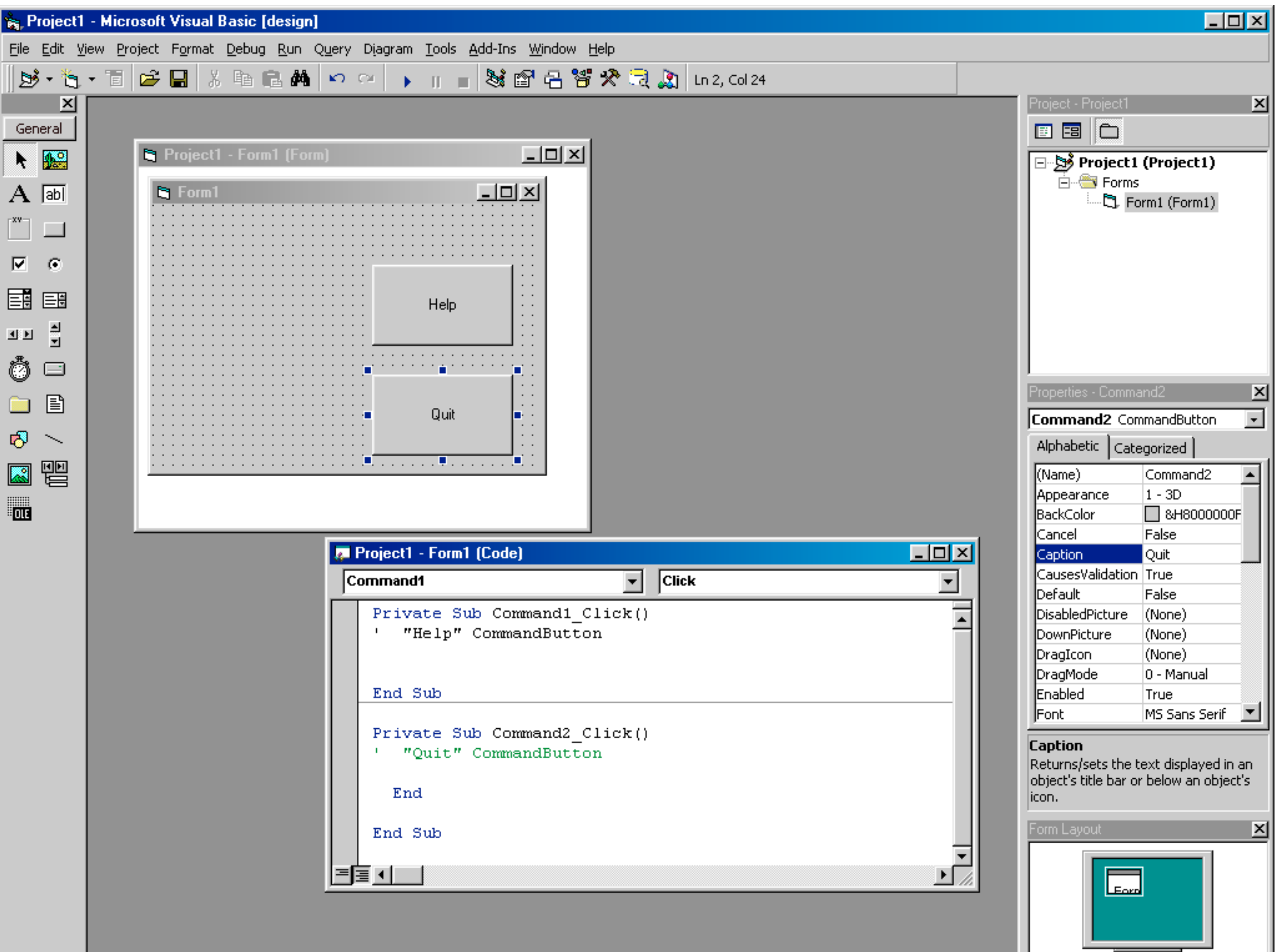

Figure 13 Visual Basic dialog box control set up

TABLE 27 Visual Basic "Controls"

CONTROL	DESCRIPTION
Pull Downs	Selected from the "Menu Editor" icon on the "ToolBar"
PictureBox	Display Bitmaps and other graphic files on the Dialog Box
Label	Put Text in the Dialog Box
TextBox	Input/Output Text Box
Frame	Put a Frame around Resources
CommandButton	Button for Code Operation
CheckBox	For Checking Multiple Selections
OptionButton	Also known as the "Radio Button". For Checking one selection for a list of Multiple options
ComboBox	Select or Enter Test in a Box/List
ListBox	List Data (with User controlled Scrolling)
HScrollBar	Provide Horizontal Scrolling in a Text or Graphic Output Control
VscrollBar	Provide Vertical Scrolling in a Text or Graphic Output Control
Timer	Cause a periodic interrupt
DriveListBox	Select a Drive for File I/O
DirListBox	Select a Subdirectory for File I/O on a Specific Drive
FileListBox	Display Files for a Specific Subdirectory on a Specific Drive
Shape	Put a Graphics Shape on the Dialog Box
Line	Draw a Line on the Dialog Box
Image	Display an Image File on the Dialog Box
OLE	Insert OLE Objects to the Dialog

Pseudo-code is a term for code which is designed to illustrate the operation of an algorithm or hardware interfaces. My choice of using C format means that you should have at least a passing understanding of the language. This section will quickly give you an understanding of how C is designed and how statements are defined.

Throughout the book, the code examples will either in assembly language or C-styled assembly language.

Declarations Constant declaration:

```
const int Label = Value;
```

Variable declaration:

```
type Label [= Value];
```

Value is an optional initialization constant. Where *type* is shown in Table 28.

Notice that *int* is defined as the "word size" of the processor/operating system. For PCs, an *int* can be a word (16 Bits) or a double word (32 bits).

There might also be other basic types defined in the language implementation. Single-dimensional arrays are declared using the form:

```
type Label[ Size ] [={ Initialization Values..}];
```

Notice that the array *Size* is enclosed within square brackets (*[* and *]*) and should not be confused with the optional *Initialization Values*.

Strings are defined as single-dimensional ASCIIZ arrays:

```
char String[ 17 ] = "This is a String";
```

where the last character is an ASCII NUL.

Strings can also be defined as pointers to characters:

```
char *String = "This is a String";
```

For the PICmicro® MCU and other Harvard-architected processors, the text data could be written into data space when the application first starts up as part of the language's initialization.

Multidimensional arrays are defined with each dimension separately identified within square brackets (*[* and *]*):

```
int ThreeDSpace[ 32 ][ 32 ][ 32 ];
```

Array dimensions must be specified unless the variable is a pointer to a single-dimensional array. Pointers are declared with the * character after the *type*:

```
char * String = "This is a String";
```

Accessing the address of the pointer in memory is accomplished using the *&* character:

```
StringAddr = &String;
```

Accessing the address of a specific element in a string is accomplished using the *&* character and a string array element:

```
StringStart = &String[ n ];
```

TABLE 28 "C" Data Types	
char	long
int	
unsigned int	
float	

In the PC, I recommended that "far" (32 bit) pointers be always used with *absolute offset:segment* addresses within the PC memory space to avoid problems with changing segments due to different "memory models."

The variable's type can be overridden by placing the new type in front of the variable in brackets:

```
(long) StringAddr = 0x0123450000;
```

Statements Application start:

```
main(envp)
   char *envp;
{  // Application Code

   :                                  // Application Code

}  // End Application
```

Function format:

```
Return_Type Function( Type Parameter [, Type Parameter..])
{ // Function Start

  :                                   // Function Code

   return value;

} // End Function
```

Function prototype:

```
Return_Type Function( Type Parameter [, Type Parameter..]);
```

Expression:

```
[(..] Variable | Constant [Operator [(..] Variable | Constant ][)..]]
```

Assignment Statement:

```
Variable = Expression;
```

C conditional statements consist of *if, ?, while, do, for,* and *switch*. The *if* statement is defined as:

```
  if ( Statement )
; | { Assignment Statement | Conditional Statement.. } | Assignment
Statement | Conditional Statement
   [else ;| { Assignment Statement | Conditional Statement..} |
Assignment Statement | Conditional Statement ]
```

The *? :* statement evaluates the statement (normally, a comparison) and if not equal to zero, execute the first statement, otherwise execute the statement after the *:*

```
   Statement ? Assignment Statement | Conditional Statement : Assignment
Statement | Conditional Statement
```

The *while* statement is added to the application following the definition:

```
   while ( Statement ) ; | { Assignment Statement | Conditional
Statement.. } | Assignment Statement | Conditional Statement
```

The *for* statement is defined as:

```
   for ( initialization (Assignment) Statement; Conditional Statement;
Loop Expression (Increment) Statement )
; | { Assignment Statement | Conditional Statement.. } | Assignment
Statement | Conditional Statement
```

To jump out of a currently executing loop, *break* statement:

```
break;
```

is used.

The *continue* statement skips over remaining code in a loop and jumps directly to the loop condition (for use with *while*, *for*, and *do/while* loops). The format of the statement is:

```
continue;
```

for looping until a condition is true, the *do/while* statement is used:

```
do
   Assignment Statement | Conditional Statement..
while ( Expression );
```

to conditionally execute according to a value, the *switch* statement is used:

```
switch( Expression ) {
  case Value:                    // Execute if "Statement" == "Value"
    [ Assignment Statement | Conditional Statement.. ]
    [break;]
  default:                       // If no "case" Statements are True
    [ Assignment Statement | Conditional Statement.. ]
}   // End switch
```

Finally, the *goto Label* statement is used to jump to a specific address:

```
  goto Label;
Label:
```

To return a value from a function, the *return* statement is used:

```
return Statement;
```

Operators Various operators are listed in Tables 29 through 31.

Directives All directives start with # and are executed before the code is compiled (Table 32). The words in Table 33 cannot be used in C applications as labels.

<table>
<tr><th colspan="2">TABLE 29 “C” Statement Operators</th></tr>
<tr><th>OPERATOR</th><th>OPERATION</th></tr>
<tr><td>!</td><td>Logical Negation</td></tr>
<tr><td>^</td><td>Bitwise Negation</td></tr>
<tr><td>&&</td><td>Logical AND</td></tr>
<tr><td>&</td><td>Bitwise AND, Address</td></tr>
<tr><td>||</td><td>Logical OR</td></tr>
<tr><td>|</td><td>Bitwise OR</td></tr>
<tr><td>^</td><td>Bitwise XOR</td></tr>
<tr><td>+</td><td>Addition</td></tr>
<tr><td>++</td><td>Increment</td></tr>
<tr><td>-</td><td>Subtraction, Negation</td></tr>
<tr><td>--</td><td>Decrement</td></tr>
<tr><td>*</td><td>Multiplication, Indirection</td></tr>
<tr><td>/</td><td>Division</td></tr>
<tr><td>%</td><td>Modulus</td></tr>
<tr><td>==</td><td>Equals</td></tr>
<tr><td>!=</td><td>Not Equals</td></tr>
<tr><td><</td><td>Less Than</td></tr>
<tr><td><=</td><td>Less Than or Equals To</td></tr>
<tr><td><<</td><td>Shift Left</td></tr>
<tr><td>></td><td>Greater Than</td></tr>
<tr><td>>=</td><td>Greater Than or Equals To</td></tr>
<tr><td>>></td><td>Shift Right</td></tr>
</table>

TABLE 30 "C" Compound Assignment Statement Operators

OPERATOR	OPERATION
&=	AND with the Variable and Store Result in the Variable
!=	OR with the Variable and Store Result in the Variable
^=	XOR with the Variable and Store Result in the Variable
+=	Add to the Variable
-=	Subtract from the Variable
*=	Multiply to the Variable
/=	Divide from the Variable
%=	Get the Modulus and Store in the Variable
<<=	Shift Left and Store in the Variable
>>=	Shift Right and Store in the Variable

TABLE 31 "C" Operator Order Of Operations

<table>
<tr><th>OPERATORS</th><th>PRIORITY</th><th>TYPE</th></tr>
<tr><td>( ) [ ] . -></td><td>Highest</td><td>Expression Evaluation</td></tr>
<tr><td>- ~ ! & * ++ --</td><td></td><td>Unary Operators</td></tr>
<tr><td>* / %</td><td></td><td>Multiplicative</td></tr>
<tr><td>+ -</td><td></td><td>Additive</td></tr>
<tr><td><< >></td><td></td><td>Shifting</td></tr>
<tr><td>< <= >= ></td><td></td><td>Comparison</td></tr>
<tr><td>== ! =</td><td></td><td>Comparison</td></tr>
<tr><td>&</td><td></td><td>Bitwise AND</td></tr>
<tr><td>^</td><td></td><td>Bitwise XOR</td></tr>
<tr><td>|</td><td></td><td>Bitwise OR</td></tr>
<tr><td>&&</td><td></td><td>Logical AND</td></tr>
<tr><td>||</td><td></td><td>Logical OR</td></tr>
<tr><td>?:</td><td></td><td>Conditional Execution</td></tr>
<tr><td>= &= |= ^= += -= *= /= %= >>=
<< =</td><td></td><td>Assignments</td></tr>
<tr><td>,</td><td>Lowest</td><td>Sequential Evaluation</td></tr>
</table>

TABLE E-32 "C" Directives

DIRECTIVE	FUNCTION
#define Label [(Parameters)] Text	Define a *Label* that will be replaced with *Text* when it is found in the code. If *Parameters* are specified, then replace them in the code, similar to a macro.
#undefine Label	Erase the defined *Label* and *Text* in memory.
#include "File" | <File>	Load the Specified File in line to the *Text.* When < > encloses the filename, then the file is found using the *INCLUDE* environment path Variable. If " " encloses the filename, then the file in the current directory is searched before checking the *INCLUDE* path.
#error Text	Force the error listed in *Text*
#if Condition	If the *Condition* is true, then Compile the following code to *#elif*, *#else*, or *#endif.* If the *Condition* is false, then ignore the following code to *#elif, #else,* or *#endif*.
#ifdef Label	If the *#define* label exists, then Compile the following code. *#elif, #else,* and *#endif* work as expected with *#if*.
#ifndef Label	If the *#define* label does not exist, then compile the following code. *#elif, #else,* and *#endif* work as expected with *#if*.
#elif Condition	This directive works as an *#else #if* to avoid lengthy nested *#if* s. If the previous condition was false, checks the condition.
#else	Placed after *#if* or *#elif* and toggles the current compile condition. If the current compile condition was false, after *#else*, it will be true. If the current compile condition was true, after *#else,* it will be false.
#endif	Used to end an *#if, #elif,* #else, #ifdef, or *#ifndef* directive.
#pragma String	This is a compiler dependent directive with different strings required for different situations.

TABLE 33 "C" Reserved Words
break
case
continue
default
do
else
for
goto
if
return
switch
while

Backslash characters Table 34 shows a list of C backslash characters cross referenced to ASCII codes.

Common C functions Common C functions as defined by Kernighan and Ritchie are given in Table 35.

TABLE 34 "C" Backslash Characters to ASCII Codes Cross Reference

STRING	ASCII	CHARACTER
\r	0x00D	Carriage Return ("CR")
\n	0x00A	Line Feed ("LF")
\f	0x00C	Form Feed ("FF")
\b	0x008	Backspace ("BS")
\t	0x009	Horizontal Tab ("HT")
\v	0x00B	Vertical Tab ("VT")
\a	0x007	Bell ("BEL")
\'	0x027	Single Quote ("'")
\"	0x022	Double Quote (""")
\\	0x05C	Backslash ("\")
\ddd	N/A	Octal Number
\xddd	0x0dd	Hexadecimal Character

TABLE 35 Standard "C" Built in Functions

FUNCTION	OPERATION
int getchar (void)	Get one Character from "Standard Input" (the Keyboard). If no Character available, then wait for it.
int putchar (int)	Output one Character to the "Standard Output" (the Screen).
int printf (char *Const [, arg . . .])	Output the "Const" String Text. "Escape Sequence" Characters for Output are embedded in the "Const" String Text. Different Data Outputs are defined using the "Conversion Characters": %d, %i – Decimal Integer %o – Octal Integer %x, %X – Hex Integer (with upper or lower case values). No leading "0x" character String Output %u – Unsigned Integer %c – Single ASCII Character %s – ASCIIZ String %f – Floating Point %#e, %#E – Floating Point with the precision specified by "#" %g, %G – Floating Point %p – Pointer %% – Print "%" Character Different C Implementations will have different "printf" parameters.
int scanf (char *Const, arg [, *arg. . .])	Provide Formatted Input from the user. The "Const" ASCIIZ String is used as a "Prompt" for the user. Note that the input parameters are always pointers. "Conversion Characters" are similar to "printf": %d – Decimal Integer %i – Integer. In Octal if leading "0" or hex if leading "0x" or "0X" %o – Octal Integer (Leading "0" Not Required) %x – Hex Integer (Leading "0x" or "0X" Not Required) %c – Single Character %s – ASCIIZ String of Characters. When Saved, a NULL character is put at the end of the String %e, %f, %g – Floating Point Value with optional sign, decimal point and exponent %% – Display "%" character in prompt

(continued)

TABLE 35 Standard "C" Built in Functions *(Continued)*

FUNCTION	OPERATION
handle fopen (char *mode)	Open File and Return Handle (or NULL for *FileName, Error). "mode" is a String consisting of the optional characters: r – Open File for Reading w – Open File for Writing a – Open File for Appending to Existing Files Some systems handle "Text" and "Binary" files. A "Text" file has the CR/LF characters represented as a single CR. A "Binary" file does not delete any characters.
int fclose (handle)	Close the File.
int getc (handle)	Receive data from a file one character at a time. If at the end of an input file, then "EOF" is returned.
int putc (handle, char)	Output data to a file one character at a time. Error is indicated by "EOF" returned.
int fprintf (handle, char *Const [, arg . . .])	Output String of Information to a File. The same "Conversion Characters" and arguments as "printf" are used.
int fscanf (handle, char *Const, arg [, arg . . .])	Input and Process String of Information from a File. The same "Conversion Characters" and arguments as "scanf" are used.
int fgets (char *Line, int LineLength, handle)	Get the ASCIIZ String from the file.
int fputs (char *line, handle)	Output an ASCIIZ String to a file.
strcat (Old, Append)	Put ASCIIZ "Append" String on the end of the "Old" ASCIIZ String.
strncat (Old, Append, #)	Put "#" of characters from "Append" on the end of the "Old" ASCIIZ String.
int strcmp (String1, String2)	Compare two ASCIIZ Strings. Zero is returned for match, negative for "String1" < "String2" and positive for "String1" > "String2".
int strncmp (String1, String2, #)	Compare two ASCIIZ Strings for "#" characters. Zero is returned for match, negative for "String1" < "String2" and positive for "String1" > "String2".
strcpy (String1, String2)	Copy the Contents of ASCIIZ "String2" into "String1".
strncpy (String1, Strint2, #)	Copy "#" Characters from "String2" into "String1".

TABLE 35 Standard "C" Built in Functions *(Continued)*

FUNCTION	OPERATION
strlen (String)	Return the length of ASCIIZ Character "String".
int strchr (String, char)	Return the Position of the first "char" in the ASCIIZ "String".
int strrchr (String, char)	Return the Position of the last "char" in the ASCIIZ "String".
system (String)	Executes the System Command "String".
*malloc (size)	Allocate the Specified Number of Bytes of Memory. If insufficient space available, return NUL.
*calloc (#, size)	Allocate Memory for the specified "#" of data elements of "size".
free (*)	Free the Memory.
float sin (angle)	Find the "Sine" of the "angle" (which in Radians).
float cos (angle)	Find the "Cosine" of the "angle" (which in Radians).
float atan2 (y, x)	Find the "Arctangent" of the "X" and "Y" in Radians.
float exp (x)	Calculate the natural exponent.
float log (x)	Calculate the natural logarithm.
float log10 (x)	Calculate the base 10 logarithm.
float pow (x, y)	Calculate "x" to the power "y".
float sqrt (x)	Calculate the Square Root of "x".
float fabs (x)	Calculate the Absolute Value of "x".
float frand ( )	Get a Random Number.
int isalpha (char)	Return Non-Zero if Character is "a"–"z" or "A"–"Z".
int isupper (char)	Return Non-Zero if Character is "A"–"Z".
int islower (char)	Return Non-Zero if Character is "a"–"z".
int isdigit (char)	Return Non-Zero if Character is "0"–"9".
int isalnum (char)	Return Non-Zero if Character is "a"–"z", "A"–"Z" or "0"–"9".
int isspace (char)	Return Non-Zero if Character is " ", HT, LF, CR, FF or VT.
int toupper (char)	Convert the Character to Upper Case.
int tolower (char)	Convert the Character to Lower Case.

TABLE 36 Functions Added To Standard "C" For The PICmicro® MCU

FUNCTION	OPERATION
inp, outp	Provide method for directly accessing system registers.
SerIn, SerOut	Serial I/O functions.

PICmicro® MCU enhancement functions Useful Functions in PICmicro® MCU C implementations are shown in Table 36.

Assembly Language Programming

Assembly-language programming is the process of creating a series of machine instructions (which are normally referred to as just *instructions*), which will perform a useful task. Assembly-language programming is often perceived as the most difficult method of programming and also the most efficient.

I have to disagree with both of these statements, once you are familiar with a processor architecture and its instructions assembly-language programming is not that much more difficult that high-level language programming. As for the efficiency issue, if an application is not properly designed or the programmer doesn't know how to take advantage of the architecture, it doesn't make any difference how it is programmed. Assembly language makes it very difficult to implement efficiently.

In writing about assembly language, I always have to remember *not* to put down things like "visualize data flows" or "mentally twist the requirements to optimize the application."

Although I heartily believe that getting into the right "head space" for writing assembly-language code is necessary, I hope I have not left you with the impression that I write code in a darkened room, with sitar music playing on an eight-track and incense burning in the air.

Just to set the record straight, I converted to CDs years ago.

Although also wanting you to work at achieving the proper mental perspective, I want to also emphasize that assembly-language programming should be initially approached from a structured-language perspective. The application code should be designed using the same basic rules and concepts that I have presented in the previous sections of this appendix. By using these concepts, an application can be designed that is based on these concepts by simply converting these concepts into the appropriate assembly-language statements. As you become more familiar with the processor you are working with, you can look for opportunities to optimize the your application code by taking advantage of the processor's architecture and avoiding redundant instructions or using advanced instructions.

Writing assembly-language applications based on the basic concepts presented earlier in this appendix requires a fairly complete understanding of the processor and how the different instructions execute in it. This book provides two different references to the PICmicro® MCU's instructions, one which is a simple one- to four-line pseudo-code representation of how the instruction executes. The second tool presented is a block diagram of the processor with the datapaths used by the instruction marked on it. The PICmicro® MCU is well suited to the graphical method of application software depiction. With the small instruction set, keeping this visual tool handy is not unreasonable. As you gain more experience with the PICmicro® MCU, you will rely on these tools less and less.

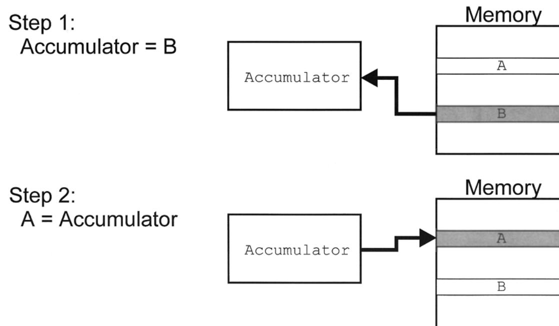

Figure 14 **Assignment statement operation**

The basic operation of any processor is moving data. In some processors, this can be accomplished in one step without any intermediate stops for the data. For most processors, this isn't available and data has to be stored temporarily in an accumulator (known as the *w register* in the PICmicro® MCU).

To implement the statement:

```
A = B
```

The data will first take the path from memory into the accumulator and then from the accumulator to the destination. This is shown graphically in Fig. 14.

And would be implemented using the two pseudo-instructions:

```
MovAcc B                ; Accumulator = B
StorAcc A               ; Accumulator = A
```

The accumulator is a register used to temporarily save the results of a move or arithmetic or bitwise operation. Some processors (such as the Intel 8086 used in the PC) have multiple registers available for this function, but many (including the PICmicro® MCU) do not. For processors that only have one accumulator, you will have to keep saving the contents of the accumulator before overwriting it. This is known as a *context register* and is very important for handling interrupts.

Five basic addressing modes are available in most processors. When a register is loaded with a constant (or a literal), the term used is *immediate addressing*. When data is passed between registers (and not memory), it is known as *register addressing* or *implicit addressing*. Loading a register or saving its contents to a variable is known as *direct addressing*. Array variable accesses use *array addressing* or *indirect addressing* while passing data to and from the stack is known as *stack addressing*. Each of these different modes is best suited for different applications.

Table 37 lists the different addressing modes and how they work.

The PICmicro® MCU only has the Immediate, Direct, and Index" data addressing modes available to it. Chapter 3 covers how these addressing modes work with the specific PICmicro® MCU registers and instructions in more detail. Elsewhere, the book shows how data stacks can be simulated in the PICmicro® MCU.

The accumulator is used to store the results from arithmetic and bitwise operations. For example, implementing the arithmetic assignment statement:

```
A = B + 47
```

TABLE 37 Assembly Language Addressing Modes

MODE	EXAMPLE	COMMENTS
Immediate	Register = Constant	Place a constant in a register. Cannot write to a constant.
Register	RegisterA = RegisterB	Pass data between Registers without affecting memory.
Direct	Memory = Register | Constant	Pass the contents of a Register or a Constant to a memory location.
Array	Register = Array [Index]	Pass data to and from an Array Variable using a specific index.
Stack	Register = Array [Index] Index = Index + 1	Store Data on a "LIFO" Memory.

using a fictional processor (which has an ALU to process data), this statement could operate as shown in Fig. 15.

During bitwise and arithmetic operations, the STATUS register is updated with the results of the operation. Normally, three flags are used to record the results of the arithmetic operation. These flags are used to respond to the conditions of an operation and should be saved when execution temporarily changes (during an interrupt or subroutine call). This makes STATUS one of the *context registers*, along with the accumulator.

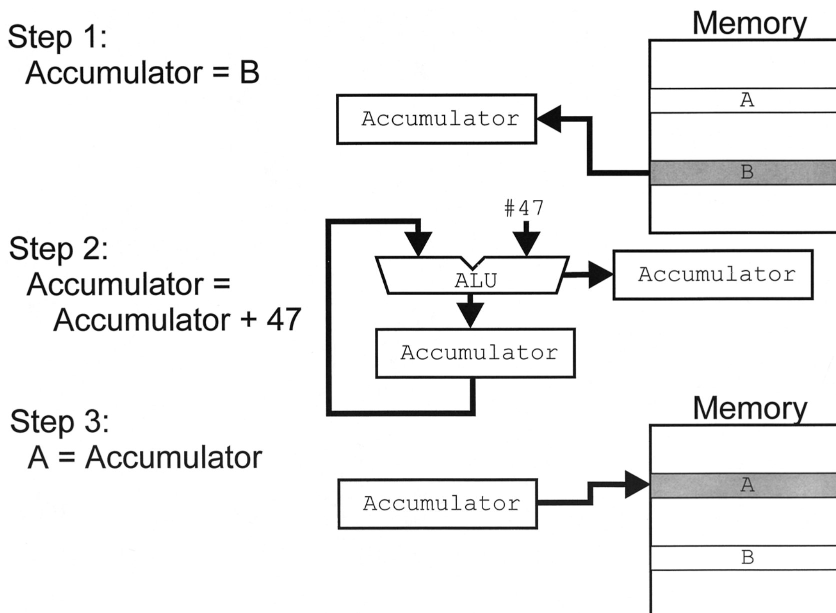

Figure 15 Expression statement operation

The Zero bit is probably the flag you will use the most often when you start developing your own PICmicro® MCU assembly-language applications. This bit, or flag is set when the results are equal to zero. For example, using the code:

```
MovAcc #47              ; Accumulator = 0x02F
AndAcc #15              ; Accumulator = Accumulator & 0x00F
                        ;             = 0x02F & 0x00F
                        ;             = 0x00F
                        ; Zero Flag is Reset

MovAcc #47              ; Accumulator = 0x02F
AddAcc #0x0D1           ; Accumulator = Accumulator + -47
                        ;             = 0x02F + 0x0D1
                        ;             = 0x0100 (actually 0x000 Stored)
                        ; Zero Flag is Set
```

In the first example, the result is not equal to zero so that the Zero bit in the STATUS register is loaded with *0* or is reset. In the second example, when a number is added to the negative of itself the result stored in the accumulator is 0x000. The ninth bit of the result is not stored in the accumulator, but the Carry flag. In this case, the Zero bit is loaded with a *1* or is set. To summarize, the Zero bit is set if the result loaded back into the accumulator is equal to zero.

In the second example, 0x0D1 is the two's complement representation of -47, as explained earlier in this appendix. When this is added to the positive value, the result is 256 decimal or 0x0100. Because only the least-significant eight bits of the result are saved, the accumulator is loaded with zero and the Zero bit of the STATUS register is set.

Although the Zero flag will be used as you start assembly-language programming, you will find that the Carry bit of the STATUS register will become more useful. Carry is generated (and used) by addition and subtraction, as well as bit-shift instructions. Carry is also often used as an Error flag for subroutines in many applications used in processors other than the PICmicro® MCU. The Carry flag was used in the IBM PC for returning the status of BIOS functions and many people continue to use this convention. The PICmicro® MCU is particularly adept at handling and testing for bit conditions. This ability makes the use of the Carry flag for returning error information redundant and limits the usefulness of the Carry flag.

For addition and subtraction, the Carry flag is set when the result overflows to the next highest byte. In the previous example pseudo-assembler code, the result stored in the accumulator of 47 and −47 was zero, but the actual result was 0x0100. In this example, after the *AddAcc* instruction, both the Zero and Carry bits of the STATUS register will be set.

The Carry flag is often used for loading in and storing data during *shift* and *rotate* instructions. These instructions are often confused because they are very similar, as can be seen in Figs. 16 and 17. The differences lie in whether or not the most-significant bit is shifted out or is used as an input to the shift.

This probably makes it more confusing, but the two cases can be shown graphically, the first is the shift, which looks like Figure 16:

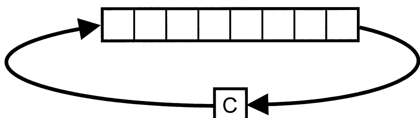

Figure 16 **Data shift operation**

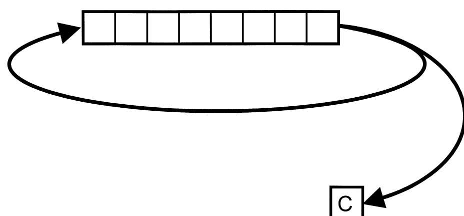

Figure 17 Data rotate operation

I tend to call this operation a *shift* because data is leaving the shifted register and new data is being loaded in. A *rotate* operation is shown in Fig. 17.

In the rotate case, data is never lost. Shift instructions can be used for a variety of tasks and can help out in making an application more efficient. Throughout the book, I use the PICmicro® MCU's shift instructions (which are called *Rotate* instructions by Microchip) for enhancing the application's execution in ways that I think you will find surprising.

The Digit Carry flag is similar to the Carry flag, except that it works at the least-significant nybble level instead of at the least-significant byte level. The Digit Carry (also known as the *Auxiliary flag* in some processors) is set after an addition or subtraction operation in which the lower nybble affects the upper nybble of the result. The name *Digit Carry* comes from the idea that each byte is broken up into two hex digits.

An example of how the Digit Carry works is:

```
MovAcc #10          ; Accumulator = 0x00A
AddAcc #0x009       ; Accumulator = Accumulator + 9
                    ;             = 0x00A + 0x009
                    ;             = 0x013
                    ; Carry, Zero Flag is Reset
                    ; Digit Carry Flag is Set
```

The lower nybble result of the operation above is actually 0x013, which cannot be stored in the four bits of the nybble. *3* is stored in the nybble and the upper nybble is incremented by one (which is the *Digit Carry*). In this example, the operation of the Digit Carry bit is quite obvious, in situations where the upper nybble is not zero, the action of the Digit Carry flag can be more subtle and difficult to observe.

The Digit Carry flag is primarily used for *Binary Code Decimal (BCD)* applications in which the result of an add/subtract operation has to be converted back to BCD format.

For example, when adding 37 and 24 as BCD, the result of a straight addition is 0x05B, which is not the BCD result 61 expected. To convert the result, the Digit Carry flags have to be checked. It is set, then 10 could be subtracted from the lower nybble's result. If it is not set, then if the value in the lower nybble is greater than 9, then 10 should be subtracted from it and the upper nybble incremented (0x010 added to the result).

Although the previous paragraph is just about impossible to understand, I could write it out using C pseudo-code as:

```
IntBCDAdd( A, B ) {

int Result = A + B;             // Do a Straight Addition

  if (Digit Carry == 1)         // If the DC is Set, Subtract 10 from
    Result = Result + 6;        // the Lower Nybble while keeping the
  else                          // Higher Nybble at the Same Value.
    if ((Result & 0x00F) >9 )   // If the Lower Nybble is 10 or greater,
```

```
    Result = Result + 6;      // Subtract 10 from the Lower Nybble
                              // and Increment the Higher Nybble.
  return Result;

} // End IntBCDAdd
```

This function probably makes less sense than the written description, but by working through the code, you should get an idea of how the BCD addition works. In the code, notice that I have recognized that taking 10 away from the lower nybble while keeping the upper nybble at the same value is the same as simply adding six to it (to get the base 10 result) when the Digit Carry flag is set. Coincidentally, adding six is also the same as incrementing the upper nybble and taking 10 away from the lower nybble result when the result is greater than 9, but the Digit Carry flag is not set.

I feel like I have gone somewhat out into left field in trying to explain how the Digit Carry flag can be used in applications. The STATUS register's Digit Carry bit can be best summarized as initially seeming like a fifth wheel, but can actually help make some operations easier to implement, as is shown in this example and some code examples given elsewhere in the book.

The Carry, Digit Carry, and Zero flags are the only STATUS flags available within the low-end and mid-range PICmicro® MCUs. Other processors (including the PIC17Cxx and PIC18Cxx) have additional flags to help understand what the result of an operation actually is. Some processors STATUS registers have explicit Negative and Overflow flags, which indicate if a result is negative or if it is an invalid two's complement number, respectively. Other flags might store the parity of the result. These features can be simulated within any processor, their inclusion is meant to simplify arithmetic operations and avoid the need to develop software routines to determine these conditions.

The STATUS flags are used to control conditional execution. Normally, an application executes "linearly" through its instructions. At specific points in the code, *jumps* or *gotos* to different locations might be warranted. Changes in execution that always take place are known as *unconditional jumps*. The STATUS flags (as well as other bits accessible by the processor) can be used to change execution "conditionally."

In most processors, the conditional jump instructions test the state of a STATUS flag and execute the jump if the condition is true. Table 38 lists the standard conditional jump instructions available to a processor, along with their opposites or complements.

TABLE 38 Different Conditional Jumps available in Assembly Language Programming

CONDITIONAL JUMP	OPPOSITE	DESCRIPTION
jz	jnz	Jump if the Zero Flag is Set.
jnz	jz	Jump if the Zero Flag is Reset.
jc	jnc	Jump if the Carry Flag is Set.
jnc	jc	Jump if the Carry Flag is Reset
jdc	jndc	Jump if the Digit Carry Flag is Set.
jndc	jdc	Jump if the Digit Carry Flag is Reset.

I include the opposites because, when converting high-level statements into assembly language, the opposite is actually needed for the execution. For example, to write:

```
if (A == B) {
 :                       // Statements
}
```

in assembly language, a *Jump on Zero* is not used, instead the opposite, *Jump on Not Zero,* is used:

```
  MovAcc A                    // "if (A == B) {"
  SubAcc B                    // "if (A == B) {"
  Jnz Skip                    // "if (A == B) {"
   :                          // Statements
Skip:                         // "}"
```

In some processors, like the PICmicro® MCU, there are no *jump on condition* instructions. Instead there are *skip the next instruction on condition* instructions. These instructions will skip over the next instruction if the condition is true. Normally, a *skip the next instruction on condition* instruction is followed by a *jump* or *goto* instruction, which simulates the *jump on condition*. These instructions can be confusing to work with if you are used to *jump on condition* because instead of executing based on the negative condition, the skip will execute based on the actual condition. To illustrate this, the previous example can be re-written using *skip the next instruction on condition* as:

```
MovAcc A                // "if (A == B) {"
SubAcc B                // "if (A == B) {"
SkipZ                   // "if (A == B) {" -Skip on Zero
 Goto Skip              // "if (A == B) {"
 :                      // Statements
Skip:                   // "}"
```

The *skip on condition* might seem like a less efficient way to implement instructions. But they provide a great deal of flexibility that is demonstrated in the book.

Along with jumps to new locations in an application, subroutines can usually be "called" in most processors. In some cases, instead of pushing the return address onto a stack, it is stored in a register and when it is time to return from the subroutine, execution jumps to the contents of the register.

Although calling and operation of subroutines is quite straightforward, I do want to mention how parameters are passed between the calling code and the subroutine.

Three basic methods are used to pass parameters. The first is passing the parameters in the processor's registers. This usually results in the fastest code execution and fewest overhead instructions. The second method is to use common variables, which are accessible by both the caller and subroutine. This method can be confusing to write if there are numerous subroutines. The last method is to pass data on the processor's data stack. For many processors, like the low-end and mid-range PICmicro® MCU, this method can be very inefficient in terms of memory and instructions. In others (such as the PC's 8088 or the PIC18Cxx), special instructions and addressing modes makes using the stack for parame-

ter passing extremely efficient. Determining the best method to pass parameters is a function of how much variable memory, program memory, and instruction cycles you can spend on the task. When you begin assembly-language programming, you will be faced with the task of converting complex statements like:

```
A = B + (C * D)
```

which will seem daunting to code at first. My recommendation for this situation is to break down the statement into single assignment expressions ordered in terms of execution priority and convert each one to assembly-language. For this example, this conversion would be:

```
Temp = C * D
A = B + Temp
```

Both of these statements are relatively easy to code.

An immediate "optimization" that you might see is to avoid using a *Temp* variable and save everything in *A*, so the two statements look like:

```
A = C * D
A = B + A
```

When you start out, I would recommend that you avoid using the destination as a temporary value until you are very familiar with assembly-language programming. Not having intermediate values will make your code harder to debug (if the wrong result is produced, you can check the intermediate values, rather than go through each instruction to find the problem). As well, in some registers in the PICmicro® MCU, where a peripheral hardware register is the final destination, problems can occur with the peripheral's operation if the intermediate value is passed before the final result.

This section introduces a lot of material in just a few pages and I hope it isn't overwhelming you. I have tried to keep this material general, with pointers to both the concepts presented earlier in the book, as well as specific points for the PICmicro® MCU. The purpose in doing this is to give you a bridge between other books that introduce assembly-language programming and use processors other than the PICmicro® MCU.

If you are feeling unsure about your ability to learn assembly-language programming, take heart in the knowledge that most of it is based on memorization. Once you understand the basic programming concepts and the processor architecture you are working with, you will find that developing assembly-language applications will largely consist of "cutting and pasting" pieces of code that you know together. I find that when I'm not sure of how to do something, by checking references (such as this book) and other people's applications (which are plentiful for the PICmicro® MCU on the Internet), I can usually develop an application very quickly and quite efficiently.

Interrupts

Developing interrupt handlers and understanding when interrupts should be used is one of the most intimidating things that new application developers will encounter as they work through their first applications. Interrupts are perceived as tools, which will make the re-

sponse of a device to a hardware event not only faster, but also much more complex. This is incorrect as interrupts are not only very easy to add to an application, but they will often also make the application much simpler as well.

As presented elsewhere in the book, interrupts are hardware requests to stop executing the current code, "handle" the hardware request, and resume executing where the code stopped to respond to the interrupt handler. This execution sequence is shown in Fig. 18.

When the interrupt handler is called, it must save all the "context" registers used by the processor. These registers include the accumulator, STATUS register, INDEX register, and any others that are used and change during the execution of the interrupt handler, which could conceivably affect the operation of the processor and the application. For programmers working with interrupts for the first time, I recommend that all possible registers are saved to avoid any potential problems.

One of the biggest mistakes made by developers creating their first interrupt handlers is to not restore the same registers that were saved or they are restored in a different order in which they were saved. The latter problem is not a specific PICmicro® MCU problem, but can be a big problem for devices, which use *push* and *pop* instructions to save the context registers.

To avoid this problem, I usually first write the context save and context restore code before the interrupt handler itself. It is my goal to be always able to see the two operations on the same editor window. I normally put in the code as:

```
Interrupt Handler() {         // Interrupt Handler Operation

      Push(Accumulator);
      Push(Status Register);
      Push(Index Register);

// #### -Put interrupt handler code here.

      Pop(Index Register);
      Pop(Status Register);
      Pop(Accumulator);

}     // Return from Interrupt.
```

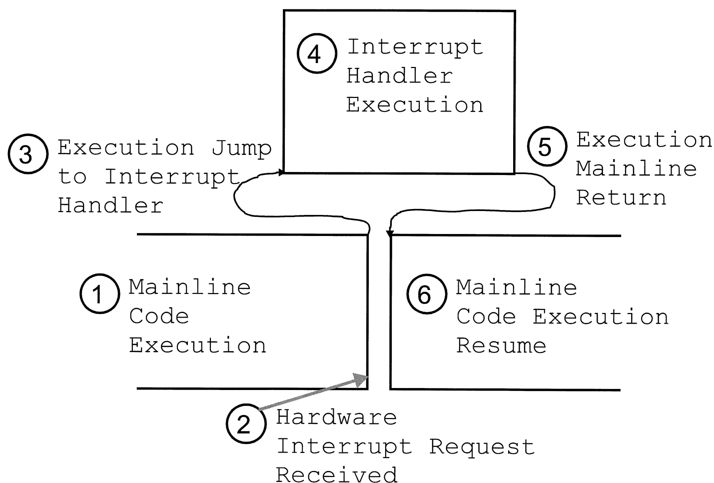

Figure 18 Interrupt execution

By "building" the interrupt handler this way, I can see the order in which the instructions are saved and then be sure that I restore them in the opposite order so that the stack isn't a problem.

Another aspect to note about this code is my use of the #### character string as a "to do" marker. For large applications, which have thousands of lines of source code, it can be hard to find a specific line that is waiting for me to update. By placing #### I am marking a place where I have to update. Other strings that can be used are *$$$$* and *%%%%.* The important feature about these strings is that they are not used in any programming language or environment that I am aware of and will cause an error if an assembler or compiler encounters them.

If I don't lead the #### string with a comment, then when I attempt to assemble or compile the code, I will get an error. This is useful when the string indicates where code that has yet to be written for the application is located.

With the context registers saved, I will next set them to the appropriate values needed for the interrupt handler to run properly. In the PICmicro® MCU, this is often done during the context register save operation.

Once the context registers are saved, then I handle the interrupt in three steps, which are:

1. Save the interrupt occurrence.
2. Reset the interrupt STATUS registers.
3. Handle the interrupt.

This might seem like a surprising order, but it is used to handle nested interrupts, as well as interrupts that can come in faster than the interrupt handler can process. The goal of this sequence is to record the operation of the interrupt and reset the hardware as quickly as possible to prevent the opportunity for an interrupt request to be "missed."

Saving the interrupt occurrence consists of saving the data relevant to the interrupt request. An example of this would be saving the byte received by the serial port. This occurrence data should be saved in a *First In, First Out (FIFO)* "queue" to ensure that the data isn't lost.

Resetting the interrupt STATUS flags consists of returning the interrupt request hardware into the state in which interrupts can be requested again. This is normally accomplished by first resetting the requesting hardware, followed by the computer system's interrupt controller.

Once the interrupt request is processed and reset, the interrupt data can be processed. If the nested interrupts are to be used in the application, they can be allowed at this point. *Nested interrupts* are interrupts that can execute while the current interrupt is being processed (Fig. 19).

With the rules and order presented in this section, you should be able to work with nested interrupts without any problems. To help ensure there aren't any problems, keep the interrupt processing as short and as fast as possible.

A good rule is to keep interrupt handling as short as possible. Long interrupt-handler processing can result in missed interrupts or unacceptably long delays in passing data to process (resulting in missed "acknowledgement windows") in the application. Often, the ideal data size for processing data is one byte, which means that fast interrupt handler operation and response is crucial to avoid missed data.

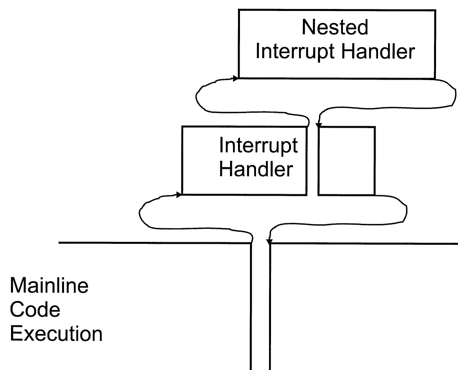

Figure 19 **Nested interrupt execution**

Fuzzy Logic

If you've ever taken (or suffered through) a course in control theory, you probably remember how difficult it was to understand what was happening and how "unnatural" it seemed. (I was never really able to visualize what exactly what was going on with the imaginary polar drawings). A more modern solution to controlling processes is to use a digital signal processor to implement the digital-control theory algorithms. This solution is as bad as trying to understand classic control theory and requires a lot of effort on the part of the person designing the system to come up with a correct response algorithm. Visualizing a problem or a situation is very important for humans to be able to understanding what is happening and enables us to come up with a solution. Traditional control theory taught in third and fourth years of college Electrical Engineering only offers very mathematically intensive solutions to the problem that are very hard to visualize and work through.

One interesting nontraditional method of controlling systems is known as *fuzzy logic*. Fuzzy logic can be considered a superset of Boolean logic, where values between *1* and *0* are input into the system and a nonbinary results are output.

The input values are manipulated in a manner similar to Boolean logic. Most people consider fuzzy logic to be a recent invention, but fuzzy logic was first described in 1965, 35 years ago as I am writing this! The main reason for fuzzy logic seeming to be very new is the almost total disdain shown for it in North America. In Asia (especially Japan), fuzzy logic has become an important technology used in many applications, ranging from air-conditioner thermostats to determining when and how much to feed infants.

Professor Lofti Zadeh of UC Berkeley, inventor of fuzzy logic, defined the following characteristics of fuzzy systems:

1 In fuzzy logic, exact reasoning is viewed as a limiting case of approximate reasoning.
2 In fuzzy logic, everything is a matter of degree.
3 Any logical system can be "fuzzified."
4 In fuzzy logic, knowledge is interpreted as a collection of elastic or equivalently fuzzy constraints on a collection of variables.
5 Inference is viewed as a process of propagation of elastic constraints.

To explain how to define a fuzzy-logic system, I will use the classic control system example of the inverted pendulum. The mechanical system is shown in Fig. 20.

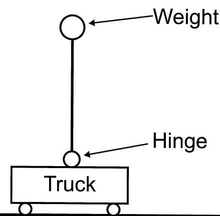

Figure 20 **Inverted pendulum**

The purpose of this exercise is to keep the weight at the end of the pendulum vertical by moving the truck back and forth, using inertia to keep the weight upright. If you're not familiar with this example, take a pencil and balance it vertically with the point down on one of your fingers. See how long you can keep it upright (the best I could do was about five seconds).

In developing a fuzzy control system, you must first "fuzzify" the system. This is done by specifying different output conditions based on different inputs.

In this example, the output is the speed and direction of the truck. The inputs are the angle of the pendulum, its angular velocity and the position of the track.

A fuzzy set is defined for each input that is a collection of patches defining different conditions. One of the parameters that I wanted for my system was to keep the truck in the center of the system (so that it could react to changes in the pendulum without running out of track to respond with). So, I defined the position as five different patches defining whether the truck was at the "Center," "Near," or "Far" away. The patches should cover every part of the base variable (*X*-axis). A sample set of patches to define the truck position is shown in Fig. 21.

This effort is to be repeated for the angle of the pendulum, as well as its angular velocity.

Once the inputs are fuzzified, a set of rules are developed governing the behavior of the outputs from different input conditions. These rules specify for given conditions, what is the output response. As can be seen in the diagram, the position could be in both the "Center" and "Near," which means that multiple rules could be valid for a given condition. This might seem unusual, but in fuzzy systems, this is normal. Multiple rule intersection helps define what the response should be.

Some rules for this system could be:

```
if Angle = Vertical AND Angular_Speed = Stopped AND Pos = Center
  then Speed = 0

if Angle = -Lean AND Angular_Speed = -Slow AND Pos = Center
  then Speed = -Slow
```

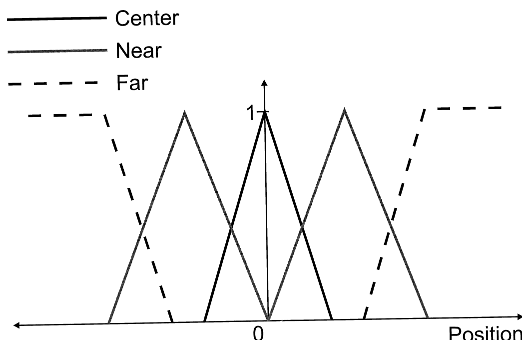

Figure 21 **Fuzzy position**

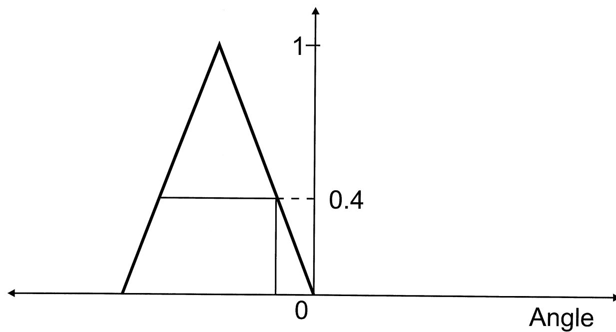

Figure 22 **Fuzzy angle**

and so on for each different case. This can be defined several different ways (such as if-then, as shown, or using a modified Karnaugh map).

With the rules in place, the inputs are run through them. If we were to look at the second rule, we might have a negative angle that maps out to what is shown in Fig. 22.

This angle returns a fuzzy value of 0.4. Now, to use this with the other fuzzy operations in rule number 2, we are ANDing this value with the *Angular_Speed* and the *Pos* to get a final result. In fuzzy Boolean operations, the mapped value (0.4 for the angle, in this example) is compared to the other mapped values and the lowest value is passed along.

ORing and NOTing can be done as well. For ORing, the maximum mapped value is passed along. For NOTing, the current value is subtracted from 1. All Boolean transformations and laws (i.e., associativity) apply to fuzzy logic, so the result can be computed easily.

This value is then used to create a weighting for the output value. This weighting is important because if multiple rules are satisfied, the output of each are combined and a final output is produced.

These different outputs are weighted together to get the actual output response. In the inverted pendulum example, the following diagram shows what happens when two rules are satisfied and two weighted outputs are combined (Fig. 23).

The output could be anywhere in the shaded area, but a center of mass is taken and used as the actual output. This response is balanced between the output of the two rules by taking the center of mass of both the responses.

This is really all there is to fuzzy logic. I realize that explaining a fuzzy control system with one example and one condition doesn't really explain how it works. But, it is a lot

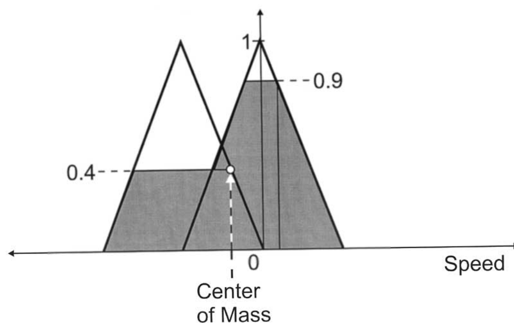

Figure 23 **Fuzzy result**

simpler than explaining how classic or digital control systems operate (which would probably take up the whole book and maybe one or two volumes).

Elsewhere, the book shows how two example control applications work with Microchip's "fuzzyTECH" fuzzy logic development system. This system makes the development of fuzzy-logic control systems quite painless and allows you to monitor the operation of the system before you "burn" it into a PICmicro® MCU.

Event-Driven Programming

So far, this appendix has described linear programming. This method of programming revolves around testing and responding immediately to different inputs as the application executes. This technique is typified in the simple application code:

```
main ( )
{

int i = 0;

   while (1 == 1) {
      output (i & 0x00F);          // Output Counter
      i = i + 1;                   // Increment Counter
      if ((i % 2) == 0)            // Delay According to whether or not
       delay( 1/2sec );            // the Counter is Odd or Even
      else
       delay( 1sec );
   }                               // Loop Forever
}  // end Example
```

In this code, the application runs loops, each of which outputs the current counter *i* value before incrementing the counter and then delaying based on the counter value. This method of programming works well when there is only one task for a processor to perform for an application.

If multiple tasks that respond to multiple inputs are required, then some form of nonlinear programming is required. *Nonlinear programming* is the term used to program applications in such a way that execution is not limited to following a predetermined "linear" path. Instead, execution is based on responding to current input conditions. There are a number of approaches to take with nonlinear programming. This section and the next describe two of the most popular techniques. The body of the book presents a real-time operating system and multi-tasking as another method to provide responses to external stimulus without having to follow a linear path.

One irony about nonlinear programming that is important to realize is that although it is considered to be an advanced programming topic and can be difficult to learn, the purpose is to simplify application programming. Nonlinear programming allows the application developer to write complex applications without having to develop methods of allowing the different I /O to interact. In each of the different methods of nonlinear programming described in this book, applications are written from the perspective of responding to individual inputs, without having to take into account how other inputs are handled or what is happening in the other tasks.

The first description of nonlinear programming is *event-driven programming*. In this style of application development, execution lies dormant until inputs are present to be re-

sponded to. These events can be timer overflows, button presses, incoming serial data, or anything else that comes in asynchronously (i.e., not on a defined schedule). If you are familiar with Microsoft's Visual Basic, you will realize that this method was used to develop Microsoft Windows applications.

If you are to look at a true event-driven program, you would find that the "mainline" would consist of:

```
main()                        // Event Driven Programming Example
{

// Initialize Variables and Hardware
// Set up Real Time Clock
// Set up Interrupt Handlers

  while (1 == 1);             // loop forever
                              // and let Interrupt Handlers respond and
                              // process the inputs
}  // End Event Driven Mainline

interrupt Event1Handler()
{

   :                          // Handle Event "1"

}  // End Event1Handler

interrupt Event2Handler()
{

   :                               // Handle Event "2"

}  // End Event2Handler

   :                               // Additional Event Handlers
```

Once execution is taking place, the processor is "spinning," waiting for events to respond to. These events are normally interrupt requests that have to be acknowledged and the incoming data responded to.

The example application presented at the start of this section could be enhanced by adding a one-second timer interrupt to the main line and a timer interrupt handler:

```
main()                         // Event Driven Updated Initial Application
{

int i = 0;

  TimerDelay = 1sec;

  interrupts = TimerHandler;

  while(1 == 1);

} // end main

interrupt TimerHandler(){      // Display "i" and Increment

  TimerInterrupt = Reset;

  output(i & 0x00F);

  i = i + 1;

} // End TimerHandler
```

This code probably doesn't look any simpler than the first example, but the advantages become apparent when more event handlers are added. To show this, I will add the code to turn on and off an LED based on the state of a button.

```
main()                        // Event Driven Updated Initial Application
{
int i = 0;
  TimerDelay = 1sec;
  interrupts = TimerHandler | ButtonHandler;
  if (Button == Up)           // Load in the Initial Button State
    LED = off;
  else
    LED = ON;
  while(1 == 1);
} // end main
interrupt TimerHandler()    { // Display "i" and Increment
  TimerInterrupt = Reset;
  output(i & 0x00F);
  i = i + 1;
} // End TimerHandler
interrupt ButtonUp( )
{
  ButtonInterrupt = Reset;
  LED = off;
} // end ButtonUp
interrupt ButtonDown( )
{
  ButtonInterrupt = Reset;
  LED = ON;
} // end ButtonDown
```

Notice that to add the button interrupt functions, the timer delay code was not affected. Looking back at the original application code, I would have to figure out how would these button routines are to be added. Before that decision could be made, I would have had to look at how the *delay(1sec)* subroutine worked and how it could be changed to accommodate the button polling LED on/off function.

Using event-driven programming techniques, the button functions are simply added without affecting the operation of the code that was written around it and are not part of the functions. This feature allows multiple functions to be created for an application by different people using different hardware.

The key to event-driven programming is to design the application so that multiple events can be handled without being "stacked" and ultimately lost. This is actually quite easy to do, but some time must be spent when the application is designed. The ideal is to

design an event-driven programmed application in such a way that it can always respond in less than half of two times the worst-case time that can be tolerated.

State Machines

Creating *state machines* in software is another technique of nonlinear programming that you can use for your applications to avoid having complex operation comparisons and sophisticated decision code. Software state machines can be implemented in a variety of different ways, depending on the requirements and capabilities of the application. This section introduces programming state machines for the PICmicro® MCU.

State machines were first designed as hardware circuits that perform sequential, decision-based operations without any logic circuits built in (Fig. 24). The current state is the address produced by combining the previous state and the external inputs of the circuit.

Software state machines work in a similar manner with a "state variable" used to select the operation to execute. The most common way to describe a state machine is to use a *select* statement to show how the different states are selected and what happens in them. For a simple "traffic-light" state machine application, the code could be shown as:

```
main()                              // Traffic Light State Machine
{

int State = 0;                      // Start State at the Beginning

  while(1 == 1)                     // Loop forever
   switch(State) {                  // Execute according to the State
    case 0:                         // Power Up - All Red Before Starting
     NSLight = Red;                 // North/South Switch Red
     EWLight = Red;                 // East/West Switch Red
     Dlay(20 Seconds); // Wait for Traffic to Stop Before
                                    // Proceeding
     State = 1;                     // Jump to the Next State
     break;
    case 1:                         // Make North/South "Green"
     NSLight = Green;
     EWLight = Red;
     Dlay(30 Seconds);              // Active for Thirty Seconds
     State = 2;
     break;
    case 2:                         // North/South "Amber"
     NSLight = Amber;
     Dlay(10 Seconds);              // Cars have Ten Seconds to Stop
     State = 3;
     break;
```

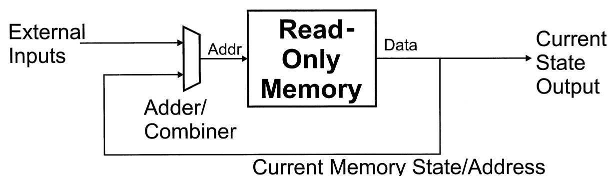

Figure 24 Hardware state-machine implementation

```
    case 3:                         // North/South Red, East/West Green
     NSLight = Red;
     EWLight = Green;
     Dlay(30 Seconds);              // Active for Thirty Seconds
     State = 4;
     break;
    case 4:                         // East/West "Amber"
     EWLight = Amber;
     Dlay(10 Seconds);              // Cars have Ten Seconds to Stop
     State = 1;                     // Start all over again
     break;
   } // End Switch

} // End Traffic Light State Machine
```

In this example, delays are put into each state. Also notice that I don't set both light values in every case. This might not be recommended based on the actual hardware. In this application, I assumed that once a value was set, it would stay set and the processor would not jump to an invalid location.

Another assumption I made was that programmable delays could be called from the mainline. In a PICmicro® MCU application, the state machine might be active every time that TMR0 overflows. In this case, if the overflow occurs at a constant time (e.g., 10 seconds), multiple delay executions for each state are required. In this case, the example application shown changes to:

```
main()                              // Traffic Light State Machine
{                                   // - with a 10 Second Constant Delay

int State = 0;                      // Start State at the Beginning

  while(1 == 1) {                   // Loop forever
     Dlay10 Seconds();              // Delay Ten Seconds
     switch(State) {                // Execute according to the State
       case 0:                      // Power Up - All Red Before Starting
         NSLight = Red;             // North/South Switch Red
         EWLight = Red;             // East/West Switch Red
         State = 1;                 // Jump to the Next State
         break;
       case 1:                      // Want 2x Red Light Delay
         State = 2;                 //  for Twenty Seconds
       case 2:                      // Make North/South "Green"
         NSLight = Green;           //  for Thirty Seconds
         EWLight = Red;
         State = 3
         break;
       case 3:                      // North/South Green for 20 seconds
         State = 4
         break;
       case 4:                      // North/South Green for 30 seconds
         State = 5
         break;
       case 5:                      // North/South "Amber"
         NSLight = Amber;           //  for 10 seconds
         State = 6;
         break;
       case 6:                      // North/South Red, East/West Green
         NSLight = Red;             //  for 30 seconds
         EWLight = Green;
         State = 7
         break;
```

```
      case 7:                   // East/West Green for 20 Seconds
        State = 8;
        break;
      case 8:                   // East/West Green for 30 Seconds
        State = 9;
        break;
      case 9:                   // East/West "Amber"
        EWLight = Amber;        //  for 10 Seconds
        State = 2;              //  Start all over again
        break;
    }  //  End Switch
  }  // End While

}  // End Traffic Light State Machine

{#### - End Code Example}
```

Notice that, in this case, the reset value has changed from state 1 to state 2 because two 10-second delays are required for the 20-second settling time to get the traffic to a known condition (stopped) before starting the process. When the green lights are on for 30 seconds, you can see that three states are encompassed to provide the delay.

This is actually the method that I use when developing Visual Basic state machines. In this case, I use one of the programmable timers to initiate the state machine and then work from there. This method is directly transferable to implementing state machines in the PICmicro® MCU's timer interrupt or responding to changes in inputs while in an infinite loop.