4
CHAPTER

Introduction to Programming

If you have never programmed before, the thought of writing and creating a program for the robot can be daunting. One of the reasons we choose the Parallax Basic Stamp 2 as the control processor for the *TAB Electronics Build Your Own Robot Kit* is how easy it is to program. I'm sure that you will find that you will be able to easily write your own programs for the Basic Stamp after reading through the chapters on this CD-ROM.

Now having said this, it is important for you to realize that there are some basic concepts that you will have to apply. These concepts are not difficult to understand, and they will apply skills that you have developed in your everyday life (like programming a VCR). As you work through the experiments that I have provided and try out your own programs for the robot, I know that you will find that programming is not that scary a task.

In this chapter, I would like to introduce you to the basics of programming and the high-level BASIC language used to program the Parallax Basic Stamp 2 (BS2) that can be plugged into the *TAB Electronics Build Your Own Robot Kit*. This chapter will give you the background to understand what I am doing when I show you how PBASIC application code is written for the robot.

When I originally wrote this chapter, I included a fairly lengthy description of the Parallax PBASIC language. Instead of subjecting you to that description, I would like to point you to Parallax's outstanding documentation included on this CD-ROM. The Parallax Basic Stamp Manual is the best resource for learning about the Basic Stamp 2's PBASIC language.

In the following chapters I will describe robot "behaviors" and walk you through the process of programming a Parallax Basic Stamp 2 and using it to control your robot.

Data Types

Computer processors are really designed for one thing and that is 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, which are 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 will be 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 opposite 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 referred to as a "nybble." The nybble uses the characters "0" to "9" and "A" to "F" to represent the sixteen different possible states stored in the four bits. The table below shows the characters represented for each of the different bit states:

Bits (3-0)	**Nybble character**
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"

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

%11000110

can be written in the nybble/hex format as:

$C6

The prefix "%" put before a string of numbers indicates to the Basic Stamp processing code that the value is a binary number. If a "$" precedes a number, it is interpreted as a hexadecimal. Often when I program, I will place a "0" (zero) after the

data type specifier (the "%" or "$") out of habit—many old assemblers that I have worked with required a zero to correctly interpret the value (especially a hex value starting with a letter code). In this chapter and in most books, the hex/nybble format is the preferred method of listing binary data.

From the table of values above, it is useful to note that the first number in any binary sequence 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" which is 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 %11111111 or $FF) 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, and so on, as is shown in the table below:

Bit	Two's power of bit	"hex" value
0	1	$01
1	2	$02
2	4	$04
3	8	$08
4	16	$10
5	32	$20
6	64	$40
7	128	$80

From this table, you should be able to see how the bits are combined to form a byte. For the example above ($C6), to make up the number, the bits 1, 2, 5, and 6 of the byte are set.

Negative values are normally stored in "2's complement" format. To generate a 2'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( %00101111 ) + 1
    = %11010000 + 1
    = %11010001
    = $D1
```

The number "NOT" operation is the same as XORing the number with $FF.

The advantage of using 2's complement format is that it is automatically produced when a number is subtracted from a smaller value: it can be added to a positive or negative 2's complement number and the result will be correct (if it is positive or negative).

For larger pieces of data, bytes are normally "concatenated" together to form the new data type. Sixteen-bit numbers have a greater range of 2'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 (thirty-two bits) can be combined to provide even larger data values and are very common for PCs and workstations.

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:

$$s\ mmmm\ x\ (2^{(e-bias)})$$

where "s" is the sign of the number, "mmmm" is known as the "mantissa" or "precision" of the value, "e" is the 2's complement exponent that denotes the order of magnitude of the mantissa. The exponent has a "bias" value subtracted from it, which is used to compensate for the number of mantissa bits in the number.

The IEEE 754 "double precision" data format uses the bit structure:

Bits	Purpose
63	Sign bit
62-52	Exponent
51-0	Mantissa

I will not go into detail explaining how floating point numbers are processed (probably much to your relief). I've just shown it here so that you can better understand how exponents in scientific notation are actually used in a computer system.

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" ($00) character. When processing these strings, I simply read through them to the first $00. Lists of ASCIIZ strings can be put together with the end of the list 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 be updated after the string. This is something that that 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 altogether.

Data types can also be extended to include multiple data types built into a single "structure." Structures are useful in data base programs and other applications that process large amounts of data. I do not find structures to be very useful in Basic Stamp applications, as there is not that much memory space that can be devoted to strings.

Programming Basics

Many people have learned how to program in a variety of different ways. I first learned 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 difficult, forcing students to learn two new concepts (programming and working with the computer) at the same time.

There are four basic programming concepts, which are very simple to understand. Once you master these basic concepts, you will recognize them and use them in "high-level" programming languages. In this section I will to introduce these four concepts as well as one additional concept that are applied in virtually all programming environments.

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 the realization that I have nothing and nobody to be angry with except myself.

When you try to compile or assemble your own Basic Stamp 2 programs, chances are the PC running the Windows Basic Stamp Editor will tell you that there are "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 to keep calm and try to develop different techniques for finding the problems.

Even 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 Basic Stamp 2 processor is only executing what it has been given.

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

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 and accept, but *you* are the weakest link in the application development process. It's easy to blame a problem on the Windows development software, the particular Basic Stamp 2 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" is probably an overly complicated way of describing 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 that is often used is a set of "cubbies" into which sorted mail is stored. In the computer memory model, each cubby, which has its own unique address, is a byte and the contents can be changed. The changeable aspect of these cubbies is where the name "variable" comes from.

In Figure 4-1, the four by four array of cubbies has been given addresses based on the row and column they are in. The address is defined by the formula:

$$\text{Address} = (\text{Row} * 4) + \text{column}$$

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.

In Figure 4-1 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 the least significant two bits of the address. These four bits (which are a "nybble") produce an address which is equivalent to the result of the formula above. 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 the formula above.

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."

The data in the memory locations can be accessed one of two ways. "Direct" memory access 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.

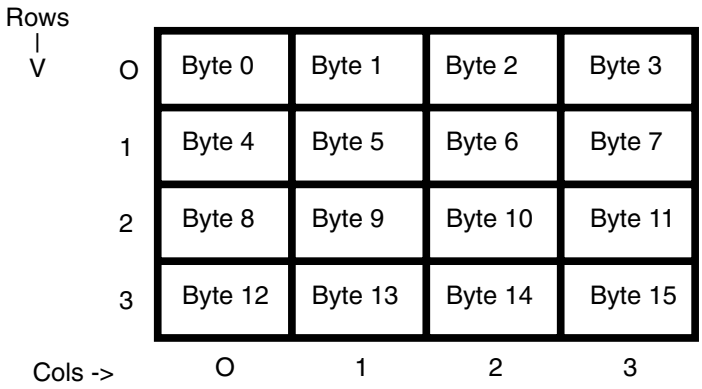

Figure 4-1 Memory "cubby" model.

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 4-2 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 $04 through $0D (decimal 13).

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:

$$\text{Address} = \text{offset} + \text{element}$$

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

$$\begin{aligned}\text{Address} &= \text{offset} + \text{element}\\ &= 4 + 9\\ &= 13\end{aligned}$$

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.

In addition to being character strings, array variables can also be a collection of bytes that can be arbitrarily accessed. For example, you may be recording the weights put on a series of scales as is shown in Figure 4-3.

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

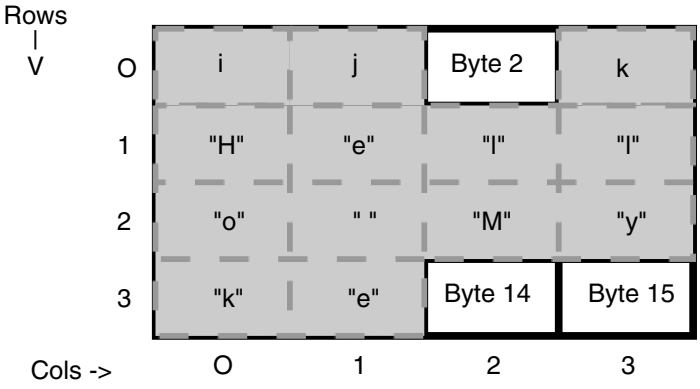

Figure 4-2 Variables in "cubbies."

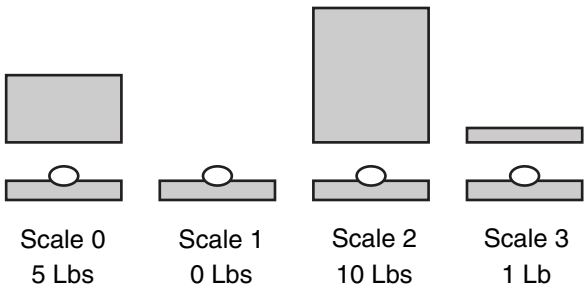

Figure 4-3 Scale values stored as array.

simply calculating the address using the element number of the scale. Using the array allows an application to access individual data points at random; no time or code penalty is assigned.

"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. This is shown in Figure 4-4.

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 above is used:

Address = offset + element

In this case the element number must be calculated using the "x" and "y" address of the element. When I set up two dimensional arrays using the Basic Stamp 2, 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.

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. The problem with doing this is that the memory required will increase geometrically for each dimension that you add. Realistically speaking, if your application requires a multidimensional array, I recommend you take another look at how you are planning to implement the code.

So far when describing array variables, I have only mentioned the memory word size (a "byte" for the examples here). Multiples of the word size can be combined to produce different data "types." In the Basic Stamp 2, sixteen-bit values, eight-bit values, and one-bit values can all be used either as single or array variables.

In most languages (including the Basic Stamp 2's PBASIC), variables are "declared" before they are used. "Declaration statements" are used to notify the compiler/

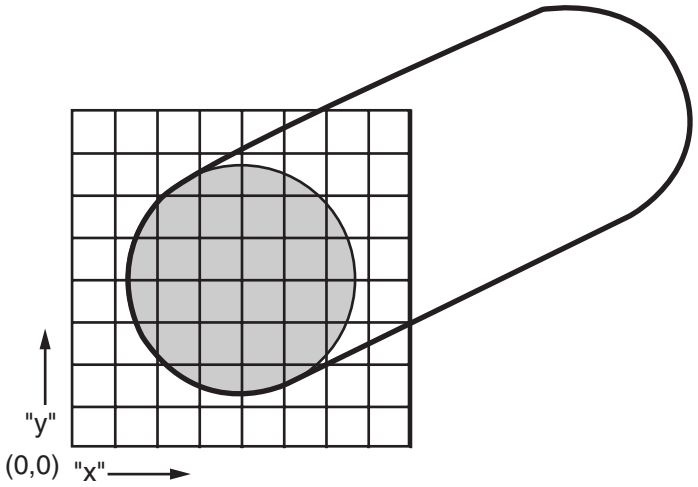

Figure 4-4 Two-dimensional array around a pipe.

assembler that space is needed for the variable and, optionally, the variable is set to an initial value.

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 Basic Stamp 2 application programming. I can summarize what you have to know for variables when programming with the following three points:

1. Variable memory is stored as an array of bytes.
2. Bytes can be combined to make larger variables or different array 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

Note that the "destination" (which can also be referred to as the "result") is on the left hand 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 (which will be explained in greater detail below). 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 what a variable is and what an array element is. 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.

In the Basic Stamp 2, variables can either be defined as an explicit memory location or you can let the compiler do it for you. I always recommend that you allow the compiler to specify the addresses instead of doing it yourself—if there is a problem with using too much space, the compiler will flag the error for you. This will save you the work of allocating memory and keeping track of variables as the program's source code changes.

According to 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. The device that does this, 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:

Symbol	Function
+	Addition
-	Subtraction/negation

More advanced functions like:

Symbol	Function
*	Multiplication
/	Division
%	Modulus

may also be available, but in small processors (like the one used in the Basic Stamp 2) these functions have to be created using a series of programmed functions.

Normally these functions work only on "integers" (whole positive and negative numbers) and not "real" numbers. Some processors (generally PC, Workstation, and Server) can work with real numbers "natively," but most (including the Basic Stamp 2) 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. These operations include:

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)

For these operations, real numbers are normally required. In the Parallax Basic Stamp 2 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 the range 0 to 1 that you are probably most familiar with. For the circle of radius 127, note that 2'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 consist of:

&	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 if only one 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 $FF 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

These operations are always built into computer processors.

The last type of operator is known as "logical" operator and is used to compare values and execute according to the results. Logical operators work only 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:

=	Compare two values for equality and return "true" if they are equal
<>	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 vice versa)

AND	Return "true" if two logical values are both "true"
OR	Return "true" if either one of two logical values is "true"

The results of these logical operations are normally not zero ("1" or "-1") for "true" and "0" for "false."

When using arithmetic functions, the "assignment" format is used with the "source" now referred to as an "expression." Expressions are made up of the arithmetic and bitwise operations listed above.

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 division by eight to get the average. The summation is enclosed by brackets (also known as parentheses) to indicate to the compiler/assembler that it has to be computed before the division can take place.

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." This means that in the example immediately above, element (x, y) will be loaded with the sum of the seven surrounding elements plus one-eighth of the last element.

In most cases, multiplication and division are at a higher priority over addition and subtraction. In others, like the Basic Stamp 2, the order of operations is defined by executing the statements left to right. Using parentheses, the order of operations that you would like the code to follow is "forced."

To avoid problems like the heat transfer example above, I always surround what I want to be the highest priority operations with parentheses to explicitly force the compiler/assembler to evaluate the portions of the expression in the lowest level parentheses 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).

From 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 contents arithmetically modified. Except for one important piece, this is all there is to 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, and I will 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.

The "goto" statement by itself isn't that useful (except for creating endless loops as the one shown above) unless it is coupled with a "conditional execution" statement. Conditional execution is code that is executed only 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 Label
```

Where "condition" is a logical mathematical operation that tests parameters to meet a specific condition.

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

```
  If (A = B) then 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 to B" are executed. Else, 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.

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.

Along with the four primary concepts, there is one more that I want to introduce to you that will make programming easier.

Subroutines are a very useful programming technique designed to eliminate the need for you to type in redundant code. Instead of repeatedly entering in the same code, a subroutine is created and then "called" each time the function it provides is required.

When a subroutine's "call" (or "gosub") 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 Figure 4-5.

When "subroutine" is called, the main execution registers (known as the "context registers") may 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 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.

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.

Editors

An editor is an application program that runs on a PC or workstation to allow a "man-readable" file to be created or changed. The editor can also be used for looking at files (which is known as "browsing"). Over the years, I have probably tried out a hundred or more different editors for developing and modifying application code, "browsing" and changing hex files, creating text (like this pdf file or Web page HTML) and sending emails. Most of these trials were taken at the suggestion of someone else because they had found a "wonderful new editor."

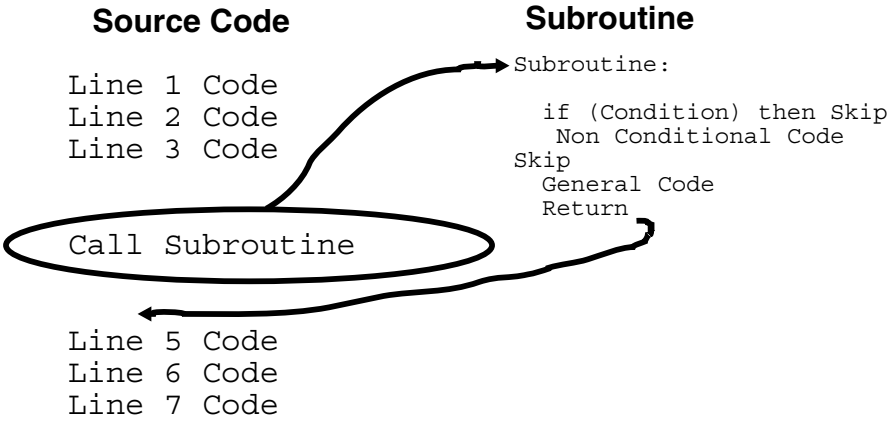

Figure 4-5 Subroutine operation.

Of all the different editors I've tried, I've found only two that I've liked and used for any length of time. For standard editing requirements I use the standard Microsoft Windows WordPad, NotePad, and Word editors. I find that I'm very picky in what I consider to be an outstanding editor, and for the most part I rely on the Microsoft data entry standards with basic text editing features.

The Parallax Windows editor that you will be using for programming the Basic Stamp 2 uses the standard Microsoft editing conventions. Once you have mastered these commands, you will be able to work between these tools without having to learn or relearn special editing commands.

A standard Microsoft editor, like WordPad (Figure 4-6), is a Word-based editor. The cursor, which is the vertical bar where characters will be placed, is moved onto the window by either the arrow keys or by using the mouse and setting its position by a left click of the mouse. When I'm editing a file, I very rarely use the mouse; instead I use the arrow keys, "Home," "End," "Page Up," and "Page Down" almost exclusively.

The following table lists the standard Microsoft editor operations:

Keystrokes	Operation
Up arrow	Move cursor up one line
Down arrow	Move cursor down one line
Left arrow	Move cursor left one character
Right arrow	Move cursor right on arrow
Page up	Move viewed window up
Page down	Move viewed window down
Ctrl–left arrow	Jump to start of word
Ctrl–right arrow	Jump to start of next word

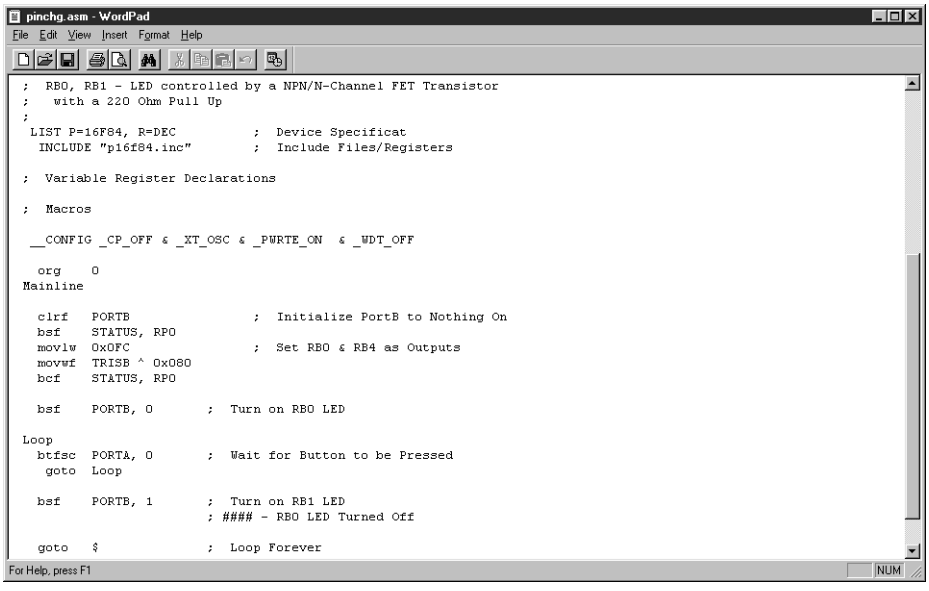

Figure 4-6 WordPad Microsoft Windows editor.

Keystrokes	Operation
Ctrl–page up	Move cursor to top of viewed window
Ctrl–page down	Move cursor to bottom viewed window
Home	Move cursor to start of line
End	Move cursor to end of line
Ctrl–home	Jump to start of file
Ctrl–end	Jump to end of file
Shift–left arrow	Increase the marked block by one character to the left
Shift–right arrow	Increase the marked block by one character to the right
Shift–up arrow	Increase the marked block by one line up
Shift–down arrow	Increase the marked block by one line down
Ctrl shift–left arrow	Increase the marked block by one word to the left
Ctrl shift–right arrow	Increase the marked block by one word to the right

To delete and move text, I use the cut and paste functions. To select text to cut and paste, I mark the text by first pressing a shift key while moving the cursor to highlight the text to be relocated. If you have marked text incorrectly, then simply move the cursor without a shift key pressed to delete the marked text. Pressing your mouse's left button and moving the mouse across the desired text will also mark it. Left-clicking on another part of the screen will move the cursor there and eliminate the highlighting for the text.

Next, the keystroke "Ctrl - X" is used, which removes the marked text from the file and places it into the windows "clipboard." "Ctrl - C" copies the marked text into the clipboard and doesn't delete it. To put the text at a specific location within a file after the current cursor location, "Ctrl - V" is used.

Note that I do not use "Delete" or "Insert." Deleting marked text destroys it completely whereas Ctrl - C saves it in the clipboard so it can be restored if you made a mistake. The insert key toggles between data insert or replace mode for Microsoft keystroke-compatible editors. Normally when an editor boots up, it is in "insert mode," which means any keystrokes are placed before the cursor. I think that this is the preferable mode to operate in.

The Basic Stamp Editor (see Figure 4-7) uses a Microsoft-compatible editor and uses the command conventions listed above.

Originally, Parallax provided a line-based editor for developing Basic Stamp applications that place a "CR"/"LF" character combination at the end of each line displayed on the screen. In a Microsoft-compatible editor, the "CR"/"LF" is used to separate paragraphs, and lines are broken up on the display in order to show all the data on them. If you were to look at a paragraph produced by a Microsoft-compatible editor on a line editor, you would find that the paragraph would be one line and most of it would not be displayed because it was past the right edge of the editor's text window.

I'm mentioning this because if you were to develop Basic Stamp 2 application code on a line-based editor, you would find that the way the code is displayed is different from what you see on the Basic Stamp Editor's window.

One thing you must watch out for with older editors is that they may put a $1A character at the end of a file. In the Basic Stamp Editor and other Microsoft-compatible editors, this character is displayed as a small square box at the end of the file. The $1A

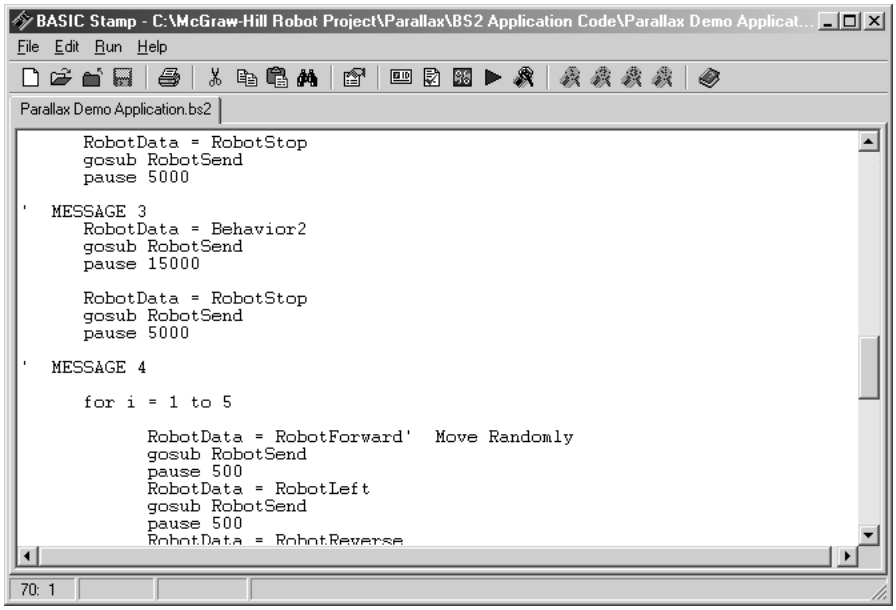

Figure 4-7 Parallax Basic Stamp editor.

character was required by MS-DOS 1.x to indicate a file's end. For some reason, even though MS-DOS and Windows no longer require this file end delimiter, and as MS-DOS and Windows have become more sophisticated in their file handling, this file end indicator has not completely disappeared. While the Basic Stamp Editor and most other tools do not have a problem with this character placed at the end of the file, some other development tools do. If you are using a tool where this is a problem you can simply delete the character in a Microsoft-compatible editor and resave the file before passing it again to the tool.