A Beginner's Guide to baZic(R)

Book I

Developed for Beginning baZic Programmers

developed by

Micro Mike's, Inc.
P.O. Box 1440
Amarillo, Texas 79105

making technology uncomplicated ... for People(R)

COPYRIGHT NOTICE

DISCLAIMER

Micro Mike's, Inc. makes no representation or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Further, Micro Mike's, Inc. reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Micro Mike's, Inc. to notify any persons of such revision or changes.

TRADEMARK NOTICES

This document will mention several names and products which have been granted trademarks. It is the intent of Micro Mike's, Inc. to acknowledge and respect the trademarks of these companies so that all the rights and privileges of these companies are preserved.

baZic is a registered trademark of Micro Mike's, Inc.

CP/M and MP/M are registered trademarks of Digital Research, Inc.

Z80 is a registered trademark of Zilog, Inc.

TABLE OF CONTENTS

A Beginner's Guide to baZic

This manual is designed to teach the programming language **baZic**, a high level BASIC interpreter, to the computer novice. This manual assumes no knowledge on the part of the user of **baZic** or any other programming language. The student should have a working computer system running **baZic** to practice the things learned from this manual.

In this manual, all of the special computer words that you will need to remember will always be shown in UPPERCASE only, since this is the only way the computer will accept and recognize these words. In the instructions, the computer will generally print something on the CRT and the user or programmer will be required to make an entry.

All user entries will be underlined so that the reader of the manual will be able to distinguish the computer printed information from the information the user or programmer is to enter. The symbol <CR> will appear at all places where the user is to press the Carriage Return or Enter Key.

All computer sessions will be indented in this manual to make them readily discernable from the text of the manual. As you are working with the computer and the **baZic** interpreter, you will not indent your responses.

Tests are given after each session. The student should place a piece of paper over the test page and try to answer each question without looking at the answer which is printed directly after each question. If the student can not answer the questions, he should reread the text and do additional exercises.

The student is encouraged to take the tests "honestly" and review his answers to the questions to be certain that he understands the concepts. The only person who has something to gain from reading this manual and taking the tests is the student wanting to learn. If you don't understand an answer go back through that lesson or, if possible, ask some knowledgeable person to help you.

The cursor is used to mark the position on the CRT where the next character is to be entered. The cursor varies from CRT to CRT but is usually a white box or an underline.

Take the sessions at any speed that makes you comfortable. However, you should probably take at least one session a day and more if you feel confident. Never move to the next session until you have mastered all previous sessions.

You should consult the **baZic** manual as a reference when in doubt about the use of any command, statement, or function described here.

If you are using floppy disks, the write-protect tab must be removed from the disk before continuing with the sessions.

As you work the excercises, take careful note of any error messages that occur when you make mistakes. **baZic**, upon the occurrence of an error, will give you a message which indicates the error. The message may not make sense to you when you first see it, but you should become proficient at interpreting error messages by the time you finish this manual.

While working with programs in this manual, you may encounter an error. Look very carefully at the program presented and the way you have entered the program into the computer. Many times the problem will be caused because you have not entered the program correctly. Correct the problem and try again.

Each Session will be divided into five parts. The first part is the OBJECT of the session. The purpose of the OBJECT is to let the student know what the session will try to teach. An attempt has been made to present one unified concept in each session and the OBJECT is used to identify the topics to be learned within that concept.

The LESSON follows the OBJECT and is the text to be learned. Once the LESSON has been completed, you should read the additional information as listed in the ADDITIONAL READING section. Here you will find references to the **baZic** manual and any other pertinent books.

The next section of each session is the TEST. Answer the questions to the best of your ability. If you can not answer the question, you may review the material in the session or use the computer to write a program which will help you determine the answer.

The last section consists of EXCERSISES for you to practice to increase your skill with information learned in that session.

It is highly recommended that you read the manaul while at the computer terminal. Practice everything you learn on the computer. The only way to really learn a computer system is by keeping your hands on the keyboard and spending lots of time working with the computer. This manual should be your guide through this process.

You should read the suggested sections of the **baZic** manual as you progress through this manual. To get the best understanding of **baZic**, read the reference manual "cover to cover" once you have finished this manual.

ADDITIONAL READING **baZic** manual Introduction.

What is a Program?

OBJECT

To instruct the student in how to recognize and write a simple program with the correct syntax, to understand the meaning of the words "statement" and "argument", and to recognize and use the **baZic** programming instructions LIST, PRINT, RUN, and SCRatch.

LESSON

A program is a series of instructions which tell the computer what it should do. One such instruction under the programming language **baZic** is called a statement. The most simple **baZic** program is a program with one statement. A LISTing of one such simple program would appear as follows:

```
10 PRINT "This is probably the simplest program"
```

In this instance the program has a line number of "10" and the single statement "PRINT" with an argument of '"This is probably the simplest program".' Under **baZic**, each step of the process of executing the program must be controlled by a line number.

Since statements are alwasys executed in line number order, line numbers guide **baZic** through the program so that the steps of the program are executed in the right order. The line number could be any number in the range of 0 to 65535 but 10 is where most programmers start numbering.

The first and only statement of the sample program is the reserved word PRINT followed by the information (enclosed in quotes) you want the program to print. A reserved word (like PRINT) is a word reserved by **baZic** to mean an instruction for it to execute.

Many of the instructions (statements) are followed by an argument for the statement to "work on." In our example program, the PRINT statement is used to cause the message, "This is probably the most simple program," to be printed on the terminal by the program.

If this program is RUN, the following will appear on the terminal:

```
This is probably the simplest program
```

Notice that the quote marks do not appear in the printout because quotation marks are used to delineate the bounds of the message to be printed. This message is the argument to the PRINT statement. Not all statements have arguments and many statements can have more than one argument.

If the reserved word PRINT was spelled incorrectly or one of the quote marks was left out, the program would not have the correct syntax. Syntax is the correct representation of a program so that the computer "knows" what action the program is to perform. For the syntax to be correct, the statement must match the syntax of the given language. Items which affect syntax can be spaces, commas, spelling, and quote marks, etc.

To write this program we would start by looking for the READY message of **baZic.** This prompt is used to tell the programmer that the language is READY to take instructions. To make sure there are no instructions already in **baZic,** we will first type the command to SCRatch any previous program. This sequence would appear as follows:

```
READY
SCR <CR>
READY
```

Now we may begin directly under the last READY message by typing the following:

```
10 PRINT "This is probably the simplest program"
```

Type everything exactly as you see it above (you should not indent since this is used to separate the computer sessions from the text). Once you have entered this information, you may RUN the program by typing:

```
RUN <CR>

This is probably the simplest program
READY
```

When you typed RUN, the program immediately executed the program, printed the READY message, and returned to the command mode so that you could enter more program or RUN the program again.

If you now wanted to view the program, you could simply type the command LIST and the program would be displayed by the computer as follows:

```
READY
LIST <CR>
10 PRINT "This is probably the simplest program"
READY
```

We now know that a program is RUN to make it execute. We can LIST the program if we want to see what it looks like. To erase a program we type SCR, which is an abbreviation for SCRatch. PRINT is a **baZic** statement which is used to print messages on the CRT.

ADDITIONAL READING

baZic manual Sections 2.1.1, 2.1.3, 2.3.1, and 3.2.1.

TEST

1. What is a reserved word?

   A reserved word is a special word used by **baZic** to mean some specific action (PRINT, etc.) to be taken by the interpreter.

2. What is a statement?

   A statement is a single instruction which tells the computer to do something for the programmer.

3. What is a program?

   A program is a set of instructions which is executed in a specific sequence to cause the computer to accomplish some task.

4. What is a line number?

   A line number is an integer number in the range of Ø to 65535 which is used to guide the order of execution of a program.

5. What is an argument?

   An argument follows a statement and is something for the statement to act upon.

6. What is the meaning of RUN?

   RUN is a command used to cause a program to begin executing.

7. What is the meaning of LIST?

   LIST is the command used when the programmer wants to view a LISTing of the program on the CRT.

8. What is the meaning of PRINT?

   PRINT is a **baZic** statement used by the programmer to cause some message which is delineated by quotes to be displayed on the CRT.

9. What is the meaning of SCR?

   SCR is the reserved word for SCRatch which instructs **baZic** to clear itself of any previous program that it might have contained.

10. What is the meaning of the word syntax?

Syntax is the way in which words and symbols are put together to form a program that will execute properly.

EXERCISES

Which of the following programs has the correct form and will work properly when run? If it will not RUN properly, tell why. You may use a computer to test each of the possibilities to see if it will execute correctly.

A. 1000 print "This is a program statement"
B. 0 PRINT "This is a program.
C. 100000 PRINT "This is a program"
D. 0 PRINT "This is a program"

Only Option D will work correctly. Option A will generate a SYNTAX error because the reserved word PRINT is not in uppercase (all capitol letters). Option B will also give a SYNTAX error because the information to be printed is not enclosed in double quote marks. Option C will return an OUT OF BOUNDS error since the line number is not within the proper range.

Write and run a program which will print the message "This is also a very simple program" on the CRT.

```
READY
10 PRINT "This is also a very simple program" <CR>
RUN <CR>

This is also a very simple program
READY
```

Write and RUN a program which prints the message "This is line one" and then prints the message "This is line two" on the next line.

```
10 PRINT "This is line one" <CR>
20 PRINT "This is line two" <CR>
RUN <CR>

This is line one
This is line two
READY
```

Working With a Program

OBJECT

To learn the use of the CAT, NSAVE, SAVE, LOAD, and PSIZE commands. Also, to learn the difference between internal memory (RAM) and external memory (disks) and how to recognize the current program.

LESSON

Internal memory, called Random Access Memory (RAM), is the working memory of the computer. The central processing unit (CPU) is closely associated with the internal memory and this is where all of the "work" is done. Internal memory is very fast in that the processor can access any part of the internal memory in a few billionths of a second.

Internal memory is called volatile because it requires an electrical current to retain its contents. This memory is said to be volatile memory since its contents disappear when the current is turned off. The current program is always located in the internal memory of the computer.

External memory retains its contents when the power is turned off; therefore it is called non-volatile. This memory is much slower to access but usually has room for much more information than the internal memory. Programs and data files are stored in the external memory. The computer has the ability to exchange information between the internal and external memory. A program becomes the current program when it is moved from the external memory to the internal memory.

Now that we have created a program (from Session 1), we may want to SAVE this program for future use. Since the internal memory of the computer is volatile and will "forget" when the power is turned off, we must store the program in non-volatile memory (floppy or hard disks).

Because the external memory can hold hundreds or even thousands of programs, we must have some method for organizing the programs on the disk so that they may be found easily. The CATalog of a disk is the listing of the names of all the programs (and data files) which are located on that disk.

If we wanted to look at a collection of names of the programs on a disk, we would look at the CATalog (sometimes called directory) of that disk. The proper command would therefore be to CATalog the disk and would appear as follows:

```
READY
CAT <CR>

BAZIC          20    54 D    1  0100            AF 0
TEST            4    56 D    2                  AF 0
READY
```

The CATalog indicates two files exist on this disk. The second file, called TEST, is the only program file. We can determine this by examining the third number from the left. The number 2 means that the file is a **baZic** program.

If you are using the CP/M version of **baZic**, your directory listing will look slightly different. The preceding example only shows one program listed, but there may be many programs on the disk. A CATalog under CP/M **baZic** would appear as follows:

```
BAZIC    COM TEST     002
```

The essential thing to learn from this discussion is that by issuing the CAT command, you are able to look at the names of the programs (and other files) on your disk drives.

If you do not have your program from the previous session, you will need to re-enter it at this time. Consult Session 1 if you are not sure how to enter the program. Remember to SCRatch any previous program. When you LIST your program, you should see the following:

```
READY
LIST <CR>
10 PRINT "This is probably the simplest program"
READY
```

If you have entered your simple program from Session I, you can now SAVE the program on the disk. At this point, we have one slight complication facing us. Because you have never before SAVEd this particular program, you will have to use the NSAVE command which stands for New SAVE. The first time a program is to be saved on a disk, you must use the NSAVE command followed by the name of the program. NSAVE causes a new file to be created for the program, while the SAVE command causes the program in memory to be copied to an existing file on the disk.

In this case we will use the name TEST for our sample program. To SAVE the program on the disk, the command sequence would appear as follows:

```
READY
NSAVE TEST <CR>
READY
```

On floppy disks, the command will take a few seconds to complete. On the hard disk, the command will be completed almost instantly. If the READY message is displayed directly after the command, we can assume that the operation was successful. Several error conditions can occur when this command is issued. These will be discussed later.

The next operation to learn is to LOAD a previously stored program back into memory. To do this, we must first erase the current program from the internal memory (RAM) by entering the SCRatch command. After SCRatching the program, we can make sure the program is gone by LISTing the program. The sequence would appear as follows:

```
READY
SCR <CR>
READY
LIST

READY
```

Since the LISTing shows no program, we are ready to bring the previously saved program off the disk back into memory. This is done by telling the computer to LOAD a specified program (in this case TEST). Use the following sequence as an example in LOADing the file TEST:

```
READY
LOAD TEST <CR>
READY
```

Again, the READY message indicates the operation was successful. If anything went wrong, an error message would be printed indicating which error occurred. To verify that the program has been successfully LOADed, type LIST to view the program. The sequence should appear as follows:

```
READY
LIST <CR>
10 PRINT "This is probably the simplest program"
READY
```

We may now modify this program and demonstrate the SAVE command. Type the following to add another line to your TEST program:

```
20 PRINT "This is now a more complex program" <CR>
```

If you now LISTed your program you would see:

```
READY
LIST <CR>
10 PRINT "This is probably the simplest program"
20 PRINT "This is now a more complex program"
READY
```

Since the file TEST is already created to hold the program TEST, you can use the SAVE command to reSAVE the TEST program. The NSAVE command is used only the first time you need to SAVE a program in order to create a file in which to SAVE the program. All additional SAVEs of the TEST program to the TEST file would appear as follows:

```
READY
SAVE TEST <CR>
READY
```

To verify that the program was saved correctly, repeat the previous sequence. First SCRatch the current program, LIST it to show that there is no program, LOAD the TEST program and then LIST it to show the program was LOADed from the disk into internal memory. The entire sequence would appear as follows:

```
READY
SCR <CR>
READY
LIST <CR>

READY
LOAD TEST <CR>
READY
LIST <CR>
10 PRINT "This is probably the simplest program"
20 PRINT "This is now a more complex program"
READY
```

You should understand at this point how a program can be modified and the resulting program SAVEd back into the same file. Of course any program residing in the file will be written over each time the SAVE command is used.

The next topic to be considered is how the size of a program is determined. When we CATaloged the disk, we saw the file name with several numbers directly after the file name. The first number is the size of the program file in blocks. Blocks are used by the computer to allocate space for disk files.

A block can hold 256 characters of information. If you wanted to see how large your program is, you could count each letter, number, space, or special character in you program and divide by 256 to arrive at an approximation of the size of the program in blocks.

If the program were large, this could be a very time consuming procedure. Since **baZic** has a command to do this task, we should take advantage of this feature. The command is PSIZE (Program SIZE). By issuing this command, **baZic** will inform us of the size of the current program in blocks. If we were to execute the command on our present program, the sequence and results would appear as follows:

```
READY
PSIZE <CR>
1 BLOCKS
READY
```

If you try to save a program which is too large for the space allocated on the disk, you will be given an OUT OF BOUNDS error. Your program will not be SAVEd. Of course this condition can be checked easily by executing the PSIZE command and comparing the results with the size number as shown by the CAT command. By comparing the two numbers, you can easily determine if your program will fit into the available space.

If your program will not fit into the existing space, you will have to NSAVE your program under another name. In future lessons you will learn how to DESTROY any files that you may not want.

ADDITIONAL READING

baZic manual Sections 2.2.1, 2.2.2, 2.2.3, 2.2.4, and 2.1.6.

TEST

1. What is the command used to determine the size of a program?

   PSIZE

2. What command lists a program?

   LIST

3. What command is used to save a new program?

   NSAVE <PROGRAM NAME>

4. What command is used to save a program in an existing file?

   SAVE <PROGRAM NAME>

5. What command is used to retrieve a program from the disk and place it in internal memory?

   LOAD <PROGRAM NAME>

6. If we wanted to erase a program from internal memory, what is the command?

   SCR

7. What is the current program?

   The current program is the program residing in the internal memory of the computer which has been LOADed from disk or entered by a programmer.

8. To determine the size of a program stored on disk, we use what command?

   CAT

9. What happens to internal memory (and your program) when the power is turned off to the computer?

   The memory forgets and your program is lost.

10. What happens to disk storage when the power is turned off to the computer?

    Nothing. All programs and data are preserved.

11. What is RAM?

    RAM is the internal Random Access Memory of the computer.

12. What is the CPU?

    The CPU is the Central Processing Unit which actually performs the work of the computer.

EXERCISES

Store the same program under two different names.

Practice writing a short program, NSAVEing it, LOADing it back, modifying it, PSIZEing it, SAVEing it back on disk, and then CATaloging the disk to see the program files listed and the size of each.

Line Numbers

OBJECT

To learn more about line numbers, how to have the computer generate them AUTOmatically, how to RENumber them, and how to DELete them.

LESSON

For the benefit of the programmer, **baZic** has the ability to generate line numbers without the user having to type the line numbers in while the program is being written. The AUTO command can be used to start the line numbering at any valid line number and increment by any value as long as the line number generated falls within the allowable range of line numbers (Ø to 65535).

The simplest case of the AUTO command is to type AUTO without any arguments. This issuance of the command will cause line numbers to be generated starting with line number 1Ø and incrementing by 1Ø for each additional line number required. To use the AUTO command, type AUTO in response to the READY prompt as shown below:

```
READY
AUTO <CR>
1Ø PRINT "This is probably the simplest program" <CR>
2Ø
```

The cursor will appear directly after the first line number (1Ø). You may now enter your program statement. When you enter a Return (<CR>) at the end of each statement, **baZic** will enter the next line number for you AUTOmatically. To terminate the AUTO mode, enter a Return immediately after **baZic** generates the line number.

If you want your program to start with line number 1ØØ, use the AUTO command and the starting line number argument. This time we would pass a single argument to the command as follows:

```
READY
AUTO 1ØØ <CR>
1ØØ PRINT "This is probably the simplest program" <CR>
11Ø
```

The next line number generated would be 11Ø. If we wanted the line numbers to be 1ØØ apart (1ØØ,2ØØ,3ØØ, etc.), we would use the AUTO command and pass two arguments (starting line number and incremental value). The following example shows this situation:

```
READY
AUTO 100,100 <CR>
100 PRINT "This is probably the simplest program" <CR>
200
```

The RENumber command is very similar to the AUTO command except that the program has already been created when we use the RENumber command and we just want to RENumber the line numbers. In many programming situations, a program is written using the AUTO command to generate line numbers which start at 10 and increment by 10. Since programmers are not perfect (just program awhile and you will find out this is true), usually one or more line numbers will have to be added to the program so that the program will execute correctly.

Once the program is completed and working correctly, we will probably want to "clean it up" by RENumbering the program so that all of the line numbers are separated by the same incremental value. To RENumber a program starting at line number 10 and incrementing by 10, issue the REN command as follows:

```
READY
REN <CR>
READY
```

baZic will now RENumber your program to the new specification. If you are in doubt, recall (LOAD) your program from the previous Session, RENumber it and LIST it to see the affect. Programs may also be RENumbered to any starting value and any incremental value as in the AUTO command (REN 18,27). If we want to RENumber a program so that the first line number is 100 and the incremental value is 5, we would issue the command as follows:

```
READY
REN 100,5 <CR>
READY
```

The DELete command works in a similar manner to DELete line numbers, except the arguments must always be supplied. Issuing the DEL command without arguments would result in **baZic** printing a SYNTAX error. Remember, if you want to DELete all the line numbers (and the associated program line), use the SCRatch command to delete the entire program. The DELete command is issued with the number of the first line you want DELeted followed by the number of the last line you want DELeted.

Enter this sample program:

```
10 PRINT "This is line one"
20 PRINT "This is line two"
30 PRINT "This is line three"
40 PRINT "This is line four"
50 PRINT "This is line five"
60 PRINT "This is line six"
```

To DELete lines 30 through 50, inclusive, issue the command as follows:

```
READY
DEL 30,50 <CR>
READY
```

Now LIST the program and you will find that the specified lines are not in the current program. The LISTing will now appear as follows:

```
10 PRINT "This is line one"
20 PRINT "This is line two"
60 PRINT "This is line six"
```

For further practice, RENumber this program to start at 100 and increment by 50. LIST the program. The sequence would be:

```
READY
REN 100,50 <CR>
READY
LIST <CR>
100 PRINT "This is line one"
150 PRINT "This is line two"
200 PRINT "This is line six"
READY
```

To delete a single line only, type the line number followed immediately by a Return. The line number and line will no longer be in the program. Using the preceding LISTing, type:

```
READY
150<CR>
READY
LIST <CR>
100 PRINT "This is line one"
200 PRINT "This is line six"
```

Notice that line 150 is now removed from the program. If we had SAVEd our original program on disk, we would need to SAVE it again if we wanted to retain this final version.

ADDITIONAL READING

baZic manual Sections 2.1.2, 2.1.4, and 2.1.5.

TEST

1. What command would be used to renumber a program so that the first line number in the program is 35 and each additional line number is 5 greater than the previous line number?

   REN 35,5

2. How do you delete a single program line?

   Type the line number followed immediately by a Return.

3. What command is used to cause **baZic** to generate line numbers automatically?

   AUTO

4. Show the AUTO command to start numbering at 1500 with an incremental value of 55.

   AUTO 1500,55

5. Show how to delete line numbers 1 to 15.

   DEL 1,15

EXCERSISES

Practice using the AUTO, REN, and DEL commands.

Commands, Statements, & Functions

OBJECT

To learn to recognize the difference between the commands, statements, and functions of **baZic** and to be able to recognize, DIMension, and use a **baZic** variable.

LESSON

baZic has two major modes: direct (or command) mode and program execution mode. Programs and commands are always entered in the direct mode while programs run in the program mode. Commands (such as LIST, RUN, SCR, etc.) may be executed in the direct mode but not in the program execution mode (in a program).

Normally statements are executed in the program mode which is initiated by the RUN command. Some statements will work in the direct mode and can be used for such things as calculations, etc. The statements which can be used directly are listed in the **baZic** reference manaul in Section 2.

You should, by now, be familiar with the commands of **baZic** since you have used most of the commands in the previous sessions. A command is always issued by the programmer while in the direct mode. The command is issued by typing the command reserved word followed by a Return (<CR>). The command is executed immediately.

A program is made by combining a series of statements . Although **baZic** has many statements, only one has been covered (PRINT). More will be learned about the other statements as each one is covered individually in later sessions.

The built-in functions may be a little harder to learn unless you have experience with mathematical functions (trig functions, etc.). A built-in function is a predefined operation which is controlled by and operates on one or more arguments and which returns, upon completion, a value based on the operation of the function.

As an example of functions, if we wanted to find the square root of some number, we would use the SQuare RooT (SQRT) function. The following sequence would cause the computer to PRINT the square root of the number 4:

```
READY
PRINT SQRT(4) <CR>
2
READY
```

The number 4 is the argument to the function and is enclosed in parenthesis. This is the value which we want the function to work on. The value returned from the function call is the number 2 which is printed by the PRINT statement associated with the function call. A function is called, or invoked, by using the function name in a statement.

baZic has many "built-in" functions, that is, functions which are contained in the **baZic** interpreter. **baZic** also has the ability to let the user "write" his own function, give it a name, and then use it as needed within a program. Several more Sessions will be devoted to functions because they are a very powerful and important part of **baZic.**

The next topic to be covered is variables. Since much of the information **baZic** deals with is constantly changing, each symbolic "piece" of information is given a name to reference that information. One such piece of information could be an accounts receivable total, a general ledger account number, a person's name, and so on.

baZic uses a strict method of naming variables. All variable names must begin with a letter of the alphabet and all letters must be uppercase only. The basic variables are the letters A to Z. Alternately, each letter may be followed by a number from Ø to 9. Obviously, this allows many variable names, but there are even more as we will find out later. Some sample legal variable names are:

A A1 C3 XØ B9 D E F5 R2 Y Z

These variable names are called numeric variables because each variable name can store one number. A string variable is used to store any other character which appears on the keyboard. A string variable name can have the same combination of letters and numbers but always has a dollar sign ($) on the end of the name. When you see the dollar sign, you should say the word "string". The variable A$ would be called "A string." Some sample legal string variable names are:

A$ A1$ C3$ XØ$ B9$ D$ E$ F5$ R2$ Y$ Z$

An additional type of variable is called an array variable. Even though there are 26Ø (26 letters times 1Ø numbers per letter) numeric variables, it is not uncommon for a **baZic** program to have more than 26Ø different numbers with which it has to work. To handle this situation, we would want to use an array variable.

Array variables have the same name as the numeric variables except the name is followed by parenthesis containing a subscript. The subscript "points" to the proper element of the array and is used to reference the variable within the array that we want. The subscript may be a variable itself, giving almost unlimited ways to access any element of the array.

Array variables use one variable name to signify an entire group of numeric variables. As an example, the array variable A( ) could represent 10 or more variables. A(1) would represent the first variable, A(2) could represent the second, and etc. As you can see, the number of variables which **baZic** can accommodate is enourmous.

To the computer, a variable is actualy a name for a place in the internal memory that is going to be used by the program to "hold" a particular piece of information whose value will change as the program executes. That is why it is called a variable.

Since each variable requires some amount of "room" to "store" its information, in many cases we must "tell" **baZic** how much room to reserve for each variable. This process of "telling" **baZic** is called DIMensioning the variable. Only string and numeric array variables need be DIMensioned. If you use either variable without first DIMensioning the variable, **baZic** will assign a standard DIMension of 10 to a string variable and 11 to an array variable.

If a string variable has a DIMension of 10, that means that 10 letters, numbers, or special characters can be stored in that string. The string "ABCDEFGHIJ" could be stored in an un-DIMensioned string variable, but the string "ABCDEFGHIJK" would not fit because it has too many letters. The last letter "K" would not be assigned to the variable. To make the "K" fit, we would have to DIMension the string to 11 before we use the variable in the program. The following shows the DIMension of the A$ variable to 11:

```
10 DIM A$(11)
```

The DIMension statement must always be before the first use of the variable in a program. Once a variable is used, it's DIMension is set to the default (10 for a string and 11 for a numeric) and cannot be changed within the program.

In a similar manner, the numeric array variables must be DIMensioned before their use. As stated before, the DIMension of an array variable is set to 11. By simply using a numeric array variable in a program, you get 11 "places" in the variable to store a number. These 11 places are referenced by using subscripts ranging from 0 through 10. If we wanted to allow 100 numerics to be stored within the A array variable, we would DIMension the variable as follows before the variable is used in the program:

```
10 DIM A(100)
```

This tells **baZic** to reserve room for 100 (actually 101 counting the array element 0) numbers to be stored within the A array variable. Each number stored within the array is called an element of the array. The elements are now referenced by subscripts from 0 to 100.

In case you need even more variables, numeric arrays can have more than 1 dimension and those dimesnsions can be as large as available memory. A two-dimension array could be visualized as a table having columns and rows of numbers. The entire table would be stored in the single array. An example would be:

```
10 DIM A(100,10)
```

This array could store 1000 numbers (actually 1111 or 101 times 11). To reference an element of this array, you would have to state the element you want to access by giving its row and column postion within the array. You might want to access element A(21,5), or A(99,9), or any other of the 1000 possible combinations.

A three dimensional array would be:

```
10 DIM A(10,10,10)
```

This array would again hold 1000 numbers (really 1331 or 11 times 11 times 11). To access one element, you must specify the position in the array giving each of the three dimensions. Care must be used in creating large arrays since the last example would use almost 8000 bytes of internal RAM storage.

The user-defined functions mentioned earlier in this Session are given names which are similar to variable names. Functions can be string or numeric functions as reflected by their names and as determined by the type of data the return (string or numeric). The function name is composed of the letters "FN" followed by a numeric or string variable name. Some legal function names would be:

```
FNA     FNA$     FNB1     FNB1$     FNX     FNY$
```

Functions will be covered in much greater detail in later Sessions.

TEST

1. What is the difference between the command mode and the direct mode of **baZic**?

   None. The command and direct mode are the same and are used by the programmer to issue commands directly to **baZic** as well as to actually write of a program. The other mode of **baZic** is the program execution mode.

2. Define command.

   A command is a **baZic** reserved word which causes the computer to take some immediate action as specified by the programmer.

3. Define statement.

   A statement is a single instruction contained within a program which causes the computer to perform some operation when a program is executing. Some statements also can be used as commands.

4. Define function.

   A function is an operation, either built-in to **baZic** or defined by the programmer, which operates upon one or more arguments to obtain a result which is passed from the function to the part of the program which called the function. Functions are named similar to variables.

5. What is a numeric variable?

   A numeric variable is a legal symbolic name used by **baZic** to store a number.

6. What is a string variable?

   A string variable is a legal symbolic name used by **baZic** to store any character, or group of characters, which appear on the terminal keyboard.

7. What is a numeric array variable?

   A numeric array variable is a legal symbolic name used by **bazic** to store a series of related numeric variables.

8. What is the standard DIMension of a string and numeric variable?

   1Ø for a string and 11 for a numeric.

9. How many dimensions can a numeric array contain?

   As many as you want, provided your computer has enough memory to support the array.

10. How many times can you DIMension a variable within a program?

   A variable can be dimensioned only once in a program. Any use of a variable before it is dimensioned causes the variable to be dimensioned to its default value.

11. What is a subscript?

   A subscript is a reference to a particular element in an array.

EXERCISES

Which of the following sets of variables are legal variables?

A.	bl	C3	D4$
B.	FR	D9	H$
C.	Hl	M6	8B
D.	X$	Y0	O0

Only D contains all legal variable names. In Option A, bl is not legal because the letter is not in uppercase. Option B is not valid because FR is not a legal name. Option C can be excused because the number precedes the letter in 8B. The last variable under Option D looks wrong but is correct since it has one letter "O" followed by the number "0."

Getting Information In and Out

OBJECT

To teach the student how to use the basic methods for exchanging information between the terminal and the internal memory of the computer by using the PRINT, !, INPUT, and INPUT1 statements.

LESSON

In the first session, we had some experience with the PRINT statement. Now we will learn more about the PRINT statement and some of its variations.

LOAD your test program (or re-enter it if you do not have it stored). You should see the following upon LOADing and LISTing the program:

```
READY
LOAD TEST <CR>
READY
LIST <CR>
10 PRINT "This is probably the simplest program"
20 PRINT "This is now a more complex program"
READY
```

If the program is RUN, the following would appear on the terminal:

```
This is probably the simplest program
This is now a more complex program
READY
```

For a change, we might want the program to PRINT the same messages but want them separated by a single line. To accomplish this change, enter the following line:

```
15 PRINT <CR>
```

When the program is LISTed, it will appear as follows:

```
10 PRINT "This is probably the simplest program"
15 PRINT
20 PRINT "This is now a more complex program"
READY
```

Now if the program is RUN, the results would be:

```
This is probably the simplest program

This is now a more complex program
READY
```

Notice that the two lines are separated by one blank line (a line space). For each PRINT statement which contains no argument (item to print), a blank line is printed.

Now let us add to line 15. By entering a line number which is the same as a line number which already exists, the new line takes the place of the old. Enter the following:

```
15 PRINT "For sure ",
```

Be sure you enter the line exactly as shown including spaces. If the program is LISTed, you should see:

```
10 PRINT "This is probably the simplest program"
15 PRINT "For sure ",
20 PRINT "This is now a more complex program"
READY
```

If this program is RUN, you will see the following on your terminal:

```
This is probably the simplest program
For sure This is now a more complex program
READY
```

As you can see, the effect of the comma was to keep the computer from printing the carriage return at the end of the PRINT statement. Normally a PRINT statement causes a carriage return and a line feed to be issued to the terminal. When you want to have two PRINT statements print their message on the same line of the terminal, include a comma at the end of the first PRINT statement and any additional PRINT statement will print on that same line.

In summary, a PRINT statement always prints a carriage return and line feed after each use of the PRINT statement, except when the statement is followed by a comma.

Since most programmers are lazy by nature, **baZic** includes a shorthand notation for the reserved word PRINT. This shorthand statement is the exclamation mark (!). Anywhere you see the reserved word PRINT, you may substitute !. If the preceding program were SCRatched and entered again using ! instead of PRINT, it would work exactly as before and would appear as follows when LISTed:

```
10 ! "This is probably the simplest program"
15 ! "For sure ",
20 ! "This is now a more complex program"
READY
```

Most programmers use the reserved word PRINT when they are first starting, but switch to ! when they get familiar with programming because fewer keystrokes are required to enter the reserved word.

The next item to be learned is how to INPUT information from the terminal to a variable in internal memory. You sharpies will have probably already figured out that we use the INPUT statement. To use the INPUT statement, enter a line number, the INPUT reserved word, and the name of a variable you want the information passed to.

SCRatch any previous program and enter the following:

```
READY
SCR <CR>
READY
10 INPUT A <CR>
20 PRINT A <CR>
```

Upon LISTing the program you will see:

```
READY
LIST <CR>
10 INPUT A
20 PRINT A
READY
```

If you RUN this program, you will see:

```
READY
RUN <CR>

?25 <CR>
25
READY
```

The question mark indicates that **baZic** is waiting for you to enter something, or in other words, the INPUT instruction is being executed. In this case, the value entered must be a number since the variable to be INPUT is A, a numeric variable. In the example, we show the number 25 being entered by the user and then the program PRINTs the number on the next line.

In analyzing this program we see that the INPUT statement is used to assign a value to the variable A. The program, upon encountering this statement, displays a question mark on the terminal and waits for the user to enter a number. When the number is entered, the program places that number in the assigned variable (numeric variable A). The next statement in the program says to PRINT the value of the variable A. Because A has been set to the value entered by the user, the program will PRINT the same number entered.

The INPUT statement has several variations. The first variation is the INPUT1 statement. The only difference between INPUT and INPUT1 is that INPUT1 suppresses the carriage return and line feed normally issued by the INPUT statement when the user makes his entry.

If the previous program is changed to use the INPUT1 statement instead of the INPUT statement the LISTing will appear as follows:

```
READY
LIST <CR>
10 INPUT1 A
20 PRINT A
READY
```

If you RUN this program, you will see:

```
READY
RUN <CR>

?25 <CR> 25
READY
```

Notice that the variable A is now PRINTed on the same line as it was INPUT.

One last case combines a PRINT statement and an INPUT statement into one. This combination is used to let the person using the program know what you want them to enter. SCRatch any previous program and enter the following:

```
READY
SCR <CR>
READY
10 INPUT "Enter a number ",A <CR>
20 PRINT A <CR>
```

Upon LISTing the program you will see:

```
READY
LIST <CR>
10 INPUT "Enter a number ",A
20 PRINT A
READY
```

If you RUN this program, you will see:

```
READY
RUN <CR>

Enter a number 25 <CR>
25
READY
```

Notice that the question mark has been replaced by the prompt message "Enter a number " which tells the user what is expected of him.

Strings also may be entered by using a string variable with the INPUT statement. Follow the sequence listed below to write a program which will enter string information:

```
READY
SCR <CR>
READY
10 INPUT "Enter a string ",A$ <CR>
20 PRINT A$ <CR>
```

Upon LISTing the program you will see:

```
READY
LIST <CR>
10 INPUT "Enter a string ",A$
20 PRINT A$
READY
```

If you RUN this program, you will see:

```
READY
RUN <CR>

Enter a string Mike <CR>
Mike
READY
```

As you can see, INPUTting a string is very similar to INPUTting a number. This program provides a good demonstration of the DIMension principle as described in a previous session. Try running this program again, but this time enter a string which is more than 10 characters long. You will see that when the string is printed, only the first ten characters will show. To correct this problem, enter the following line:

```
5 DIM A$(80)
```

Upon LISTing the program you will see:

```
5 DIM A$(80)
10 INPUT "Enter a string ",A$
20 PRINT A$
READY
```

Now you can RUN the program and enter a string of any size up to 80 characters before **baZic** "cuts" the string off.

If your INPUT statement is expecting a string and you enter a number, the INPUT statement will accept the number as a string. However, if you try to enter a string when the INPUT is expecting a number, **baZic** will print the message:

```
INPUT ERROR--RETYPE THE INPUT
```

The program will then patiently wait for you to enter the correct type of information (a number).

We are now to the stage of beginning to write useful programs. You should now have a basic understanding of how to collect information from the person using the program and how to PRINT that information back to the terminal. Writing useful programs involves these steps, as well as many others. Generally, after information is entered using an INPUT statement, it is stored, added, subtracted, and otherwise manipulated to get the results you want before it is PRINTed.

ADDITIONAL READING

baZic manual Sections 3.2.1, 3.2.2, and 3.2.3.

TEST

1. How do you make **baZic** print a line space (blank line)?

   Use the PRINT (or !) statement without an argument or a comma.

2. How do you make two print statements print on the same line?

   Follow the first PRINT statement argument with a comma. A comma causes the carriage return and line feed to be suppressed.

3. What is the shorthand notation for the reserved word PRINT?

   !

4. What statement is used to gather information from the user of a program?

   INPUT

5. Which input statement is used to suppress the carriage return and line feed normally generated by an input statement?

   INPUT1

6. What is the standard prompt printed by an input statement?

   ?

7. Show a sample input statement which prints a prompt instead of the standard question mark.

   INPUT "This is a prompt", A$

8. What distinguishes a numeric input from a string input?

   The variable name. If the variable name is numeric, a number is to be entered. If the variable name is a string name, a string is to be input.

9. What happens if you enter a number in response to a string INPUT?

   The number is entered as a string. This means that it cannot be used in any arithmetic calculation without first converting the number to a numeric variable.

10. What happens if you enter a string in response to a numeric INPUT?

    baZic will print an error message and wait for a number to be entered.

EXERCISES

Write a program which will input a person's name, age, and state and then print the results.

```
10 DIM A$(50)
20 INPUT "Enter your name ",A$
30 INPUT "Enter your age ",A
40 INPUT "Enter your state ",B$
50 PRINT "Your name is ",
60 PRINT A$
70 ! "Your age is ",
80 ! A
90 ! "Your state is ",
100 ! B$
```

Line Editor

OBJECT

To learn the use of the **baZic** line editor and EDIT command.

LESSON

At this time we will digress for one lesson to learn the line editor instead of actual programming. By now, you should be tired of typing and re-typing program statements. The line editor will not relieve you of the task of typing programs into **baZic**, but it will help a great deal in changing lines already entered.

Start by LOADing (or typing if neccessary), the program from previous Sessions. A LISTing should appear as follows:

```
10 PRINT "This is probably the simplest program"
20 PRINT "This is now a more complex program"
READY
```

In a previous session, we changed the PRINT statement to its abbreviated form (!). Before, we had to re-type the entire line to make a single change. Now we will use the line editor to simplify this task. With the above program residing as the current program, enter the editor by typing:

```
READY
EDIT 10 <CR>
10 PRINT "This is probably the simplest program"
```

As you can see, the editor has now displayed the line you told it you wanted to edit. The cursor will be under the "1" of the line number.

All of the editor commands are invoked by holding the Control Key while pressing the specified letter. The Control Key is much like a second Shift Key in that pressing the Control Key and another character cause the character key to "mean" something different. In this case it means it is a command to the line editor.

Now to edit line 10. Press the Control Key and the A key. Notice that a 1 is printed directly below the 1 in the line number of the line to be edited. Press the same combination (Control A) again, and the 0 will be printed. Press the same combination one more time and the space will be "displayed" and the cursor will be under the P of the PRINT statement. So what does Control A do? It prints one character from the line to be edited to the new line we are forming directly below the old line.

The display should now look like this:

```
READY
EDIT 10 <CR>
10 PRINT "This is probably the simplest program"
10
```

Now enter an exclamation mark (the shorthand for PRINT). The P should be overwritten by the exclamation mark and the cursor should now be below the R of PRINT.

The display will now appear as follows:

```
READY
EDIT 10 <CR>
10 PRINT "This is probably the simplest program"
10 !
```

The "RINT" needs to be deleted, so type Control Z four times. Control Z is used to delete characters from the old line so that they do not appear in the new line. A percent sign (%) is displayed for each character that is deleted. The display should now appear like this:

```
READY
EDIT 10 <CR>
10 PRINT "This is probably the simplest program"
10 !%%%%
```

From this point on, a "^" will be substitued for the word "control". When you see the symbol "^A", it means to enter a Control A (press the A key while holding down the Control key).

To complete the edit, type ^G. The ^G command is used to "copy to the end of the line." Everything from the character at the cursor position to the end of the old line will be copied from the old line to the new line. The display will now show:

```
READY
EDIT 10 <CR>
10 PRINT "This is probably the simplest program"
10 !%%%% "This is probably the simplest program"<CR>
```

To enter the new line as a line in the program, you must follow the ^G command with a carriage return. A carriage return terminates the edit in that the new line now becomes the old line and is placed in the program.

If, for some reason, you didn't want the edited version of the line to be placed in the program, you could enter a ^N. The ^N command is the cancel command. The cursor will advance to the next line below the line being edited as if a Return had been entered, except the old line remains in the editor for further editing.

If you want to see the entire line as it has been edited, you may enter the ^G command again and a Return or ^N. The display will now show:

```
READY
EDIT 10 <CR>
10 PRINT "This is probably the simplest program"
10 !%%%% "This is probably the simplest program"<CR>
10 ! "This is probably the simplest program"<CR>
```

The line has now been edited. This procedure may sound like it takes a long time and maybe is not worth the effort. This entire procedure can be executed in a matter of seconds, once the programmer becomes accustomed to using the line editor.

As a short review, we have now learned four editor commands; ^A copies one character from the old line to the new, ^Z deletes one character from the old line so that it does not appear in the new line, ^N cancels the new line being edited, and ^G copies all characters from the current cursor position to the end of the line.

In the next exercise, we shall edit line 20 and use a new command. Type:

```
READY
EDIT 20 <CR>
20 PRINT "This is now a more complex program"
```

In the previous example, we typed ^A until the cursor moved under the P of the PRINT statement. This time, we will let the computer search for and find the P. The command is ^D. When you enter the ^D command, nothing will happen. **baZic** is now waiting for you to type another letter, the letter for which you want it to search. Enter a P and the cursor should move to locate itself directly under the P of the PRINT statement.

The ^D (search) command searches for the first occurrence of the specified character. If the character appears more than once, you must execute the command the proper number of times to get to the correct character. Many times you can search for some other character which appears only once before the character you want and then search for the desired character. If the character you specified does not appear in the line, the editor will "ring the bell" of your terminal to tell you that you made a "dumb" mistake.

Using the same sequence as before, enter a "!" below the P, type four ^Z's to erase the RINT, type ^G to copy to the end of the line, and enter Return to terminate the edit. To view the line again, enter the ^G command again followed by a Return. The entire sequence should appear as follows:

```
READY
EDIT 20 <CR>
20 PRINT "This is now a more complex program"
20 !%%%% "This is now a more complex program"<CR>
20 ! "This is now a more complex program"<CR>
```

The line editor has two more commands: ^Y which is the insert command, and ^Q or Backspace which is the backspace command. The ^Q (or you can use the Backspace key) command is used to backup while editing.

The ^Y command is more complex as it is used to toggle the insert mode on and off. As a practice, let's insert the word "much" between "a" and "more" of line 20. Begin by entering the edit mode by typing:

```
READY
EDIT 20 <CR>
20 PRINT "This is now a more complex program"
```

Now search for the letter "m". Enter ^D and a lowercase "m" as the character to search for. The cursor should advance to below the letter m of the word more. We are now in the correct position to insert the new word. Enter ^Y and the editor will print a less than symbol (<) to tell you that the text which follows is being inserted. Now type "much ". Be sure to type a space after the word "much." Enter another ^Y to exit the insert mode and the editor will print a greater than sign (>) to indicate the insert mode is off. Enter a ^G and a Return to complete the edit. The entire sequence would appear as follows:

```
READY
EDIT 20 <CR>
20 PRINT "This is now a more complex program"
20 PRINT "This is now a <much >more complex program"<CR>
```

By typing ^G and Return (or ^N) again, you can see the edited line:

```
20 PRINT "This is now a much more complex program"
```

You should now know the editor commands but practice is the only way in which you can become proficient in using the editor commands.

There are two addtional times you can use the line editor without having to type EDIT Line#. When you have entered any line of text into a program, that line is in the editor and can be edited immediately. All of the edit commands are available. The other situation is the user response to an INPUT statement. The user may use the line editor to edit any response to any INPUT statement.

3.5.9 END a program

END

The END statement is similar to the STOP statement except the END statement causes the program to terminate to the direct mode with no recourse to CONTinue. END need not be the last line in baZic as with many other BASICs since baZic will assume an END statement when the last line of a program is executed.

The END statement may occur anywhere in a program and will not cause the program to "end" unless the END statement is executed. There may be any number of END statements in a program, or there may be none. There is an implied END statement at the end of the program.

Examples of the use of the END statement follow:

```
100 END
120 IF A$="END" THEN END
```

Branching

OBJECT

To learn the elementary branching techniques available in **baZic** and the statements GOTO and ON GOTO.

LESSON

If no branching capabilities within **baZic** were available, programming would indeed be a chore. Every single operation would have to be defined explicitly for each occurrence of the operation. Programs would have many line numbers and would be very large.

Luckily, this is not the case. By using a simple GOTO statement, control can be passed to any valid line number in your program. As an example, enter the following program:

```
10 DIM A$(50)
20 INPUT "Enter your name ",A$
30 PRINT A$
40 GOTO 20
```

Everything should be familiar to you except line 40. Line 40 says to continue processing at line 20. If this program is RUN, the user will be asked for his name continually, the name will be printed, and the question will be asked again. This is called an endless loop since there is no provision for ending the program. RUN the program to see the results.

There is one method available to STOP the program. While the program is RUNning, press Control C (^C). The computer will print STOP IN LINE 20, or whichever line you were executing when the ^C was executed. ^C is called the panic button, since it allows you to stop whatever is going on.

The GOTO is the simplest case of a branching statement. In the next few sessions, we will learn more. The GOTO statement does have a twin statement which is very similar. This statement is the ON GOTO statement which allows the user to branch to more than one line number from one statement. SCRatch the previous program and enter the following:

```
10 INPUT "Enter a number from 1 to 3 ",A
20 ON A GOTO 30, 50, 70
30 PRINT "This is option 1"
40 GOTO 10
50 PRINT "This is option 2"
60 GOTO 10
70 PRINT "This is option 3"
80 GOTO 10
```

In this example, line 10 is used to INPUT a number. Hopefully, the user will enter a number between 1 and 3 since this program does no check on the validity of the user's response. If the user does not enter a value in the proper range, line 20 will generate a SYNTAX error. You may want to SAVE this program for the next lesson.

Line 20 shows the use of the ON GOTO statement. Depending on the value of A (1, 2, or 3), line 20 will branch to line 30, line 50, or line 70. If A is equal to 1, processing will continue at line 30. If A is equal to 2, processing will continue at line 50, etc. Notice that each option will print a message indicating which option was selected and is followed by a GOTO statement to line 10. This program is also an endless loop and can only be terminated by using ^C.

These two examples show the basic use of the GOTO and ON GOTO statements. The GOTO statement can be one of the most powerful features of **baZic** but also has the ability to be one of the worst features. The problem resides in the fact that the GOTO statement of any BASIC interpreter gives the programmer too much flexibility to branch to any point in a program. This often results in a program which has no structure.

When a program is written, the programmer should take the time to divide the program into modules, each of which does one particular task, has one entry point and one exit point. These are the basic premises of structured programming which all programmers, new or old, should adhere to. GOTOs should be used only at the end of modules to return to some controlling module, or within the module to branch to internal points or the exit point. Many problems can arise when the programmer tries to branch out of the middle of one module into the middle of another module.

This is not saying that the program won't work. Many times the program will work, even if the logic is overly complex because of unneccessary branching. The worst problem with unneeded GOTOs is the program becomes very hard to follow if it is branching in every direction constantly. This is hard on not only the programmer who wrote the program, but any other individual who tries to understand the program.

The best approach is to research the task to be performed by the program and divide the program into the proper modules, each designed to accomplish only its specific task. GOTOs are not inherently bad, but should be used sparingly in the appropriate places. One of the worst temptations to new programmers is to use GOTOs without reguard to the guidelines previously stated.

The ON GOTO statement is fine to use since it allows an orderly transfer of program control to different sections of the program. However, the ON GOTO should branch to the beginning of a module so as to preserve the concepts of structured programming.

ADDITIONAL READING

baZic manual Sections 3.3.1 and 3.3.4.

TEST

1. What is an endless loop?

   A program which has no provisions for termination, and which, by appearance sake, will continue forever.

2. How do you activate the panic button?

   Press the Control and the C key simultaneously.

3. What is structured programming?

   Programming where program modules are written to perform each individual task, each module having only one entrance and exit point, and where all modules are arranged in a logical fashion so that the program has continuity and order as a whole.

EXERCISES

Practice writing programs using the GOTO and ON GOTO statements.

Relational Operators

OBJECT

To learn the meaning of =, <, >, <=, >=, <>, LET, REM, and the words constant and operator.

LESSON

Everyone knows the meaning of equal--if two items are the same in every way they are said to be equal. The concepts to be discussed in this session are similar to the equal concept. While programming in **baZic**, variables and constants will be compared to determine if they are equal. Variables also will be set equal to other variables or constants by use of the LET statement. If variables are not equal, they can be larger than or less than other variables or constants.

The symbols used to compare variables with other variables or constants are called the relational operators. Usually a statement consists of an operator (which specifies the action) and an operand (which receives the action). In this section, all of the operators are relational since they are used to determine the relationship between variables and variables or variables and constants. Operators may be of other kinds (arithmetic and Boolean), and are used to "operate" on the variables. The variables would be called the operands.

To set a variable equal to another, use the LET statement as follows:

```
10 LET A=B
```

This statement sets the variable A so that it is equal to the value stored in variable B. A variable may also be set to a constant, as below:

```
10 LET A=15
```

In this case, the variable A will contain the constant 15 upon the execution of line 10. Most programmers only use the reserved word LET when they first begin programming. Like the PRINT statement, there is a shorthand for the LET statement. The shorthand is to assume the word LET by leaving it out entirely. The previous example would appear as follows in the shorthand notation:

```
10 A=15
```

As you can see, variables can be set to any value that you might want and can also be changed as many times as you want.

One of the problems in using variables is that it is easy for the programmer to forget what each symbolic variable name represents. One way to solve this problem is the use of the REMark statement. What does the REMark statement do? Nothing!

Everything appearing to the right of the REMark statement is unconditionally ignored by **baZic.** The REMark statement is used to place messages within a program to help remind the programmer of things easy to forget. The last example could be helped by a REMark statement, such as:

```
10 A=15\REM A IS THE NUMBER OF ACCOUNTS
```

Notice that before we can use the REMark statement after another statement which has the same line number, we must use a statement separator symbol which is a backslash (\). The REMark statement should be the most used statement in any program, because it helps the programmer, and anyone else who has to work with the program, understand and remember how the program works.

Now to return to the relational expressions. You should know the meaning of equal, but what about less than or greater than? These relations are as they sound and are represented by the symbols < and >, respectively.

If you have trouble remembering which symbol means what, look at the symbol. The less than symbol (<) can be distinquished by the fact that whatever appears on the left side of the symbol is less than what appears on the right side. Notice the left side of the symbol is "less than" the right side of the symbol. The same logic can be applied to the greater than symbol (>) in that the left side is greater than the right side.

The equal sign can be "added" to either the greater than or lesser than sign to arrive at two new relations; greater than or equal to (>=) and less than or equal to (<=). As examples, the following are true:

```
5 <= 5
5 >= 5
5 >= 4
5 <= 6
```

The less than and greater than symbols can be combined into one more significant relation, not equal. The not equal relation is defined by the combination of the two symbols (<>). When this combination is spotted within a program it is pronounced "not equal to."

The relationals can also be used to compare string variables and constants and to set string variables to other string variables or string constants. As an example:

```
10 A$="FILENAME"
```

TEST

1. What is the symbol for less than?

   <

2. What is the symbol for greater than or equal to?

   >= (also =>)

3. What is the symbol for equal to?

   =

4. What is the symbol for less than or equal to?

   <= (also =<)

5. What is the symbol for greater than?

   >

6. What is the symbol for not equal to?

   <>

7. What is the reserved word LET used for?

   LET is used to assign values to variables.

8. What is shorthand for the reserved word LET?

   Simply leave out the word LET.

9. For what reason is the REM statement used?

   REMark is used to add remarks within a program to document the program, making it easier for the programmer to remember how the program functions and what the variables names stand for.

10. What is a constant?

    A constant is a value which appears in the program but does not change within a program. A constant is often times used to "set" the value of a variable which can and does change its value throughout the program.

11. What is an operator?

    An operator is a symbol which represents an operation to be performed on one or more operands.

Strings variables may be compared just as we compared numeric variables. The comparisons are made by starting with the first character of each string, using the ASCII value to compare the value for each character. If the first characters are the same, **baZic** will continue advancing through the string comparing one pair of characters at a time until a difference is found where one string is greater or lesser than the other string.

Under this convention, the string "ABCD" would be less than (<) the string "BBCD" and the string "BBCD" would be greater than "ABCD." The string "AABC" would be greater than the string "AAAC."

To find the value of any particular character, look up the letter in the ASCII table in the back of the **baZic** manual. You should notice that numbers are "less than" letters of the alphabet and upper case letters are "less than" lower case letters.

Strings can be set to the "null" string by setting them to a double quote (A$=""). This null string contains "nothing" and is "less than" any other string.

ADDITIONAL READING

baZic manual Section 5.2.

12. What is an operand?

    An operand is the "quantity" upon which an operation is performed.

EXERCISES

Which of the following are true situations?

```
A.   6=5
B.   6<=5
C.   6<5
D.   6>5
E.   6>=5
```

D and E are the only valid relations. All others are false.

What is the value of A when the following program has completed execution?

```
10 A=20
20 B=A
30 B=30
40 A=B
50 B=50
```

A is equal to 30.

Of the following strings, which are greater than the string "This is a string":

```
A    "THIS IS A STRING"
B    "THIS IS NOT A STRING"
C    "THIS IS NOT THE LONGEST STRING IN THE WORLD"
D    "U"
```

Only option D is larger than the specified string. Each of the others begins with the same letter, but since the second letter of A, B, and C is upper case and the second letter of the string to be compared is a lower case letter, the specified string is "greater than" each of the three even though B and C are longer strings.

Decision Making

OBJECT

To learn how **baZic** makes decisions by the use of the IF THEN ELSE statement.

LESSON

The concept of the IF THEN ELSE statement is used by people everyday of their lives, but most people are not aware that they are implicitly making decisions by this convention. You might say, "If it rains I'll go to the store, but if it doesn't rain I'll go swimming." A translation of this sentence could be, "IF it rains THEN I'll go to the store, ELSE I'll go swimming."

This is the same way that **baZic** makes decisions. **baZic** would "look" at a situation, and based on the outcome, decide to do one of two things. A typical program decision-making statement would have the form:

```
10 IF A=0 THEN GOTO 20 ELSE GOTO 30
20 PRINT "THIS IS THE FIRST CHOICE"
30 PRINT "THIS IS THE SECOND CHOICE"
```

Line 10 is the decision-making line. **baZic** examines the variable A to determine if the value of A equals 0. If this situation is true, the THEN clause will be executed and processing will continue on line 20. If the statement is not true (A equals any value except 0), the ELSE clause will be executed and processing will continue at line 30.

The IF THEN ELSE statement is one of the most powerful statements in **baZic** and allows most any problem to be solved by breaking the problem into a series of decisions, each of which can be solved by the IF THEN ELSE statement.

Any of the relational operators from the previous session could be used to determine which clause of the IF THEN ELSE statement is to be executed. Some example uses of the IF THEN ELSE follow:

```
10 IF A>B THEN 30 ELSE 40\REM GOTO 30 IF A GREATER THAN B

10 IF A<>B THEN 30 ELSE 40\REM GOTO 30 IF A NOT EQUAL B

10 IF A<=B THEN 30 ELSE 40\REM GOTO 30 IF A LESS THAN OR
                                    EQUAL TO B
```

Notice a difference between these examples and the first examples of this session. In each of these examples, the GOTO statement is implied and not explicitly stated in the IF THEN ELSE statement. If the argument to the THEN or ELSE clause is a line number, there is no need to say GOTO although the syntax of the language certainly allows you to say GOTO if you want.

The argument to the THEN and ELSE clause can be any statement. The THEN clause can be followed by only one statement but the ELSE clause can be followed by as many statements as will "fit" on the line.

The ELSE clause is not required. An example of not using the ELSE clause would be:

```
10 INPUT A
20 IF A=0 THEN 50
30 PRINT "A DOES NOT EQUAL 0"
40 GOTO 10
50 PRINT "A IS EQUAL TO 0"
60 GOTO 10
```

If the IF argument is false, **baZic** will execute the next statement which logically follows, be it on the same or next line of the program. In our example, if the user inputs any value except 0, line 20 will evaluate false, and processing will continue on line 30 of the program.

To practice the IF THEN ELSE statement, either LOAD (if you saved the program) or re-type the following:

```
10 INPUT "Enter a number from 1 to 3 ",A
20 ON A GOTO 30, 50, 70
30 PRINT "This is option 1"
40 GOTO 10
50 PRINT "This is option 2"
60 GOTO 10
70 PRINT "This is option 3"
80 GOTO 10
```

This program has a major fault in that there is no check on the users response. The user could easily enter a number less than 1 or greater than 3, which would cause the program to generate a SYNTAX error and stop execution. The reason this condition generates a SYNTAX error will be given in the session on errors.

To correct the problem, add the following program lines to the program:

```
READY
12 IF A<1 THEN 10 <CR>
14 IF A>3 THEN 10 <CR>
```

Now LIST the program. It should appear as follows:

```
10 INPUT "Enter a number from 1 to 3 ",A
12 IF A<1 THEN 10
14 IF A>3 THEN 10
20 ON A GOTO 30, 50, 70
30 PRINT "This is option 1"
40 GOTO 10
50 PRINT "This is option 2"
60 GOTO 10
70 PRINT "This is option 3"
80 GOTO 10
```

The program will now do a check of the variable entered. If the value is less than 1 (line 12) the program will branch to line 10 which will display the prompt and request the information be entered again. If the value of A is greater than 3 (line 14), the program will again branch to line 10. This process will continue until the user enters a value in the correct range.

RUN this program and check the results.

As a last exercise, RENumber this program and then LIST the program to see the affect. The procedure should appear as follows:

```
READY
REN <CR>
READY
LIST <CR>

10 INPUT "Enter a number from 1 to 3 ",A
20 IF A<1 THEN 10
30 IF A>3 THEN 10
40 ON A GOTO 50, 70, 90
50 PRINT "This is option 1"
60 GOTO 10
70 PRINT "This is option 2"
80 GOTO 10
90 PRINT "This is option 3"
100 GOTO 10
```

Notice that the line numbers in the ON GOTO have been changed to reflect the new line numbers of the lines to which they were pointing.

ADDITIONAL READING

baZic manual Sections 3.3.6 and 8.3.2.

TEST

1. What is the decision making statement?

   IF THEN ELSE

2. Is the ELSE clause required?

   No

3. Is the GOTO statement required as an argument to the IF THEN ELSE statement, if the statement is to branch to a line number?

   No

4. What part of the IF THEN ELSE statement is evaluated as true or false?

   The argument to the IF.

5. If the IF clause evaluates true, what happens?

   The THEN clause statement is executed.

6. If the IF clause evaluates false, what happens?

   The ELSE clause or next logical statement is executed.

EXCERSISES

Practice writing programs using the IF THEN ELSE statement.

FOR NEXT Loops

OBJECT

To learn to recognize and use the FOR, NEXT, STEP, EXIT, and END statements.

LESSON

The FOR NEXT loop is one of the most used and most powerful branching and control structures in **baZic.** These statements allow the program to set up and control complex repetituous situations with little work on the part of the programmer.

Basically, the FOR NEXT loop is a counter. The loop is established with a control variable and a set of range variables. The programmer "tells" the loop to keep the processing within the loop as long as the conditions are met. Once the conditions are exceeded, control passes from the loop to the statement which immediately follows.

The basic form of the FOR NEXT loop is:

```
10 FOR N=1 TO 10
20 PRINT N
30 NEXT N
40 END
```

In this example program, N is the control variable, 1 is the beginning value, and 10 is the limit value. The FOR statement marks the beginning of the loop. The NEXT statement marks the end of the loop. Any statement, or statements, between these two will be executed repeatedly until the loop is completed (the control variable equals or exceeds the limit value).

This program will PRINT all of the integer numbers from 1 to 10. If we examine the flow of processing, we would start at line 10 where the control structure is established. The statement says to begin by setting N equal to the number 1. Line 20 says to PRINT the number. At line 30, the NEXT statement says to increment N by 1, return to line 10, and continue through the loop.

This process will continue until the variable N is equal to 10. Line 20 will print the last value (10), and processing will continue to line 30 where the NEXT statement will cause processing to continue with line 40 (a new statement which means to END the program). When the loop is completed, N will equal 11, 1 more than the value of the control variable.

The same program could be written using an IF THEN ELSE statement and a counter instead of the FOR NEXT statement. It would appear as follows:

```
10 N=1
20 PRINT N
30 N=N+1
40 IF N>10 THEN 50 ELSE 20
50 END
```

If we wanted to make the previous program PRINT all of the numbers from 1 to 10 but in increments of .1, we would modify the program to appear like this:

```
10 FOR N=1 TO 10 STEP .1
20 PRINT N
30 NEXT N
40 END
```

The only change to the program is the addition of STEP .1 in line 10. This causes N to be incremented by .1 instead of 1 for each journey through the loop. The STEP value adds even more flexibility to the FOR NEXT loop. By changing the STEP value, the loop can be made to retain control through a wide range of values.

If the starting value is greater than the limit value and the STEP value is a negative number, the loop can be made to count "backwards." The following example demonstrates the loop PRINTing from 10 to 1, STEPping by -1:

```
10 FOR N=10 TO 1 STEP -1
20 PRINT N
30 NEXT N
40 END
```

The starting value, limit value, and STEP value need not be constants and can be variables to lend even more flexibility to the loop. The same loop could be used, by changing the starting, limit, and step values, to count backwards or forwards by any STEP values. A FOR NEXT loop using all variables would appear as follows:

```
10 FOR N=A TO B STEP C
20 PRINT N
30 NEXT N
40 END
```

A FOR NEXT loop can be executed zero times (that is not executed at all). If the starting value already exceeds the limit value and the STEP value is a positive number, the loop will be skipped by **baZic** and processing will continue on the statement immediately following the FOR NEXT loop.

In some cases, we may not want to execute all of a FOR NEXT loop. We can EXIT from a FOR NEXT loop by using the EXIT statement. Internally, **baZic** has a counter which keeps track of where the program is within the loop. To leave a loop before the limit value has been reached, we must always close out the loop. This is accomplished by telling **baZic** to EXIT the loop. A program which demonstrates this statement is as follows:

```
10 FOR N=1 TO 10 STEP 1
20 PRINT N
25 IF N=5 THEN EXIT 40
30 NEXT N
40 END
```

This program would PRINT the numbers from 1 to 5 but would stop at 5 instead of 10. Line number 25 compares N to see if it is 5. Once N becomes 5, the EXIT statement causes program flow to branch to line 40 where the program ENDs.

FOR NEXT loops can be nested. That is, one FOR NEXT loop can be placed within another. The FOR NEXT loops can be nested to any level which your available memory will support. In other words, one loop can be within another, which is within another, which is within another, etc.

When loops are nested, the innermost loop must be contained completely within any exterior loops. If you have noticed, in the preceding examples, the NEXT statement is always followed by the control variable for the loop. This variable is not required by **baZic** but is helpful when loops are nested. **baZic** will always examine the variable to see if the proper NEXT is being executed for the proper loop. If the loop control variable does not match, the program will terminate with a CONTROL STACK error. A sample nested loop would be:

```
10 FOR N=1 TO 3 STEP 1
20 PRINT "N=",N
30  FOR X=1 TO 3
40  PRINT "X=",X
50  NEXT X
60 NEXT N
70 END
```

In this example, the innermost loop (the X loop) is nested within the N loop. When loops are nested, the innermost loop is usualy executed most often. The program above would produce the following output:

```
N=1
X=1
X=2
X=3
N=2
X=1
X=2
X=3
N=3
X=1
X=2
X=3
```

You can see that statment 40 was executed 9 times, 3 times for each time statement 20 was executed.

The example shows the actual statements indented one space so that it is easy to see what belongs to what loop. This is not necessary but is done by many programmers to make the program more "readable."

ADDITIONAL READING

baZic manual Section 3.3.7.

TEST

1. What is a loop?

   A loop is a control structure used in programming to control repetitious tasks.

2. How are loops formed in **baZic**?

   By using the FOR NEXT statement.

3. What is the step value?

   The STEP value is the amount the loop step value will be incremented or decremented each time through the loop.

4. What is the limit value of a FOR NEXT loop?

   The limit value is the value which is being compared with the present value of the loop counter to determine if the loop has completed the proper number of repetitions. When the limit value is reached, the loop is terminated.

5. What is the starting value of a FOR NEXT loop?

   The starting value is the value assigned to the loop control variable the first time through the loop.

6. How do you execute a FOR NEXT loop zero times?

   By setting the limit value greater than the starting value when the loop is first encountered by **baZic**.

7. What is the EXIT statement used for?

   The EXIT statement is used to terminate a FOR NEXT loop before the limit value is reached.

8. What is nesting?

   Nesting is when one FOR NEXT loop resides completely within another FOR NEXT loop.

9. What is the END statement used for?

   The END statement is used to end a program.

10. How many FOR NEXT statements can be nested?

    As many as your computer has internal memory to handle. Each FOR NEXT loop takes a predetermined amount of storage to store the starting, limit, and step values.

EXCERSISES

Write a program which uses FOR NEXT statements to enter and then print 10 numbers.

```
10 FOR N=1 TO 10
20 INPUT "ENTER A NUMBER ", A(N)
30 NEXT N
40 FOR X=1 TO 10
50 PRINT A(X)
60 NEXT X
70 END
```

The preceding program uses the A array variable to store each of the ten numbers which are to be entered. Once all of the numbers have been entered, the program prints each of the values. Notice that two different loop control variables are used, and the A array is referenced by both (by N during input and by X during printing). The same loop counter could have been used in both parts of the program because the second use of the N variable would have reset the variable to the proper number.

Subroutines

OBJECT

To learn the concept of subroutines and the **baZic** statements GOSUB, ON GOSUB, and RETURN.

LESSON

Subroutines are probably the saviour of computers. Without subroutines, computers, as we know them today, would not exist. Programs, of all levels, have many parts which are executed over and over again and it is generally a subroutine which is "doing the work." Subroutines conserve memory and structure programs making them indispensable in modern programming.

A subroutine is a specialized "piece" of a program (module), designed to accomplish one particular purpose, and available for call from any other module of a program. GOSUBs (subroutine calls) are different from GOTOs in that every subroutine called has at least one RETURN statement which causes processing to branch back to the statement immediately following the GOSUB statement.

The following program demonstrates the use of a subroutine:

```
10 FOR N=1 TO 3
20 GOSUB 50
30 NEXT N
40 END
50 PRINT "THE NUMBER IS ",N
60 RETURN
```

Notice the END statement at line 40. This statement keeps the program from "falling into" the subroutine which is lines 50 and 60. If this program is RUN, the results would be:

```
READY
RUN <CR>

THE NUMBER IS  1
THE NUMBER IS  2
THE NUMBER IS  3
READY
```

If we follow the execution of the program, we see the main control of the program is controlled by the FOR NEXT loop. The action in each step is simply a call of the subroutine (20 GOSUB 50). The processing is changed to the subroutine which PRINTs the message.

When the message is PRINTed the RETURN statement at the end of the subroutine causes control to pass to the next statement following the subroutine call (30 NEXT N). The process is continued three times until the limit value is reached in the FOR NEXT loop. When processing falls through the FOR NEXT loop, the END statement is executed which stops the program and prints the READY message.

The ON GOSUB statement is very similar to the ON GOTO statement, except when the subroutine is completed and the RETURN executed, processing continues at the next statement which follows the ON GOSUB statement. A sample program using the ON GOSUB statement is:

```
10 FOR N=1 TO 3
20 ON N GOSUB 50,70,90
30 NEXT N
40 END
50 PRINT "THIS IS THE FIRST SUBROUTINE"
60 RETURN
70 PRINT "THIS IS THE SECOND SUBROUTINE"
80 RETURN
90 PRINT "THIS IS THE THIRD SUBROUTINE"
100 RETURN
```

Running this program would produce the following results:

```
READY
RUN <CR>

THIS IS THE FIRST SUBROUTINE
THIS IS THE SECOND SUBROUTINE
THIS IS THE THIRD SUBROUTINE
READY
```

Again, this program uses a FOR NEXT loop as the main control. N is incremented to "feed" the ON GOSUB statement. Each time through the loop, a different subroutine is called. Each subroutine ends with a RETURN statement. The program ENDs when the limit is reached and the END statement is executed.

ADDITIONAL READING

baZic manual Sections 3.3.2, 3.3.3, and 3.3.5.

TEST

1. What is a subroutine?

   A subroutine is a single routine which accomplishes a particular task and is terminated by a RETURN statement.

2. What happens when a RETURN statement is encountered during the execution of a subroutine?

   Control immediately passes to the statement which follows the GOSUB which made the subroutine call.

EXCERSISES

Practice using GOSUBs in your programs.

If we take 20, subtract 2 and multiply by 5 we would get 90. If you run this program, the answer printed will be 10. This is because the precedence of operators controls **baZic** so that the ambiguities previously mentioned do not cause problems. In the example, the multiplication has the precedence so it is performed first, so the problem becomes 20 minus 10 which equals 10.

Now that you have some idea of the meaning of precedence, here is a list of the precedence of the operators starting with the operators which have the highest precedence (will be performed first):

OPERATOR	FUNCTION
-	Negation
^	Exponentiation
* /	Multiplication and Division
+ -	Addition and Subtraction

Whenever two or more operators are involved in a calculation, **baZic** first "looks" at the entire calculation and performs the calculation starting with the operations with the highest precedence.

If you want the precedence changed, use parenthesis. Here is the preceding example using parenthesis to alter the precedence so that the value becomes 90 instead of 10:

```
10 PRINT (20-2)*5
```

Now the subtraction operation is done first. This results (18) is then multiplied by 5 to get the value 90.

If you are not familiar with the term "exponentiation", please refer to a math book for a complete explanation. Exponentiation is used to raise a number to a power, that is, the number is multiplied by itself the number of times specified. The number 2 raised to the 2nd power would be 4 (2*2). The number 2 raised to the 3rd power would be 8 (2*2*2).

Exponentiation is specified in **baZic** by using the up arrow key (^). The number following the up arrow is the exponent. An example is:

```
10 PRINT 2^3
```

Of course the answer would be 8.

Strings may also be "added" together although that operation is called concatenation. An example of concatenating strings follows:

```
10 A$="THIS "+"IS"
```

Arithmetic Operators

OBJECT

To learn the arithmetic operators for negation (-), addition (+), subtraction, (-), multiplication (*), division (/), and exponentiation (^).

LESSON

The ability to do arithmetic is a necessary part of **baZic** and can be used in the direct (command) mode or the program mode. As an example, type the following in the direct mode:

```
READY
PRINT 10-5 <CR>
5
READY
```

In this example, we tell the computer to PRINT the answer to the arithmetic operation of subtracting 5 from 10. The answer is PRINTed immediately on the next line. Arithmetic can be used in any program just as easily. Here is the same problem as solved by a program:

```
10 PRINT 10-5
```

When the program is RUN, the answer (5) will be printed on the terminal. These two examples should show how easy arithmetic operations can be used in **baZic**. As a matter of fact, the direct mode is often called the calculator mode, since calculations can be used so easily.

In the preceding examples, we used the minus sign to indicate the subtraction operation. All of the symbols are staight-forward except the sign for multiplication. Since most computer terminals have no multiplication sign on the keyboard, **baZic** uses an asterisk (*) to signify the multiplication operation.

The arithmetic operators can be mixed in virtually any combination to allow complex calculations. However, the precedence of the operators must be considered. The precedence of operators means that some operators are considered before other operators in a mixed numeric calculation. Consider the following calculation:

```
10 PRINT 20-2*5
```

TEST

1. Name the operators and give the symbol used to specify that operation.

   Negation (-), Exponentiation (^), Multiplication (*), Division (/), Addition (+), and Subtraction (-).

2. What is the meaning of precedence and what is the order of precedence for the **baZic** arithmetic operations?

   Precedence is used to establish the order in which **baZic** handles numeric operators. The operators in order of precedence are: negation, exponentiation, multiplication and subtractions, and addition and subtraction.

3. What are parenthesis used for in numeric operations?

   To change the precedence of numeric operations.

4. What happens when you concatenate two strings together?

   They are added together.

5. What is the exponent in the following problem: 45^3.14?

   3.14

EXERCISES

Give the result of the following numeric operations.

```
5*10-2+3/4
```

     48.75

```
8+3+25/5+8*9
```

     208

```
60/5+34-7+10^2
```

     1039 or 1.039E3

Write programs using the arithmetic operators to add both numbers and strings.

If A$ is PRINTed after RUNning this program, A$ would be "THIS IS". Remember that the length of a string cannot be changed within a program. If the string is not long enough to hold the results, all characters past the dimension will be dropped. Here is an example where A$ is DIMensioned to the default value of 10 and the concatenation results in a string longer than 10:

```
10 A$="THIS IS "+"MY LAND"
```

If A$ is printed after running this example, A$ will contain only "THIS IS MY". "LAND" will be not be added to A$ since there is not enough room within the string.

ADDITIONAL READING

baZic manual Section 5.1.

Boolean Operators

OBJECT

To learn the use of the Boolean Operators, NOT, AND, and OR.

LESSON

George Boole was an English mathematician and logician who, in the mid 1800's developed the basic rules of logical algebra which describe propositions whose outcome can be described as either true or false. We have already studied the IF THEN statement which is totally dependent upon deciding if a situation is true or false.

The three logical operators to be learned in this session (NOT, AND, and OR), are, like the IF THEN concept, integral parts of our everyday language. We think nothing of saying, "I'll buy this item and that one.", meaning both items will be purchased. Another common statement might be, "I want to go to Dallas or Houston", meaning that you want to go to Dallas or to Houston but will not be going to both. A last situation is represented by a statement, "I'm not going", where the phrase "I'm not going" is the oposite of the statement "I'm going."

The logical operators (Boolean operators) work within a **baZic** program in the same manner we use them in our everyday speech. If we want two conditions to be met, we use the AND operator. If we want one (and only one) condition of two, we use the OR operator. If we want the opposite of a situation, we use the NOT operator.

As an example:

```
10 IF A=1 AND B=2 THEN 30
```

Line 10 will branch to line 30 only for the condition where A is equal to 1 and B is equal to 2. Any other values for either A or B will result in the program executing the statement which immediately follows the IF THEN statement.

Another example would be:

```
50 IF A=1 OR B=2 THEN 70
```

In this case, processing will branch to line 70 if A=1 or if B=2. If either condition is met, the IF THEN statement will be true and the THEN clause will execute.

For all situations involving logical operators, a value of 1 signifies a true situation and a value of Ø signifies a false situation. For instance, consider the following part of a program:

```
10 IF NOT A THEN 40
```

This statement is saying, in effect, "If A is not true (not equal to 1) then branch to line 40." The logical operator NOT can be used in many situations to reverse the "sense" of the operation.

ADDITIONAL READING

baZic manual Sections 5.3 and 5.4.

TEST

1. Who was George Boole?

   An English mathematician who developed the true/false logical operations.

2. What is Boolean logic?

   Logic based on the assumption that the results to the operation will always be either true or false.

3. When is a situation true when using the AND operator?

   When both parts of the operation are true.

4. When is a situation true when using the OR operation?

   When either part of the operation is true.

5. When is a NOT operation true?

   When the results of the operation evaluate to a logical 1.

EXERCISES

Determine the outcome (true or false) of the following operations. (Will the THEN clause be executed?)

```
A=Ø and B=Ø
IF A=Ø AND NOT B THEN
```

true

```
IF A=Ø OR B=1 THEN
```

true

Math Functions

OBJECT

To gain proficiency in the use of the **baZic** built-in functions which return values for the ABSolute value, SiGN, INTeger value, LOGarithmic value, EXPonetial value, SQare RooT, SINe, COSine, and ArcTaNgent of a number.

LESSON

If you need to do math, **baZic** has some functions for you. Did you ever "hate" all those formulas in school which were so very hard to calculate? They become easy with **baZic.** About all you really have to do is enter the formulas into **baZic** as syntactically correct program statement lines.

Remember that you can use variables or constants in your calculations. You can "feed" the equations with the proper INPUT statements or from stored data as you will learn later. A sample formula would be:

```
10 A=INT(3.1416*R^2)
```

Enter and RUN this simple program to calculate the square root of a number:

```
READY
SCR <CR>
READY
10 PRINT "This program calculates ", <CR>
20 PRINT "the square root of a number"<CR>
30 PRINT <CR>
40 INPUT "What is the number? ",N <CR>
50 PRINT <CR>
60 PRINT "The square root of the number is ",SQRT(N) <CR>
70 END <CR>

RUN <CR>

This program calculates the square root of a number

What is the number? 16 <CR>

The square root of the number is 4
READY
```

Lines 1Ø and 2Ø combine (because of the trailing comma of line 1Ø) to form the title of the program. Line 3Ø prints a Carriage Return to issue a line feed to separate the title from the prompt. The prompt is printed and the number input by line 4Ø. Another PRINT statement separates the prompt from line 6Ø which PRINTs the answer.

To make the program calculate the square root of many numbers change line 7Ø from an END statement to a GOTO 4Ø statement. If this modification is made, the program will form an endless loop. To rectify this situation change line 4Ø and 7Ø (use the line editor) and add line 45 to appear as follows:

```
4Ø INPUT "What is the number? (Ø to END)",N <CR>
45 IF N=Ø THEN END <CR>
```

The program LISTing would now appear as follows:

```
1Ø PRINT "This program calculates ",
2Ø PRINT "the square root of a number"
3Ø PRINT
4Ø INPUT "What is the number? (Ø to END)",N
45 IF N=Ø THEN END
5Ø PRINT
6Ø PRINT "The square root of the number is ",SQRT(N)
7Ø GOTO 4Ø
```

This is now a useful program (if you need to know the square roots of many numbers and you only have a computer and not a book with a table of square roots in it).

We have been working with the SQuare RooT function but we need to learn the other numeric functions of **baZic.** If you are unsure how to use any of the functions, consult the proper reference material for more information.

The ABSolute value function returns the absolute value of the number passed to it. The affect is to make the number positive. If the number passed is positive, the number is returned as positive, but if the number is negative, the function returns the positive (absolute) value of the number. This function is generally used with variables when the programmer does not know the actual value of the number. An example is:

```
PRINT ABS(-5) <CR>
5
READY
```

The SiGN function is used to return a value which indicates if the argument is positive, zero, or negative. If the argument is a negative number, the function returns a -1. A zero is returned for a number which is zero, and a +1 is returned if the argument is positive.

The INTeger function is used to remove the fractional part of a number. If the argument is already an integer, nothing is done to the number. If the argument is not an integer, everything right of the decimal point is stripped from the number. An example would be:

```
PRINT INT(3.1416) <CR>
3
READY
```

The LOGarithm function, naturally, returns the natural logarithm of the number passed to the function (argument).

The EXPonential function returns an approximation of the value of e raised to the power of the argument.

The SINe, COSine, and ArcTaNgent functions are all trig functions to use if you have a lot of triangles around with unknown sides and angles.

ADDITIONAL READING

baZic manual Sections 4.1, 4.1.1, 4.1.2, 4.1.3, 4.1.4, 4.1.5, 4.1.6, 4.1.7, 4.1.8, and 4.1.9.

TEST

1. What is a numeric function?

   A numeric function is an operation which is passed an argument and through a numeric calculation it returns a value which is based on the value of the argument passed.

2. What is the ABS function?

   Absolute value

3. What is the SQRT function?

   Square root

4. What is the SGN function?

   Sign of a number

5. What is the LOG function?

   Logarithm of a number

6. What is the INT function?

   Integer value of a number

7. What is the SIN function?

   Sine of a number

8. What is the EXP function?

   e to the power

9. What is the COS function?

   Cosine of a number

1Ø. What is the ATN function?

   Arctangent of a number

EXCERSISES

Write programs using numeric functions.

Additional Print Info

OBJECT

To learn the additional PRINTing capabilities of **baZic** which include PRINTing to a device number, TABbing, formatted PRINTing, cursor adressable PRINTing, and the CLear Screen statement.

LESSON

baZic has much more PRINTing capabilities than have been previously discussed. One such capability is the ability to PRINT to a device number, such as a printer. If **baZic** could only PRINT to the display terminal, it wouldn't be good for much. However, by including a print device number in a print statement, we can direct the printing of a program to as many as eight different devices, if these devices are defined on your machine. Assembly language routines must be in place to "talk" to each defined device.

Normally, device Ø, 1, and 2 are defined in most systems. Device Ø is usually the terminal or CRT with which you have been working. Device 1 is ususally the printer of the system. Device 2 is a combination of the other two in that it prints to both the CRT and the printer.

Recall our first program:

```
10 PRINT "THIS IS PROBABLY the simplest PROGRAM"
```

Now lets change this program to PRINT on the hard copy device (printer). Use the line editor to change this line by typing EDIT 1Ø. Use ^A to advance to the space between the reserved word PRINT and the beginning of the print argument. Now use ^Y to insert a print device specification (#1,). Enter another ^Y to terminate the insert and a ^G to copy to the end of the line. A Return will insert the line into the program. The line should now appear as follows:

```
10 PRINT#1, "THIS IS PROBABLY the simplest PROGRAM"
```

RUNning this program would cause the message to be displayed on the printer. We can make the program even more responsive to our needs by changing the "1" to a variable such as D. Edit the line so that the D replaces the 1 (EDIT 1Ø,Return,^D1,D,^G,Return). Also add line 5 so that the entire LISTing would appear as follows:

```
5 INPUT "Enter print device number (Ø, 1, or 2) ",D
10 PRINT#D, "THIS IS PROBABLY the simplest PROGRAM"
```

You now have control over which device will receive the print message.

Many times, you may want the printed message to start at a location other than the far left-hand margin. This is accomplished by TABbing to the proper location. The TAB function is inserted in the PRINT statement to cause the printout to be offset from the left margin the amount specified in the TAB function. The TAB amount is always measured from the left margin and not from the current position of the cursor or printhead of the printer. Here is the previous program using a TAB function to offset the message to start at column 20:

```
5 INPUT "Enter print device number (0, 1, or 2) ",D
10 PRINT#D,TAB(20),"THIS IS PROBABLY the simplest PROGRAM"
```

The next two topics are used for printing on a CRT type device only and will not work on most printers. They are the CLear Screen and PRINT@ (cursor addressing) statements. The CLS (CLear Screen) statement is used any time you want the screen of the CRT to be cleared. The statement can be used as a direct command or as a program statement. A sample use of this statement would be:

```
10 CLS\REM CLEAR THE CRT SCREEN
20 INPUT "ENTER A NUMBER ",A
30 PRINT "THE NUMBER IS ",A
```

This program would now clear the CRT and the prompt will be printed on the first line of the CRT. This leads us to the next print statement which is PRINT@. By including the at-sign (@) on the end of the PRINT statement and including row and column coordinates as arguments, this PRINT statement can be made to print anywhere on the CRT. As an example, if we wanted the message to be printed on line 10 and column 20 of the CRT, we would modify the program as follows:

```
10 CLS\REM CLEAR THE CRT SCREEN
20 INPUT "ENTER A NUMBER ",A
30 PRINT@(10,20), "THE NUMBER IS ",A
```

The PRINT@ statement is now included to direct the message to the proper location. If we wanted the prompt message to be printed at a specific location, we would modify the program to be:

```
10 CLS\REM CLEAR THE CRT SCREEN
20 PRINT@(2,1),"ENTER A NUMBER ",\INPUT A
30 PRINT@(10,20), "THE NUMBER IS ",A
```

Notice we use a PRINT@ statement to print the prompt and a backslash to separate the INPUT statement from the PRINT statement.

The next problem to tackle is called formatted printing. In the previous example, if we were asking for a dollar amount, we would want to print the number so that it looks like a dollar amount. We would want a dollar sign, commas in the correct place, and two digits to the left of the decimal point. Formatted printing is used to make this happen. Here is the same program re-written to ask for a dollar amount and to format the printout into a dollar figure:

```
10 CLS\REM CLEAR THE CRT SCREEN
20 PRINT@(2,1),"ENTER A NUMBER (DOLLAR AMOUNT) ",\INPUT A
30 PRINT "THE NUMBER IS ",%$C11F2,A
```

The PRINT@ has been removed from line 30 to enforce the print format specifications directly before the variable to be printed, but print formatting and PRINT@ can be used together.

The print format begins with a percent sign (%). This "tells" **baZic** that what follows is a print format specification. The dollar sign ($), says to insert a dollar sign at the left hand side of the number to be printed. The C says to insert commas at the appropriate places (every third digit to the left from the decimal place).

The 11F2 is the size specification. It says that the entire number, including all print specification (dollar signs, commas, decimal point, etc.), is to be 11 characters or less. The F is the specification of the type of format. The 2 says there will be two digits to the right of the decimal place. The entire field will be right justified (the right margins will line up) and if two or more numbers are printed in a column using this format specification, they will line up along the decimal point.

Add line 40 to the program to read:

```
40 GOTO 10
```

Now you should RUN this program several times, each time entering a different value, to see how the formatted printing works. If you enter a number which results in more than 11 digits total, you will get a FORMAT error and the program will terminate.

All of the new print enhancements learned in this lesson can be freely intermixed in a PRINT statement. You must, however, becareful about using CLS or PRINT@ on a printer. Remember that the shorthand for PRINT is ! and can be substituted anywhere you would use the reserved word PRINT.

ADDITIONAL READING

baZic manual Sections 3.2.1, 3.2.1.1, 3.2.1.2, 3.5.4, 4.5.4, and 7.1.5.

TEST

1. What is added to a PRINT statement to direct the printout to another device?

   #1,

2. Can the print number specification be a variable?

   Yes, this is often the best way.

3. What is the TAB function used for?

   To move the starting column postion of a printout from the first column to the specified column.

4. Write a statement to tabulate to position 40.

   10 PRINT TAB(40),"THIS IS IT"

5. How can you clear the screen under **baZic**?

   Use the CLS statement.

6. What statement is used to address the cursor on the CRT?

   PRINT@ or !@

7. What arguments are passed to the PRINT@ and !@ statements?

   Row and column coordinates of where you want to print.

8. What is the use of a percent sign (%) in a PRINT statement?

   It signifies the number printed is to be formatted.

9. What happens when a dollar sign is included in a print specification?

   The number is printed with a dollar sign at the left of the number.

10. What is a common print format specification for printing dollar amounts?

    %$C11F2

EXCERSISES

Write programs using the TAB function, device numbers, cursor addressing, screen clearing, and formatted printing.

The RAISE TO POWER operator (or ^ as it appears in baZic programs) requires additional explanation. baZic uses the LOG and EXP routines to evaluate ^ for non-integer powers, negative powers, and all powers greater than 30. Hence, if you delete LOG, the expression, 12^3 , will be evaluated properly. However, the expressions, 12^3.1 , 12^-3 , and 12^31 will result in SYNTAX ERRORs.

On completion of the selected modification SHORTB does a JMP to the 2D00H entry point of baZic. Selection of item 8, NONE OF THE ABOVE, returns you to baZic without modifying baZic. In all cases, any program which was in RAM will be lost.

If you want a permanent copy of the shortened baZic, you must exit to DOS by typing BYE and save baZic on the disk by typing SF BAZIC 2D00 (DOS version). The procedure would appear as follows:

```
READY
BYE
+SF BAZIC 2D00
+JP 2D00
READY
```

Table 1 lists the number of bytes of memory that can be recovered for your programs or data by selecting the various options. The savings are the same for both the hardware floating point and software floating point versions.

Table 1

BYTES RECOVERED BY DELETING MATHEMATICAL FUNCTIONS

	baZic08	baZic10	baZic12	baZic14
ATN	166	177	188	199
COS-SIN	395	432	454	476
LOG	657	704	750	784
EXP	957	1015	1079	1125
^	1025	1083	1147	1193
SQRT	1174	1234	1300	1348
RND	1266	1326	1392	1440

9.2 XREF

The XREF program is not available in the CP/M version of baZic.

XREF is an assembly language program which analyzes a baZic program and prepares a table of the variables used and the program lines in which these variables occur. Such a table can be extremely useful in documenting or de-bugging your programs. XREF requires RAM from BD00 to BFFF Hex to operate.

The following instructions for using XREF assume that you are in baZic and have your program loaded:

Strings

OBJECT

To become more familiar with **baZic** strings, sub-strings, and simulated string arrays.

LESSON

We have touched on strings in a previous session, but there is much more to learn. Strings are used to store all alphanumeric information. Alphanumeric means alphabetic characters as well as numbers, and special characters. In the next session, we will learn how strings may be converted to numbers and numbers converted to strings.

In **baZic**, strings may be as long as you want, if you have enough internal memory in your computer to support the string. Most BASIC interpreters allow strings to be only about 250 characters in length.

Once a string is defined, any part of the string can be accessed easily. Look at the following program:

```
10 DIM A$(60)
20 A$="NOW IS THE TIME FOR ALL GOOD MEN TO COME TO THE AID"
```

Every character position in A$ can be represented by a number, starting at the first postion in the string. To represent this numbering, A$ is printed again with the string position numbers directly under the string:

```
A$="NOW IS THE TIME FOR ALL GOOD MEN TO COME TO THE AID"
    123456789^123456789^123456789^123456789^123456789^1
             10        20        30        40        50
```

If, for some reason, we wanted to put the word "TIME" into B$, we would first decide where the word resides in A$. Looking carefuly at A$, we see that "TIME" is located from postion 12 to postion 15. To set B$ equal to the word "TIME", use the following equate statement:

```
20 B$=A$(12,15)
```

This statement says to set B$ equal to the part of A$ which is located between position 12 and 15. This process is called randomly accessing a string. As you can see from the example, we can access any part of a string by simply using the position numbers of the part to be accessed.

Now suppose we wanted B$ to be equal to the entire last part of the string ("TO THE AID"). We could accomplish this task in the same manner as the previous example, but since we want the end of the string, there is an easier way as shown in the next example:

```
20 B$=A$(42)
```

Since we want all of the string from postion 42 to the end, we need only enter the starting position and **baZic** will copy the string from the starting postion to the end. Any part of a string is called a sub string. If B$ were not DIMensioned large enough to hold the string, only characters upto the DIMension of B$ will be copied since B$ can never hold more characters than it is DIMensioned to hold.

In a similar manner, part of A$ could be set to the value of B$. An example of this would be:

```
20 A$(12,15)="TIME"      or
20 A$(12,14)=B$
```

So now we have learned how to pass information back and fourth between strings.

One of the features which **baZic** does not have that many other BASICs have is string arrays. String arrays are similar to numeric arrays in that many strings can be stored with one string name. Although **baZic** does not support this feature directly, it is very easy to simulate. Overall, the string handling capabilities of **baZic** are far greater than other BASICs.

A simple way to demonstrate a simulated string array is to use this feature in a program which would be similar to one used under a BASIC which has string arrays. One of the most simple cases is where the programmer wants to INPUT 10 names, each of which is 20 characters or less in length. The following program INPUTs the names into a simulated string array (A$) and then prints each element of the "array" showing how each substring is accessed by a record number:

```
10 DIM A$(20*10)
20 FOR N=1 TO 10
30 INPUT "ENTER A NAME ",A$(N*20-19,N*20)
40 NEXT N
50 INPUT "ENTER THE ELEMENT NUMBER TO PRINT ",N
60 PRINT A$(N*20-19,N*20)
70 GOTO 50
```

Line 10 DIMensions A$ to the array size (20 characters per name times 10 names). Lines 20 to 40 INPUT the names into the simulated array. We use a formula to calculate the position to put each name. Notice the first time through the loop, N equals 1 so the position evaluates to (1,20). The second time through the loop, N is equal to 2, so the position evaluates to (21,40).

The array element is accessed in the same manner. Line 50 asks for the array element number and then prints the appropriate name using the same computation.

ADDITIONAL READING

baZic manual Section 8.3.1.

TEST

1. What is alphanumeric information?

   Any character on the keyboard is alphanumeric.

2. How is alphanumeric information stored by **baZic**?

   In strings.

3. How are the character positions numbered in strings?

   Starting at 1 as the far left character and continuing to the number of characters in the string. Each space, charcter, special symbol, or number counts as one position.

4. What is a substring?

   A substring is any part of another string.

5. What is a string array?

   A string array is a string used to store more than one string in an orderly manner so that any element of the master string can be accessed through its element number.

6. Does **baZic** support string arrays directly?

   No.

7. Can **baZic** simulate a string array?

   Yes

EXCERSISES

Write programs using strings, substrings, and string arrays.

String Functions

OBJECT

To master the use of the string functions LEN, CHR$, ASC, VAL, and STR$.

LESSON

The string function are provided to help the programmer work with strings. If the programmer wanted to know the length of a string, all he would have to do is call the LENgth function. All string functions are called similar to other functions in that an argument must be passed to the function. In the case of the LENgth function, you must pass the variable name of the string you wish to find the length of.

The LEN function is used as follows:

```
10 A=LEN(A$)  or
PRINT LEN(A$)
```

After executing line 10 of the first example, the variable A will contain the length of A$. The second example will print the length of the string.

When a string is DIMensioned, the LENgth of the string becomes its dimension. When a string is in use, the length is the number of valid characters contained in the string.

One common use of the LENgth function is to center a string. Most times in a program, the programmer has no idea of the length of the string to be centered since the string may have been entered by a user. Assuming the print device is 80 characters wide, any string can be centered as demonstrated by the following program:

```
10 DIM A$(70)
20 INPUT "Enter a name to be centered ",A$
30 PRINT TAB(40-LEN(A$)/2),A$
```

To center the string the program must first find its length. The length is divided by 2 and subtracted from half of the screen width. The result is to TAB to the appropriate place to begin printing the string so the string will be centered. Notice the precedence of operations assures that the length of the string is divided by 2 before the results is subtracted from 40.

A computer actually has no way of storing letters, numbers (as we know them), or other symbols you see on the keyboard. All have to be stored as binary numbers. Therefore, each letter, number, or special symbol has a unique numeric code which represents the symbol internally in the computer. The name of the code which is used by most computers is the American Standard Code for Information Interchange, better know as the ASCII code.

The string function which returns the ASCII value of a character is the ASC function. To use this function, pass the string which you want to know the ASCII value as the argument to the function call. String constants may also be passed to return the ASCII value of the character. The function works only on the first letter of the string. A list of all the ASCII characters and their associated codes is contained in Appendix B of the **baZic** manual.

A sample use of the ASC function is:

```
PRINT ASC("A") <CR>
65
READY
```

baZic also provides a method of converting an ASCII code back to its character equivalence. This function is called the CHR$ (character string) function. An ASCII value is passed to the function and the function returns the appropriate character. A sample use of this function would be:

```
PRINT CHR$(65) <CR>
A
READY
```

The last two string functions are used to convert numbers within a string to numeric variables and to convert numeric variables to strings. The VALue function is used when you have a number which is stored as a string and you want the number stored as a numeric variable. An example of this function is:

```
PRINT VAL("1234") <CR>
 1234
READY
```

An error will be returned if you try to executed this function on a string which does not contain valid numeric characters. The reverse of this function is the STRing$ function. This function takes, as its argument, a number (or numeric variable) and converts this number into the string of your choice. As an example:

```
PRINT STR$(1234)
 1234
READY
```

Normally, you wouldn't need to PRINT the value returned from the function, but would set a string equal to the value returned. A sample program to demonstrate this would be:

```
10 A=1234
20 A$=STR$(A)
30 PRINT A$\REM A$=" 1234"
```

If you are using formatted variables, the string will be set to the current format which is in effect at the time the STR$ function is called.

ADDITIONAL READING

baZic manual Sections 4.2, 4.2.1, 4.2.2, 4.2.3, 4.2.4, and 4.2.5.

TEST

1. What is the LEN function used for?

   To find the length of a string.

2. What is the VAL function used for?

   To convert a string to a numeric variable.

3. What is the ASC function used for?

   To find the ASCII value of the first character of a string.

4. For what is the CHR$ function used?

   To change an ASCII code to its string (character) equivalent.

5. For what is the STR$ function used?

   To convert a numeric variable into a string variable.

EXERCISES

Practice writing programs using the LEN, ASC, STR$, CHR$, and VAL functions.

Input Functions

OBJECT

To become familiar with the additional methods of inputting data into **baZic** through the use of the INCHAR$, INP, INSTAT, and OUTSTAT functions.

LESSON

In previous sessions we have learned the INPUT statement as a means of getting information from the keyboard into the internal memory of the computer. A very useful input function is the INCHAR$ (INput a CHARacter string). This function differs in several important ways from the INPUT statement. When the INCHAR$ function is called, the device number must be passed as an argument to the function. If you want to gather data from device zero (the terminal), the function would be used as follows:

```
10 A$=INCHAR$(0)
```

The results of the INCHAR$ function is always a single character string. Numbers can be input with this function, but they must be first input as string variables and then converted to numeric variables. Another major difference is that the INCHAR$ function takes only one keystroke at a time. When the function is called in a program, the first keystroke entered by the user will be the value entered into the string. The user does not have to enter a carriage return to terminate the input.

The INCHAR$ function will also allow the entry of "control" characters (non-printing characters formed by pressing the control key and any other alphabetic character). Another major difference is that the line editor may not be used during a character input using the INCHAR$ function since only one character is being input and the function "takes" the input as soon as the key is pressed.

A simple program using the INCHAR$ function, is:

```
10 PRINT "Continue with this program? (Y or N)"
20 A$=INCHAR$(0)
30 IF A$="N" THEN END
40 IF A$="Y" THEN 10
50 PRINT "You didn't enter a Y or an N"
60 GOTO 10
```

RUN this program to examine the results. If you enter any character, notice that the function "grabs" the character immediately and continues execution. This situation differs from the INPUT statement which must be terminated by a carriage return before **baZic** continues processing.

The following program is a practical example of the use of the INCHAR$ function:

```
10 A2=0\B$=""
20 PRINT
30 PRINT "ENTER A NUMBER OR SOMETHING ",
40 A1$=INCHAR$(0)
50 A3=ASC(A1$)
60 IF A3=8 THEN 110
70 IF A3=13 THEN 110
80 B$=B$+A1$
90 A2=A2+1
100 !A1$,
110 IF A3<>8 THEN 170
120 IF A2=0 THEN 170
130 IF LEN(B$)=1 THEN B$=""
140 IF LEN(B$)>1 THEN B$=B$(1,LEN(B$)-1)
150 A2=A2-1
160 !CHR$(8)," ",CHR$(8),
170 IF A3<>13 THEN 40
180 PRINT
190 IF B$="" THEN 260
200 FOR N=1 TO LEN(B$)
210 IF ASC(B$(N,N))<48 THEN EXIT 260
220 IF ASC(B$(N,N))>57 THEN EXIT 260
230 NEXT N
240 PRINT "YOU ENTERED THE NUMBER ",B$
250 GOTO 10
260 PRINT "YOU ENTERED THE STRING ",CHR$(34),B$,CHR$(34)
270 GOTO 10
```

Obviously, this program is a "little" more complex than previous program examples, therefore, a line by line explanation will be given. The purpose of this program is to accept characters one at a time through the use of the INCHAR$ function and to convert these single characters into a separate string for storage. The last purpose of this program is to determine if the information entered is a string or a number.

10 The variable A2 must be set to zero. Since the INCHAR$ function only returns one character, we must use another string to "hold" the characters as we get them one by one. The variable A2 is used to "point" or keep track of the position within the string which stores these characters. B$ is "zeroed" to clear any previous characters from the string and to set the length to zero.

20 The PRINT statement is used to print a line space.

30 This line prints the prompt statement which tells the user to "Enter a number or something."

40 Here is the actual input of information. The one character string is input into A1$ from device 0.

50 This line converts the character input into its ASCII equivalent. This conversion makes it easy to determine which character was input.

60 & 70 There are two situations where a special character is input. The user may want to back up to correct a mistake or he may terminate the input with a carriage return. Line 60 makes a check for a backspace character while line 70 makes a check for a return. Lines 60 and 70 could have been combined into one statement using the Boolean operator OR:

```
60 IF A3=8 OR A3=13 THEN 110
```

80 If the character input was not a backspace or a carriage return, it must be a character the user wants to input. This line adds the last character input to B$ which acts as the acumulator for all the characters input.

90 We must keep track of the number of characters in B$. Since we have just added a character we must "bumb" the value in A2 by 1. A2 now "points" to the last character entered into B$.

100 This line displays the last character input by printing A1$. Notice that the PRINT statement is followed by a comma so that a carriage return is not printed so the cursor will remain on the same line.

110 For processing to get to this line, the character entered must be a backspace or a carriage return. If the character was not a backspace, processing will branch to line 170. Lines 120 through 170 are used to handle a backspace.

120 This line checks to make sure there is at least 1 character in B$. If A2 is zero, there are no characters so there is no way the user can back up. Processing branches to line 170 if A2 is zero.

130 If there is 1 character in B$ and the user wants to back up (he has entered a backspace), we would want to clear B$. The easiest way to clear the string is to set is equal to "nothing". This is accomplished by line 130.

140 Line 140 "substracts" 1 character from B$. If B$ is greater than 1, there is enough characters in the string to erase 1. This task is accomplished by setting the string equal to itself, minus the last character. By setting the LENgth of the string to one less than it previously was, we have effectively erased the last character.

150 Since we have taken 1 character from B$ we must now subtract 1 from the pointer A2.

160 This line backs up 1 space, prints a space charcater (to erase the character being deleted, and backs up again to position the cursor to the correct place. The backspace character is "printed" by using the CHaRacter string function, since the backspace "character" is a non-printing character.

170 Line 170 acts as a dual purpose line. It receives the processing from line 110 if the character input was not a backspace and it also terminates the backspace handling section by returning processing to line 40 if A3 is not a return. If A3 is a return, the user has finished his input and the return is PRINTed by line 180.

190 This line is a special check for an empty string. If the user has entered nothing except a carriage return, B$ will be equal to "" and line 190 will pass processing to line 260. Lines 200 through 270 are used to determine if the user entered a string or a number. The routine provided is used to display the string or number and then return to let the user enter another. Of course, more useful routines could be substituted to use the numbers or strings input in your program.

200 Lines 200 through 230 are used to "look" at each character in B$ to determine if the character input was a string of a number. A FOR NEXT loop is established from 1 to the number of characters in B$ (LENgth of B$).

210 & 220 Line 210 "looks" at character N to see if the ASCII value of that character is less than 48 which is a zero (0). If it is, the program branches to line 260. The EXIT statement is used to terminate the FOR NEXT loop since we have determined that B$ is a string and not a number. Line 220 is similar, except it looks to see if the character is greater than 57 which is the number 9. Check your ASCII chart in the back of your **baZic** manual for the ASCII codes for the numbers 0 to 9.

230 This process is repeated until all of the characters in B$ have been checked. If any character is not a number, B$ is considered to be a string. If all the characters are numbers, B$ is printed as a number. The VAL function could be used to convert B$ to a numeric variable.

240 & 250 These two lines handle the situation where a number has been input. A message is printed and B$ is printed to indicate that B$ contained a number. Line 250 returns processing to line 10.

260 & 270 These two lines handle the situation where a string has been input. A message is printed indicating this fact and the string itself is then printed.

The INP function is used only in special cases to input a single character from a specified port. You must have some knowledge of Z8Ø machine language to understand this function. The function is passed the port number from which you want the character. The Z8Ø supports 256 ports (numbered Ø to 255). When you call the INP function, it returns the character that resides at the port at the instant the function is called.

Care should be used when using the INP function. To see the value at all 256 ports, run the following program:

```
1Ø FOR N=Ø TO 255
2Ø A=INP(N)
3Ø PRINT "PORT NUMBER ",N," HAS THE VALUE ",A
4Ø NEXT N
```

The INSTAT and OUTSTAT functions are not really used to input information, but rather to determine if the port is ready. Again, a knowledge of machine language is required to understand how information is actually input. In most systems, there is a control port for each device port. When a character is pressed at the keyboard, a port goes true to indicate that a character is ready at the device port. The INSTAT and OUTSTAT functions monitor the control ports to let you know if a character is ready.

A sample use of these functions would be to monitor the keyboard during a printout. For each record printed, you could call the INSTAT function and if a character was ready, input it and terminate printing if the right character was input. The OUTSTAT function could be used to print a document or file at the same time you are entering information.

ADDITIONAL READING

baZic manual Sections 4.3, 4.3.1, 4.3.2, 4.3.3, and 4.3.4.

TEST

1. What argument is passed to the INCHAR$ function?

   The device number.

2. How many characters are passed by the INCHAR$ function for each call of the function?

   One character is passed for each call of the function.

3. Is a return necessary to terminate a response to the INCHAR$ function?

   No.

4. Can numbers be input using the INCHAR$ function?

   Yes, but they must be specifically handled by additional routines.

5. Can control characters be input using the INCHAR$ function?

   Yes. All characters on the keyboard can be input using this function.

6. Can the line editor be used during an INCHAR$ function call?

   No.

EXCERSISES

Change the INCHAR$ program so that the delete key can be used to back up in addition to the backspace key.

Add line 65 to read:

```
65 IF A3=127 THEN 110
```

and change line 110 to read:

```
110  IF A3<>8 AND A3<>127 THEN 170
```

Change the INCHAR$ program so that all numeric responses are changed to a numeric variable before printing the variable.

Add line 235 to read:

```
235 B=VAL(B$)
```

and change line 240 to read:

```
240 PRINT "YOU ENTERED THE NUMBER ",B
```

Miscellaneous Functions

OBJECT

To learn the use of the **baZic** functions RND, EXAM, FREE, CALL, and ADDR.

LESSON

All of the functions listed in this session are unrelated functions so they will be described in no particular order.

The random function (RND) is used to generate a random number. Care must be used when you refer to a random number. It could be argued that there are no random numbers, but if there are, this function DOES NOT generate one. What this function does do is generate a pseudo-random number. For all pratical purposes, the numbers are random.

This function is called by passing an argument which is called the seed. When the same seed is used as an argument, the same sequence of random numbers is generated. For testing purposes, it is often necessary to generate the same random sequence, so just use the same seed.

The seed may be a negative one, zero, or a positive number between zero and one. If the argument is a negative one, **baZic** will use the refresh register of the Z8Ø microprocessor to obtain a "random" number to use as the seed. If the argument is zero (Ø), the previous number generated will be used as the seed. Any number between Ø and 1 can be used as a seed to generate a "standard" psuedo-random sequence.

The following functions are related to the machine language of the Z8Ø processor. If you are not familiar with the concepts of byte and address, you should inquire before continuing with this session.

The EXAM statement is used to "look" at a physical address within internal memory. The argument must be in the range that can be addressed by the Z8Ø. This range is from Ø to 65535 decimal. This function could be used to look at the **baZic** interpreter itself or any program or data which resides in memory.

In a **uDoZbaZic** system, you could determine the starting address of **uDoZ** by the following program:

```
1Ø PRINT EXAM(1)+256*EXAM(2)
```

The FREE function is used to return the amount of memory which is free or unused in your system. The function is called by passing a dummy argument, that is, any argument is allowed. A sample function call would be:

```
PRINT FREE(Ø)
```

The value returned is the amount of memory that has not been used to store the operating system, the **baZic** interpreter, the **baZic** program, and any associated data. The value is always a decimal number.

The CALL function is a very special function and greatly extends the power of **baZic.** The call function gives the user the ability to make a machine language call. In this situation, the values passed as arguments are placed in internal Z8Ø registers and control is passed to the specified address so that the machine language routine can "do it's thing." Great care should be used when CALLing machine language since there is no error trapping and catastrophic results can easily occur.

The CALL function is passed the address of the routine and a single value which is to be placed in the DE register pair. The machine language routine is terminated by the use of the RETurn instruction. The value in the HL register pair is returned from the function call.

The ADDR function is used to return the address of a **baZic** variable. This function can be used in conjuction with the EXAM function and FILL statement to manipulate variables in internal memory. One use of this function under **uDoZbaZic** is to define a buffer which can be used to pass data for disk backups or copies.

The argument for this function is name of the variable for which you want to know the address. As an example, to return the address of the variable B$, call the function as shown below:

```
A=ADDR(B$)
```

A will now contain the decimal address of the variable B$.

ADDITIONAL READING

baZic manual Sections 4.5.1, 4.5.2, 4.5.3, 4.5.5, and 4.5.6.

TEST

1. What are the allowable arguments to the RND function?

   -1, Ø, and any number between Ø and 1

2. What argument would you use to have **baZic** select the seed?

   -1

3. Are random numbers really random?

   No, just a psuedo-random sequence.

4. What does the EXAM function do?

   It returns the value in the memory cell of the address passed.

5. What number system is passed and returned from all **baZic** functions?

   Decimal.

6. What is returned from the FREE function?

   The amount of unused memory in the system in decimal.

7. For what is the CALL function used?

   The CALL function is used to pass arguments and control to a machine language routine which passes a value back upon completion of the routine.

8. What is returned from the ADDR function?

   The address of the variable passed as the argument.

EXCERSISES

Write a program to generate random numbers in the range of 1 to 1Ø.

Use the EXAM function to write a program to look at the first 1Ø instructions of **baZic**.