KAYPRO

S-BASIC®

S-BASIC©

A Language Facility

For CP/M© and its Derivatives

By KAYPRO Division, Non-Linear Systems
533 Stevens Avenue
Solana Beach, CA 92075

S-BASIC, first revision, January 1983
S-BASIC, second printing, May 1983

End User's copy of License. For information only. Do not fill out.

COMPUTER PROGRAM END USER LICENSE AGREEMENT

AGREEMENT between Topaz Programming of San Diego, California . . . as licensor hereinafter referred to as TOPAZ and of

_____

_____

_____

hereinafter referred to as END USER LICENSEE.

RECITALS: TOPAZ develops proprietary computer programs and licenses the distribution and use of said programs. S-BASIC is a trademark of TOPAZ and is the designation used by TOPAZ for its language compiler package of computer routines. An S-BASIC UNIT of TOPAZ material shall consist of a diskette containing a CP/M compatable copy of S-BASIC, a user's reference manual, and four copies of the END USER LICENSEE agreement. CP/M is the trademark of Digitial Research Corporation. TOPAZ product materials include, but are not limited to, manuals, license agreements, proprietary computer programs, and media upon which TOPAZ's proprietary computer programs are recorded.

GRANT OF LICENSE: Subject to the terms and conditions of this agreement TOPAZ grants to END USER LICENSEE a non-exclusive, non-transferable license for the use of S-BASIC, and its companion reference manual.

TERMS AND CONDITIONS

ARTICLE I. EXCLUSIVE SOURCE. END USER LICENSEE shall obtain all TOPAZ product materials through TOPAZ or an authorized LICENSEE and no other source. END USER LICENSEE shall make no copies of any kind of any portion of the materials furnished unless specifically authorized to do so in writing signed by an officer of TOPAZ or otherwise specifically permitted by a provision of this license agreement.

ARTICLE II. LIQUIDATED DAMAGES. END USER LICENSEE recognizes that TOPAZ has expended considerable time and expense to develop TOPAZ's products and TOPAZ would be damaged by unauthorized copying or reproduction of TOPAZ's product materials. In the event END USER LICENSEE breaches this agreement by unauthorized copying or reproducing of TOPAZ's product material END USER LICENSEE agrees to pay TOPAZ as liquidated damages, the sum of $5,000 (Five Thousand Dollars) for each occurance of the unauthroized act of copying or reproducing of TOPAZ product materials.

ARTICLE III. ARCHIVAL COPIES. END USER LICENSEE may make archival copies of those portions of TOPAZ's product(s) that are provided on machine readable media, provided such copies are for the END USER LICENSEE'S personal use and that no more than one such copy is in use at any time except as provided in ARTICLE IV of this agreement. END USER LICENSEE agrees to label each arhival copy with a readable reproduction of TOPAZ's copyright notice, product name, and END USER serial number as furnished with the vended media. Failure to so label each archival copy shall be considered a breach of ARTICLE II under the terms of which liquidated damages are due.

ARTICLE IV. MULTIPLE COPY USE. TOPAZ use licenses are applicable to a single computer installation. In the event that END USER LICENSEE should desire to use TOPAZ product or any portion thereof in more than one computer, the required fee for each such use must be paid. In the event that END USER LICENSEE wishes to incorporate all or any part of TOPAZ's products in an END USER LICENSEE system or product to be sold, an APPLICATIONS LICENSE must first be obtained from TOPAZ. The required license fee must be paid for each such instance of use, and the required registration procedure must be followed.

ARTICLE V. REWARD FOR INFORMATION LEADING TO SUCCESSFUL PROSECUTION. In the event any party, including END USER LICENSEE, should provide information to TOPAZ that leads to successful prosecution and recovery of liquidated damages due to unauthorized reproduction of TOPAZ product materials, TOPAZ will pay the sum of $2,000 to the provider of such information. The decision to pursue legal action shall be at the sole option of TOPAZ. Furthermore, any sum paid will only be paid once, and in the event of multiple claimants, will be distributed at the sole discretion of TOPAZ.

ARTICLE VI. LIMITED WARRANTY POLICY. TOPAZ warrants that all materials furnished by TOPAZ to END USER LICENSEE is an accurate manufacture of TOPAZ product and will replace any such TOPAZ furnished material found to be defective, provided such defect is found prior to or within ten days of purchase by END USER LICENSEE. However, TOPAZ makes NO express or implied warranty of any kind with regard to performance or fitness for any particular purpose for any TOPAZ product by any user. Furthermore, TOPAZ IS NOT RESPONSIBLE for any loss or inaccuracy of data of any kind or for any consequential damages resulting therefrom whether through TOPAZ's negligence or not. TOPAZ will not honor any warranty where TOPAZ product has been subjected to physical abuse or used in defective or non-compatible equipment. TOPAZ will not honor any express or implied warranty of any kind with regard to performance or fitness for any purpose for any TOPAZ product made by any DEALER.

ARTICLE VII. UPDATE POLICY. TOPAZ may, from time to time, revise the performance of its products and the content of its documentation. It is TOPAZ's intention to provide such revisions to END USER LICENSEES at nominal cost through TOPAZ DEALER LICENSEES and DISTRIBUTOR LICENSEES. However, nothing in this ARTICLE nor any other portion of the Agreement shall be construed to mean that TOPAZ, is obliged to alter it's products in any way, or to furnish such alteration, if made, to any party.

ARTICLE VIII. GOVERNING LAW. This agreement shall be interpreted in accordance with the laws of the State of California. In the event any part or this Agreement is invalidated by court or legislative action, the remainder of this Agreement shall remain in binding effect.

ARTICLE IX. LEGAL FEES. In the event of legal action to enforce the provisions of this agreement or secure damages as may result from a breach of this agreement, the prevailing party shall be entitled to reasonable attorney's fees, and his costs in addition to any other amounts awarded by the court.

ARTICLE X. WAIVER. The failure of TOPAZ to insist, in any one or more instances, upon a strict performance of any of the provisions of this agreement, or to exercise any option herein contained, shall not be considered as a waiver or relinquishment for the future concerning said provision or option, but the same shall continue and remain in force or effect. The failure of TOPAZ to pursue its remedies or assert its rights upon notice of the breach of any of the terms of this agreement by the END USER LICENSEE shall not constitute a waiver of such rights or remedies with respect to the same or any other breach.

ARTICLE XI. TERMINATION FOR MATERIAL BREACH. In the event of a material breach of the terms of this agreement by END USER LICENSEE TOPAZ may, at its option, withdraw the license granted END USER LICENSEE and require the immediate return of all TOPAZ material and all copies of TOPAZ material. In no event will TOPAZ be liable for consequential damages due to termination even if TOPAZ has been advised of the possibility of such damages.

ARTICLE XII. IMPROVEMENTS AND PATCHES. Any improvements or patches to TOPAZ programs made by END USER LICENSEE or at his direction shall become the exclusive property of TOPAZ, except as may be provided in this licensing agreement. END USER LICENSEE shall provide copies of any such improvements or patches within ten days of a written request from TOPAZ for such material. Copies as may be requested by TOPAZ must be in a form or on media immediately usable by TOPAZ and not require transcription. END USER LICENSEE providing improvements or patches to any S-BASIC program material shall have the right to use himself such improvements or patches at no extra charge by TOPAZ.

ARTICLE XIII. ENTIRE AGREEMENT. This Agreement constitutes the entire agreement between the parties and supersedes any prior agreements.

PREFACE

This S-BASIC User's Guide was written in two parts: a Beginner's S-BASIC section, and an S-BASIC Reference section, each section having a separate index at the back of the User's Guide. It is recommended that, if you have never programmed a computer before or if you are not familiar with BASIC computer languages, you read through the Beginner's Manual and try the example programs. The S-BASIC Reference Manual provides a more complete and technical description of S-BASIC for those already familiar with BASIC programming and also for those who have completed working with the Beginner's Manual and who want to learn more details of S-BASIC programming.

TABLE OF CONTENTS

Chapter 1

YOUR S-BASIC DISKETTE

A diskette is used to hold information. This stored information is called a file. A disk can hold a number of files, depending on how much information is in each file.

To use S-BASIC in your computer, you will use two diskettes:

* A word processor/text editor disk, which will be the disk with which you write your programs.

* A S-BASIC disk, which has only S-BASIC and blank programming space on it.

You will make the S-BASIC diskette by copying part of your CP/M S-BASIC disk onto a blank, formatted disk.

1. Turn on your computer.

2. Put the disk marked CP/M S-BASIC in drive A. You will see A> on your screen. This is the prompt and is waiting for you to enter something on the keyboard.

3. Put a blank, formatted disk in drive B. (Note: If you don't know how to FORMAT a blank disk, refer to your computer's User's Guide).

4. Enter the following, pressing RETURN after each line. You will see an asterisk (*) appear when your computer is ready for the next line.

```
A>PIP
*B:=A:SBASIC.COM[OV]
*B:=A:OVERLAYB.COM[OV]
*B:=A:BASICLIB.REL[OV]
*B:=A:USERLIB.REL[OV]
*
```

5. Press RETURN. Your S-BASIC diskette is now completed.

6. Take the diskette out of drive B, and label it: S-BASIC

You now have the diskette you need to begin learning how to program in S-BASIC computer language.

The above instructions are in a generalized form. If you have any trouble, please refer to your computer's User's Guide for more information.

It would also be advisable for you to study your word processor/ text editor User's Guide and practice using it until you are familiar with it. Then, go onto the next chapter in this manual to begin learning S-BASIC.

Chapter 2

FUNDAMENTALS OF S-BASIC PROGRAMMING

In the last chapter, we explained that a diskette holds information in the form of files. Now, we will be more specific, and explain the filename.

The first part of a filename lets the computer know which drive holds the diskette with the file you want to use. If the file you want is on the diskette in drive B, then B: would be the first part of its filename. Similarly, if the file you want is on the diskette in drive A, then A: would be the first part of its filename. To avoid confusion, it is a good idea to always include the first part of the filename when entering any file into your computer. For example, if you enter B:TRYOUT.BAS your computer knows to look for that file in drive B.

The second part of the filename is the actual name part. It can be up to eight letters long. In the filename B:TRYOUT.BAS, the name is TRYOUT.

The last part of the filename is the extent. The extent may be up to three letters long. When you are writing programs in S-BASIC, your original files will have filenames with the extent, BAS. It is important to put a period (.) between the name and the extent in a filename, so your computer can tell them apart. Other extents you will become familiar with while using S-BASIC are BBX and COM. The extent, COM, is a special case in itself. When a filename has an extent of COM, you don't need to include it in the filename when you tell the computer to get (call up) that file. For example, if you had a file with the filename, B:TRYOUT.COM, then all you would have to enter to call up this file would be B:TRYOUT.

To illustrate this, let's write a short program in S-BASIC.

1. Turn on your computer.

2. Put your word processor/text editor disk in drive A.

3. Put the S-BASIC disk you made in the last chapter in drive B. You should see a prompt, A>, on your screen.

4. Create a new file with your word processor/text edit. Name the file: B:TRYOUT.BAS Remember that you need to include BAS as your filename's extent whenever you write an S-BASIC program.

5. Type the following into the file, B:TRYOUT.BAS:

PRINT "This is my first S-BASIC program."

Press the RETURN key at the end of the line.

This demonstration program uses the PRINT statement, which tells the computer to print on the screen whatever is in quotation marks following the PRINT statement.

6. Now, save this program on your diskette in drive B.

7. Quit the word processor/text editor, and return to the CP/M operating system.

8. To use this program, type B: and press the RETURN key. This will change the prompt, A> to B>.

9. Then, enter: SBASIC TRYOUT.BBX
Press RETURN.

The extent BBX contains useful information in each of its letters:

* The first letter indicates which drive holds the diskette storing the program you wrote.

* The second letter indicates which drive holds the diskette which will receive the finished program.

* The third letter tells the computer where to list the program (X = screen, Y = printer, and Z = no listing). The third letter can also tell the computer to create a file of the program listing on the diskette in either drive (B = drive B, A = drive A) under a filename with the extent PRN. For example, if you had entered SBASIC TRYOUT.BBB above, then the computer would have listed the program, not on the screen, but in the file TRYOUT.PRN on the diskette in drive B.

Entering SBASIC TRYOUT.BBX started the computer translating your program into a language that the computer understands. The computer, complex machine that it is, only understands electronic signals. So there is a stepladder of increasingly-compacted languages connecting the computer to you, the programmer. The simple statement PRINT, which you used in your first program, eventually expands into a long string of electronic signals that

your computer can understand. To change your program into this long string of signals (called machine language), S-BASIC uses a "compiler."

You will see your program written on the screen and the message will appear:

******End of program******

Then, the computer will begin compiling your program into machine language. It will take a while for the computer to compile your programs, especially as they become longer and more complex. When it is done, you'll see on the screen:

Compilation complete

Now, you're ready to run your first program.

10. Type in: B:TRYOUT. Notice that you didn't have to type in the extent part of the filename (the full filename is B:TRYOUT.COM), because after a BAS files is compiled into machine language, it is put in a COM file. COM stands for COMMAND and means that the file is written in machine language and can run directly on the computer.

Now press the RETURN key. On the screen you'll see:

This is my first S-BASIC program.

Congratulations, you have written your first S-BASIC program!

In summary, in this chapter you have learned that you need to store your programs on the S-BASIC diskette in drive B, using the extent BAS so that your filename will resemble the form, B:NAME.BAS where NAME is the name you have chosen for your program's filename. To start the compilation of your file into machine language, you change the prompt A> to B>, and then enter: SBASIC NAME.BBX. To run your program, you enter: B:NAME. You have also been introduced to the PRINT statement.

So, now that you know the fundamentals of the programming process, you are ready to learn more in the following chapters of the statements that make S-BASIC such a powerful programming tool for your computer.

Chapter 3

LEARNING SOME S-BASIC STATEMENTS

In this chapter, you will be introduced to some S-BASIC statements by writing three sample programs on your computer.

Program 1

1. First create a new file with your word processor/text editor, called: B:SAMPLE1.BAS.

2. Enter the following program. Remember to press the RETURN key after each line, including the last line:

```
VAR X,Y,Z=REAL
LET X=2
LET Y=3
LET Z=X+Y
PRINT Z
```

The first statement VAR stands for variable and lets the computer know that you are using certain characters that will later stand for numbers. In this case X, Y, and Z will get numerical values in the next few lines. Think of X, Y, and Z as Post Office Boxes into which you put pieces of paper with numbers written on them.

The second line introduces the LET statement. Here you are telling the computer to make X equal to 2 (to store a piece of paper with the number 2 on it in P.O. Box X).

In the third line, you use the LET statement to make Y equal to 3 (to store the number three in P.O. Box Y).

The fourth line uses the LET statement to make Z equal to X and Y added together. And, since you have already told the computer that X=2 and Y=3, X plus Y is just another way of saying 2 plus 3, or 5.

The fifth and final line tells the computer to PRINT Z on the screen. So, when you run this program, what will appear on your screen is the number value of Z--in this case, 5.

Try running this program now.

3. Remember that you must first save your program.

4. Then, when the file is written on the diskette, exit to the CP/M operating system.

5. Type B: and press the RETURN key to change the prompt A> to B>.

6. To tell the computer to compile the program into a language that the computer can understand, enter: SBASIC SAMPLE1.BBX. You have to do this after each new program that you write before you can run it.

We will assume from now on that you know how to perform these steps after writing a new program. When you have done all of this, run your program by entering: B:SAMPLE1

The number 5 will appear on the screen. And this is exactly what was expected. Z was printed, and Z=X+Y, with X=2 and Y=3.

The next program will be almost the same as the last, except that you'll learn how to use the INPUT statement.

Program 2

Create a new file called B:SAMPLE2.BAS and enter the following program:

```
VAR X,Y,Z=REAL
LET X=5
INPUT Y
LET Z=X+Y
PRINT Z
```

In this program, you once again used the VAR statement to tell the computer you are using the characters X, Y, and Z and that they will later stand for certain numbers.

Line two says that X will be set equal to 5.

Line three introduces the INPUT statement, which you will see demonstrated below.

Line four makes Z equal to X plus Y. In fact, the only major difference in this program from the last one is that you do not tell the computer in the program what the character Y equals. The third line will take care of this, as you will see.

Get ready to run this program. You will see a question mark (?) appear on the screen. This is how the computer asks what you want to input. In this case, you know from the program that INPUT is waiting for you to enter a numerical value for Y. If, for example, you enter 7 and press the RETURN key, the computer will print 12. That is, your program tells the computer to PRINT Z, and Z=X+Y with X=5 and Y equal to whatever number you input, which was 7 this time, so Y=7. Z=X+Y or Z equals 5 plus 7. The answer, in this case, is 12. Experiment with this program by running it and inputting different numbers for Y.

Program 3

This program will use two INPUT statements and multiply numbers together instead of adding them, as you did with the first two programs.

Create a new file called B:SAMPLE3.BAS and enter the following:

```
VAR X,Y,Z=REAL
INPUT X
INPUT Y
LET Z=X*Y
PRINT Z
```

Lines two and three are INPUT statements, one for the value of X and one for the value of Y.

Line four is making Z equal to X times Y. The asterisk sign (*) is what you use between numbers or characters to multiply them together. Note that line four had to come after the lines that tell the computer what X and Y will equal, because the value of Z depends on the computer knowing the values of X and Y before it can compute what Z will be.

Line five will simply print Z , which is the result of multiplying X times Y.

Run this program, and you will see a question mark (?). Enter the number 3, and press RETURN. This is the X number value. You'll then see another question mark (?). Enter 4, and press RETURN. This is the Y number value. The computer should print 12 on the screen; that is, the product of 3 times 4. Play around with running this program for a while until you are familiar with it.

So, in this chapter, you have used some fundamental statements: VAR, LET, PRINT, and INPUT. In the next chapter, you will become more familiar with the PRINT statement.

Chapter 4

MORE ABOUT THE PRINT STATEMENNT

You used the PRINT statement in programs in the last chapter to print a number on your screen. You can print a list of numbers also, and how they will look on your screen depends on what punctuation mark you use to separate what you want to print. The following program will help to illustrate this. Create a file called B:PRINT1.BAS. Then, enter the program:

```
VAR A,B,C,D,E=REAL
LET A=1
LET B=2
LET C=3
LET D=4
LET E=5
PRINT A;B;C;D;E
PRINT A,B,C,D,E
```

The first six lines simply tell the computer what the values of A, B, C, D, and E will be. Lines seven and eight look almost the same, except that semicolons (;) are used in line seven to separate what is to be printed, and commas (,) are used in line eight to separate what is to be printed. The effect this has on what you will see on the screen of your computer will be easy to see if you run this program. Do so now.

You can see by having run the program that a semicolon (;) separates the numbers with a blank space, that is, if the number is positive. If the number is negative, then there would be a minus sign (-) before the number in place of the space. The comma (,) separates the numbers by a number of blank spaces. What the comma (,) in a PRINT statement does, in fact, is TAB 15 blank spaces between what you want printed, thereby spacing what is printed into columns.

You can use the PRINT statement to print equations on your screen. The following demonstration program will help to illustrate this. Call it B:PRINT2.BAS. Enter the program:

```
VAR A,B,C=REAL
LET A=2Ø
LET B=3
LET C=A+B
REM This will print an equation.
PRINT A;" +";B;" =";C
```

This program uses a feature of the PRINT statement that prints material enclosed in quotation marks following a PRINT statement. Note the use of the semicolons (;). Run the program and see what happens.

You will see: 2Ø + 3 = 23. So, not only were the number values of A, B, and C printed, but also the plus (+) and equal sign (=), which were enclosed in quotation marks in the line containing the PRINT statement.

The REM and COMMENT Statements

Notice that, in the sixth line of the above program, you used the REM statement, which stands for REMark. This statement allows you to put a single line remark into the body of your program to remind you of what's going on in this part of the program.

For the same purpose, you can also use the COMMENT statement, which allows you to insert multiple line comments into the body of your program. COMMENT statements are used in the following general manner:

```
COMMENT
This is a comment, which can be any number of
lines long and can contain all characters:
!#$%^&*() and so on... It is used to remind the
programmer something about a certain portion
of a program, and, as you can see, it can go
on and on and on and on.....................
END
```

The COMMENT statement must have an END statement after the text of the comment is written. Now, it is important to understand that the REM and COMMENT statements appear only in the body of your program but will not be printed on the screen when you run the program.

Printing Strings

Not only will the PRINT statement print single characters on your screen, but it will also print groups of characters. We call this group of characters a string. To do this, enclose the characters you want to appear on your computer's screen in double quotation marks. Into a program file called B:PRINT3.BAS, enter the following:

```
VAR A,B=REAL
LET A=4
LET B=A*A
PRINT "This program will multiply four times itself."
PRINT
PRINT A;" times";A;" equals";B
```

Notice that the fifth line of this program is simply a PRINT statement, which produces a blank line on your screen when the program is run.

Also notice that in this and the last program, a blank space is included in the quotation marks before the word we want printed. This is because, when we print a string, there is a space automatically inserted after what you put in quotation marks, but not before. In other words, if the sixth line of the above program had been: PRINT A;"times";A;"equals";B , then what would have appeared on your screen would have been:

```
4times 4equals 16
```

INPUT Strings

The INPUT statement can also have strings after it. Write the following program to see what happens. Call it: B:PRINT4.BAS Enter the program:

```
VAR A,B=REAL
PRINT "This program multiplies numbers times themselves."
INPUT "What number do you choose";A
LET B=A*A
PRINT A;" times";A;" equals";B
```

When you run this program, you will see the material in the quotation marks following the PRINT statement in line two printed out along with the material in the quotation marks following the INPUT statement (plus a question mark) in line three. This is called a prompt. After you enter a number and press the RETURN

key, the multiplication will be performed, and you will see the results. If you choose the number 3, it should look like this;

```
This program multiplies numbers times themselves.
What number do you choose?3
3 times 3 equals 9
```

So, in this chapter, you have looked deeper into uses of the PRINT statement. In the next chapter, you will learn more about the LET and VAR statements.

Chapter 5

WORKING FURTHER WITH THE LET AND VAR STATEMENTS

You've been using the LET statement in most of the programs you've written so far. For example, you've written: LET Z=X+Y. The statement LET, however, is optional. You don't really need it for the statement to work. In other words, you could have written the above statement as Z=X+Y, and it would have worked just as well. From now on, we'll be writing LET statements without the LET at the start of the line.

You can write multiple operations (that is, addition, subtraction, multiplication, division, and so on) within a LET statement, but the computer will carry out certain operations before it will do others. For instance, if you write the line, Z=X+(Y-A), into a program, the computer will do what's in the parentheses before it does anything else. In other words, the computer will subtract A from Y before it adds the result of this subtraction to X.

The way in which the computer decides which operations come before others in a statement is called the rules of precedence. The table below gives these rules. The operations described on line 1) are performed by the computer before those operations in line 2), and those in line 2) are performed by the computer before the operations described in line 3), and so on. That is, the operations in line 1) have precedence over those in line 2). The operations which are on the same numbered lines below have equal precedence, meaning the computer won't try to perform one before the other.

Table of Precedence

1) operations enclosed in parentheses
2) ^ or ** (exponentiation)
3) making a number positive or negative (+ or -)
4) * (multiplication) and / (division)
5) + (addition) and - (subtraction)
6) = (equals), < (less than), > (greater than), <= (less than or equal to), >= (greater than or equal to), and <> or # (not equal to)
7) logical operations (NOT, AND, OR XOR, IMP, and EQV)

The VAR Statement

The work you have done with the VAR statement in the programs you've already written has so far dealt only with one variable type: REAL. There are, however, six variable types that you can use in S-BASIC programming. They are:

* REAL
* REAL.DOUBLE
* FIXED
* INTEGER
* STRING
* CHAR

1. The REAL variable type lets you use numbers with approximately 6 digits of accuracy, with a decimal point "floating" (that is, the decimal point positioned somewhere in the six digits). For example:

2.457 .00201 98.32 -4.1212 675.301

If the number is larger than 6 digits, then scientific notation is used to represent the number. For example:

5.123E+12 6.321E-7 2.7314E+11

E means "times ten to the power of." The first example above, for instance, would stand for 5.123 times ten to the twelfth power. The range of the exponent following the E is approximately +38 to -38.

2. The REAL.DOUBLE variable type is the same as the REAL variable type, except that numbers can have approximately 12 digits of accuracy, instead of 6. For example:

1.34567945628

The above number would be accepted as a number of the REAL.DOUBLE variable type, but, if you tried to use it as a number of the REAL variable type, it would be rounded off to:

1.34568

3. The FIXED variable type was designed for use when working in a dollars and cents format. You can use numbers with up to 8 digits to the left of the decimal point and 3 numbers to the right of the decimal point, although only 2 numbers to the right of the decimal point will be displayed. For example:

5.75 250.05 236.74 27431617.99

When a FIXED variable type number is printed, it is automatically formatted so that all decimal points will line up, which makes it useful for long lists of numbers. If a number is entered which is too large or small, an error message is given. You can use a number that has 3 digits to the right of the decimal point, but this will get rounded off to 2 digits when displayed by the computer using the PRINT statement.

4. The INTEGER variable type lets you use an integer (that is, a positive or negative number with no decimal point) in the range of: +32767 to -32767. If you use a number more positive or negative than these, it will simply wrap around in this range. For example, if you use a number bigger than 32767, this will wrap around into -32767 and upwards. You can also use hex numbers (hexadecimal numbers are base 16 numbers) in the INTEGER variable type. The last digit of the hex number must be an H. For example:

ØA6FH or 3CFH

5. STRING is a non-numeric variable type that may contain any group of characters (except ØØH) up to 8Ø characters long. You can extend or shorten this length, using the VAR statement, as will be described below. Strings are entered either enclosed in double quotation marks or by pressing the RETURN key after typing them in.

6. The CHAR variable type is a single character that can be entered either as a number (decimal or hexadecimal) or as a character enclosed in single quotation marks. For example:

```
5       12        'q'        ØBH        'p'          2FH
```

The VAR statement is used to establish what type a variable will be. The VAR statement should be close to or at the top of any program you write and always has to be before any LET or other statements that use the variables you want to specify. Generally, a VAR statement will resemble the following form:

```
VAR <name>=<type>
```

For example:

```
VAR X=REAL
VAR A=REAL.DOUBLE
VAR ACCOUNT=INTEGER
VAR YOURNAME=STRING
```

You can also name more than one variable of the same type in a VAR statement. For example:

```
VAR X,Y,Z=INTEGER
```

To avoid confusion, you should be careful not to mix variable types. For instance, you may write a program starting with:

```
VAR A,B=REAL
VAR X=INTEGER
A = 2.118
X = 5 + A
```

In the third line of the above program, A equals 2.118. The fourth line makes X equal to 5 plus A, but X has been designated as an INTEGER in line two, so that A is converted into an integer so the operation X = 5 + A can be performed. Therefore, the three digits to the right of the decimal point of A are dropped, and you lose that much accuracy. Or, if A had been too large a number, conversion into an integer could cause problems. Again, to avoid confusion and errors, don't mix variable types.

To understand the VAR statement further, write the following program. Call it: B:VARTYPE.BAS. Enter the program:

```
VAR X=REAL
VAR Y=REAL.DOUBLE
VAR A,B,C=FIXED
VAR I=INTEGER
VAR YOURNAME=STRING
VAR RESPONSE=CHAR
INPUT2 "Enter a five digit number and see what happens.";X
INPUT2 "Enter a longer number.";Y
INPUT2 "Enter a dollar and cents figure.";A
INPUT "Another dollar and cent figure";B
INPUT2 "Enter an integer.";I
INPUT "What's your name";YOURNAME
INPUT "What's your response character";RESPONSE
C = A + B
PRINT "This is your REAL number", X, YOURNAME
PRINT "This is your REAL.DOUBLE number", Y, YOURNAME
PRINT "These are your dollar and cent figures:"
PRINT A
PRINT B
PRINT "Added together:"
PRINT C
PRINT "This is your integer", I, YOURNAME
PRINT "And your response, ";YOURNAME;", is: ";RESPONSE
```

The above program is a massive program, because all the variable types are used. This is an example program only, and when you write programs, it would be best to select only those variable types that are necessary for a particular program.

Note that, when you enter an input for a CHAR variable type, as for the variable RESPONSE above, you don't need to press the RETURN key. For all other variable types, you have to press RETURN after you enter your input.

Play around with this program for a while until you become more familiar with the six variable types.

Different Forms of the INPUT Statement

Notice that some of the INPUT statements in the above program have a 2 after them. There are four options for using the INPUT statement:

1. INPUT - generates a question mark and returns to the next

line on the screen after an input has been entered.

2. INPUT1 - generates a question mark but doesn't return to the next line after an input is entered.

3. INPUT2 - no question mark is generated but does return to the next line after an input is entered.

4. INPUT3 - no question mark is generated and no return to the next line after an input is entered.

So, in this chapter, you have been introduced to the six variable types as well as the rules of precedence for LET statements. In the next chapter, you'll learn some statements that redirect the flow of execution of a program.

Chapter 6

THE IF...THEN, GOTO, AND GOSUB STATEMENTS

An S-BASIC program normally is executed from one line to the next. For example, say you have written the following program:

```
VAR X = STRING
VAR Y = INTEGER
INPUT "What is your name";X
INPUT "What year were you born in";Y
Y = 1983 - Y
PRINT "So, ";X;", you are";Y;" years old, right?"
```

The way the computer will execute this program, once you have compiled and are running it, is to execute the first line, and then go on to the second line. After executing the second line, it will go on to the third, and so on.

This normal flow of execution can be controlled and redirected, using a variety of statements. The first such statement we'll look at is the IF...THEN statement.

The IF...THEN Statement

Create a new file called: B:IFTHEN1.BAS. Enter the following program:

```
VAR X = STRING
VAR Y,Z = INTEGER
INPUT "What is your name";X
INPUT "What year were you born in";Y
Y = 1983 - Y
PRINT "So, ";X;", you are";Y;" years old, right?"
INPUT "In the year 2000, you'll be how many years old";Z
IF Z = Y+17 THEN PRINT "Right!"
```

Run the program a few times using different answers to the third INPUT statement. You'll notice that when you answer correctly the word Right! will be printed, but when you answer incorrectly, Right! isn't printed. This is because of the IF...THEN statement.

For instance, say you input the following:

```
What is your name? Bowb
What year were you born in? 1958
So, Bowb , you are 25 years old, right?
In the year 2000, you'll be how many years old? 42
 Right!
```

In this case Y = 1983 - 1958, which equals 25. The last line of your program is saying that IF Z = Y+17, THEN PRINT "Right!" So, if your input for Z equals Y+17 (25+17 or 42), then the word Right! will be printed. If your input for Z does not equal Y+17, then Right! won't be printed.

There is another statement that can be used with an IF...THEN statement, and that is the ELSE statement. Call up the last program file you created (B:IFTHEN1.BAS), and change it as follows:

```
VAR X = STRING
VAR Y,Z = INTEGER
INPUT "What is your name";X
INPUT "What year were you born in";Y
Y = 1983 - Y
PRINT "So, ";X;", you are";Y;" years old, right?"
INPUT "In the year 2000, you'll be how many years old";Z
IF Z = Y+17 THEN PRINT "Right!" ELSE PRINT "Wrong!"
```

Only the last line has been altered by adding the ELSE statement. Run the program a few times and see what happens. Input different answers for the third INPUT statement. You will see that, if your input satisfies the conditions of the IF...THEN statement, then the word Right! will be printed, but if your input doesn't satisfy these conditions, then the ELSE statement will be implemented, and the word Wrong! will be printed.

The BEGIN...END Statement

Not only can PRINT statements be used in an IF...THEN statement, but other statements can be used as well. Also, a group of lines can be used, using the BEGIN...END block structure. For example:

```
IF A > B THEN
   BEGIN
      B = B + 1
      INPUT "Sorry, pal.  Try again";A
   END
```

In this example, the two lines between BEGIN and END are treated as a single statement. When using the BEGIN...END block structure, it is important to keep track of where variables are declared and used. Consider the following program:

```
10  VAR A,B=INTEGER
    B=100
    INPUT "Pick a number";A
----IF A > B THEN BEGIN
       VAR X=INTEGER
b      X = A - B
1  b---IF X > 50 THEN BEGIN
o  1      VAR Y=INTEGER
c  o 2    Y = A/10
k  c    PRINT "Your number was too high. ";Y;" would be better."
   k------END
1      X = A/5
       PRINT "Your number was too big.  Try something more like ";X
-------END
    VAR Y = STRING
    PRINT "That number is almost right.  Still, I need another one."
    INPUT "Want to try again";Y
    IF Y = "Y" THEN GOTO 10
```

In the above program, there are two BEGIN...END block structures. The designations, block 1 and block 2, are not part of the program; they are just included for the purpose of clarification. Before the first BEGIN...END structure (block 1), the variables A and B are declared as INTEGERs. These will stay as they are throughout the program. At the beginning block 1, the variable X is declared as an INTEGER. This will only stay valid within block 1. Similarly, in the second BEGIN...END structure (block 2), which is nested in block 1, the variable Y is declared as an INTEGER. This is only valid within block 2. Notice that in the last few lines, Y is again declared as a variable. Since block 2 has been executed, and the computer is done using the variable Y declared there, we can use the name Y again outside of that block as a name for another variable, this time as a STRING.

A and B are global variables, meaning global to the entire program. X is a local variable and is local to block 1 and global to block 2.

Also notice the GOTO statement in the last line of the above program. This is another extremely useful statement when working with the IF...THEN statement.

The GOTO Statement

The GOTO statement simply tells the computer to go to a certain line in your program that you have specified. In the following program, you'll combine the GOTO statement with the IF...THEN statement.

Create a new file called: B:IFTHEN2.BAS. Enter the program:

```
      VAR I=CHAR
10    PRINT "ROW, ROW, ROW YOUR BOAT"
      PRINT "GENTLY DOWN THE STREAM"
      PRINT "MERRILY, MERRILY, MERRILY, MERRILY"
      PRINT "LIFE IS BUT A DREAM..."
      PRINT
      INPUT "DO YOU WANT TO SING AGAIN";I
      IF I='Y' THEN GOTO 10
      PRINT
      PRINT "HAPPY DREAMS THEN."
```

In this program, only if you input a Y will the verses repeat. The IF...THEN statement sets up the condition that IF I='Y' THEN GOTO 10. Notice that you had to specify which line you want to be line 10 by entering it into your program. It would be useful to indent each line of your programs with the TAB key from now on, so that you have room to number your lines, so you can use GOTO statements. Also, you don't have to call your lines by numbers. You could GOTO 0LOOP or GOTO 0NAME just as well as GOTO 10 or GOTO 100. And, lastly, you can use an IF...THEN statement without using the GOTO statement, and it will do the same thing. For example, in the program above, you could have written:

```
      IF I='Y' THEN 10
```

This would do the same thing as: IF I='Y' THEN GOTO 10.

Also, you could have written: IF I THEN 10
If I equals Y or T, then the condition is satisfied. This is a built-in feature for the CHAR variable type.

The GOSUB Statement

The GOSUB statement is like the GOTO statement, except that after the computer is done doing the subroutine, it will come back to the line after the GOSUB statement by using the RETURN statement.

Here is an example. Suppose you wanted to write a game in

S-BASIC, and in various parts of the game you want to give the player the option of exiting as follows:

```
100   INPUT "TO EXIT PRESS E...";X
```

You could repeat this line a number of times or use a GOTO 100 statement, but you may want to go back to where you were after executing line 100. The way to do this is with a GOSUB statement.

Say you're at a point in your program where you want to ask the user if he wants to exit. You'd write:

```
      GOSUB 100
```

The computer would go to line 100 and execute it:

```
100   INPUT "TO EXIT PRESS E...";X
      IF X = 'E' THEN RETURN
```

The GOSUB statement remembers where it came from. The RETURN statement tells the computer to go back to the statement after the GOSUB statement. Your program might look something like this at that point:

```
      GOSUB 100
      If X = 'E' THEN PRINT "SO LONG, BUDDY!"
```

So, the RETURN statement would direct the computer to the line after the GOSUB statement that sent it to line 100. That is, the second line above.

Also, the GOSUB statement can direct the computer to execute anything from one line to a whole block of lines, the only condition being that a RETURN statement must be at the end of whatever you want that part of the program to do. This block of lines ending in a RETURN statement is called a subroutine and is very useful when you have something you want to repeat a number of times.

Concerning the GOTO and GOSUB statements, it is important to note that S-BASIC incorporates various powerful statements which can be used instead of repeatedly using GOTO or GOSUB statements. You will be introduced to several of these statements in the next two chapters.

So, in this chapter you have learned the IF...THEN...ELSE statement as well as the GOTO and GOSUB statements. In the next chapter you'll learn some other statements which will repeat some parts of your programs one or more times.

Chapter 7

THE REPEAT...UNTIL, WHILE...DO, CASE, AND FOR...NEXT STATEMENTS

The REPEAT...UNTIL Statement

The REPEAT...UNTIL statement will repeat a statement or a block structure (using BEGIN...END) until a certain expression is true. As an example, create a file called: B:REPEAT.BAS. Enter the following program:

```
VAR X=REAL.DOUBLE
VAR A,Y,Z=INTEGER
A=10
X=10
Z=1
PRINT "This program will print out powers of two in binary numbers."
PRINT
INPUT "Where should it stop (at what power)";Y
REPEAT BEGIN
    PRINT "2 to the power of";Z;" =";X;" in binary numbers."
    Z=Z+1
    X=X*A
    END UNTIL Z=Y+1
```

Run the program, and enter 7 for your input. Notice that you get a printout of 2 to the power of 1 up to 2 to the power of 7, and that the REPEAT...UNTIL statement will repeat until Z=Y+1, when the condition of the UNTIL statement is satisfied. When this condition is satisfied, in this case when Z=8, the computer won't be directed back to the REPEAT statement and won't, therefore, perform block structure that follows the REPEAT statement. So you won't, in this example, get a printout of 2 to the power of 8.

You can GOTO out of the REPEAT...UNTIL statement or RETURN from it back to a GOSUB, in case this is necessary.

The WHILE...DO Statement

The WHILE...DO statement works much in the same way as the REPEAT...UNTIL statement, yet it is somewhat different. Create a file called B:WHILEDO.BAS. Enter the following program:

```
      VAR I=FIXED
      INPUT "How much money do you have to work with";I
10    WHILE I > 1.00 DO BEGIN
          PRINT "For each transaction, more money must be taken."
          I = I/2
          PRINT
          PRINT "You now have $";I;" left."
          GOTO 10
          END
      PRINT "Sorry, Charlie, but you only have $";I;" left."
      PRINT "You should make a visit to the bank."
```

The WHILE...DO statement in the program above will do everything in the BEGIN...END block structure while I is greater than 1.00. Notice that if you originally input 1.00 as "How much money do you have to work with," then immediately you would have seen the message on your screen:

```
Sorry, Charlie, but you only have $       1.00 left.
You should make a visit to the bank.
```

In other words, with the WHILE...DO statement, the statement or block following DO may never be executed if the conditions set forth after WHILE aren't met. On the other hand, using the REPEAT...UNTIL statement, the statement or block following REPEAT will be executed at least once, since the conditions to be met don't occur until after the UNTIL statement.

As in the REPEAT...UNTIL statement, you can GOTO or RETURN out of a WHILE...DO statement, if necessary.

The CASE Statement

Another statement of the same type is the CASE...OF statement. It allows you to select a single statement or block structure (using BEGIN...END) from a group of many such statements or blocks.

Create a file named: B:CASE.BAS. Enter the following:

```
VAR RESPONSE=STRING
INPUT2 "Pick an animal (CAT, DOG, BIRD, or MAN)";RESPONSE
CASE RESPONSE OF
   "CAT":    PRINT "Meowww..."
   "DOG":    PRINT "Bowowow..."
   "BIRD":   PRINT "Tweet, tweet..."
  "MAN":    PRINT "Four score and seven years ago..."
  RESPONSE: PRINT "Please enter CAT, DOG, BIRD, or MAN!"
END
```

Try running the above program a few times, entering different inputs for RESPONSE each time. What the CASE statement does is evaluate your input and compare it with each expression between the CASE and END statements. In this case, your input is compared to CAT, DOG, BIRD, and MAN. If your input is equal to any of these expressions, then the statement following that expression is executed. If your input is not equal to any of these expressions, the eighth line, included as an otherwise clause, will be executed. It is important, when using the CASE statement, to have an END statement after the last expression and statement or block following the CASE statement, or the computer won't know when to stop its evaluations.

Unlike the REPEAT...UNTIL and WHILE...DO statements, you cannot GOTO or RETURN from within a case statement.

The FOR...NEXT Statement

Another useful statement is the FOR...NEXT statement. Create a file called: B:FORNEXT.BAS. Enter the program:

```
VAR X,Y=INTEGER
Y=99
FOR X = 1 TO 99
    PRINT
    PRINT Y;" bottles of beer on the wall,"
    PRINT Y;" bottles of beer."
    Y=Y-1
    PRINT " If one of those bottles should happen to fall,"
    PRINT Y;" bottles of beer on the wall."
    PRINT
NEXT X
```

Before you run this program, you should know about a command for the computer that will stop the printing on the screen. This is useful when you are running long programs, because only a certain amount of printout will fit on one screen, and your program, when running, may be longer than this, such as the above example. In such a case, the screen will continue printing all the material, yet it will scroll by so quickly that you probably would have trouble reading it. To stop the computer printing on the screen at any point, press the CTRL key and the S key at the same time. This will allow you to read a screen full of material before scrolling forward. To start the computer scrolling again after you have stopped it, simply press another CTRL-S.

Run the program above as an illustration of this scrolling. Remember, CTRL-S will stop the scrolling, and pressing CTRL-S again will start it up again.

Another useful command, which you can use while your program is running, is the trace command. To start the tracing while the program is running, press the CTRL key and the T key at the same time. Try this while running the above program, and see what happens.

What the trace command does is display the line numbers of your program as they're executed. To turn off the trace command, enter another CTRL-T.

The FOR...NEXT loop in the above program will simply repeat itself until X has gone from 1 to 99. X automatically is increased by 1 each time.

You can, however, control the value added to your variable for each repetition of the loop by using STEP. For instance:

```
FOR A = Ø TO 1ØØ STEP 5
```

This will count from 1 to 1ØØ in steps of 5 (that is, 5, 1Ø, 15, 2Ø, etc.). Some more examples:

```
1.    FOR I = 1 TO 1Ø STEP .Ø1

2.      FOR N = 2Ø TO 1 STEP -1

3.    VAR LETTER=CHAR
      FOR LETTER= 'A' TO 'Z'
```

In example number one, I will be increased from 1 to 1Ø in steps of .Ø1. In example two, N will be decreased from 2Ø to 1 by -1 for each repetition of the loop. In example three, LETTER will be advanced through the letters of the alphabet from A to Z. Remember that, for each FOR statement, you need an equivalent NEXT statement somewhere after it.

As with the CASE statement, you cannot GOTO or RETURN out of a FOR...NEXT loop.

You've learned quite a bit in this chapter. The next chapter will deal with FUNCTIONs and PROCEDUREs. You are encouraged to experiment with each of the statements presented in the order in which they are given until you are familiar with them. The sample programs should be some help, but you are, of course, not limited to writing only these.

Chapter 8

THE FUNCTION AND PROCEDURE STATEMENTS

FUNCTIONs

FUNCTIONs and PROCEDUREs are statements which you define. Say that, for example, a program you are writing requires that a variety of numbers be cubed, that is, N x N x N. Create a file called: B:FUNCTION.BAS. Enter the following program:

```
FUNCTION CUB(I=INTEGER)=INTEGER
END=I*I*I
VAR X,Y,Z,A,B,C=INTEGER
INPUT "Pick a number";X
INPUT "Another";Y
INPUT "Yet another";Z
A=CUB(X)
B=CUB(Y)
C=CUB(Z)
PRINT
PRINT X;" cubed equals ";A
PRINT
PRINT Y;" cubed equals ";B
PRINT
PRINT Z;" cubed equals ";C
```

Run this program to see if it works.

The FUNCTION statement is set up with the name of the function after the word FUNCTION. In the above case, the name used was CUB. Then, one or more variables are declared within parentheses after the name. In the example above, the variable is (I=INTEGER). Next, you write the type of variable that the result of your function will be. In the above example, the whole CUB function will result in an integer, =INTEGER. Lastly comes the END statement which is the end of the function and will determine the result of the function. As an example:

```
FUNCTION CUB(I=INTEGER)=INTEGER
   VAR Z=INTEGER
   Z=I*I*I
   END=Z
```

PROCEDUREs

PROCEDUREs are very similar to FUNCTIONs. PROCEDUREs might be best explained as being like named subroutines. There is no type declared for the PROCEDURE and no result, as there is in FUNCTIONs. Create a file called: B:PROCEED.BAS. Enter the program:

```
      PROCEDURE CLEAR.SCREEN
         PRINT CHR(26);
      END of CLEAR.SCREEN

      VAR A=INTEGER
      VAR B=CHAR
      VAR X=STRING
      INPUT "What's your first name";X
       FOR A=1 TO 200
          PRINT X;
          NEXT A
05     INPUT2 "When you want to clear the screen, enter Y.";B
       IF B <> 'Y' THEN 05
       CLEAR.SCREEN
       PRINT "The screen, it seems, has been cleared."
```

In the above program, the first three lines are the PROCEDURE, which we called CLEAR.SCREEN. So, when you call up this newly-defined statement later in the program, the screen will be cleared. As in the FUNCTION statement, the PROCEDURE statement must be concluded with an END statement. And it would be best if you place both FUNCTIONs and PROCEDUREs at the beginning of programs. Another example of a PROCEDURE:

```
PROCEDURE PRINT.SUM (A,B,C=INTEGER)
   PRINT A+B+C
END
```

Variables declared within both PROCEDUREs and FUNCTIONs are valid only within those statements, just as in a BEGIN...END statement described previously.

You have done a lot with S-BASIC now that you've gotten this far. You are, in fact, no longer a beginner and will probably be wanting more specific information and details that are not offered in this part of the manual. Go on and read the S-BASIC reference portion of this manual, and you should find what you need to write programs to solve more complex problems.

TABLE OF CONTENTS

INTRODUCTION TO S-BASIC

S-BASIC has features for the most advanced and comprehensive applications, but it can be used by the beginner. Read the manual; use what applies to your situation. If you have questions, consult a book or two that covers programming in general terms. If you have further questions, write us at the address given on the title page.

FEATURES

S-BASIC means Structured BASIC. It:

* Is a high-level language for use in the CP/M environment.
* Maintains the flexibility of BASIC, yet includes the power of advanced structured techniques.
* Is a true compiler (translator) that converts the user's program directly into machine language for the computer to run. This gives the user speed and power not available from an interpreter or a compiler/interpreter combination.
* Has source line information and program trace control to make easy the writing and debugging of programs.
* Has a rich assortment of variable types to allow the use of:

    single characters
    large strings of characters
    two types of floating point
    integer
    fixed point

* Includes long multiline comments in the source
* Documents all variables

MACHINE LANGUAGE

Microcomputers recognize their own special, low-level machine language.

Machine language is the instructions that the computer follows to do something simple, such as add, subtract, multiply, and divide. But, it cannot handle numbers as you understand them.

More complex operations are done by a long series of the simple operations.

COMPILER (TRANSLATOR)

The Z80 cannot understand BASIC.

The compiler (translator) compiles (translates) your BASIC program into machine language.

This compiled (translated) program is called by several names, such as:

code, run-time code, run-time, object code, machine code, and COM file.

No matter what it is called, it is machine language. It is what the Z80 can execute (understand) directly.

BASIC

BASIC is a high-level computer language.

High-level languages were invented so that humans would not have to bother with the tedium of low-level machine language, which consists of zeros and ones.

The same high-level language can be found on different computers, because different compilers can be written to translate a high-level language into different low-level machine languages.

PROGRAMMING

BASIC has different versions, because there is a difference of opinion on which features should be included.

Programming in high-level language can be thought of as two tasks:

* The understanding of programming ideas
* The implementation of those ideas in a specific high-level language.

Once you understand the basic programming ideas, learning a new programming language is simple. Once you have learned to program in S-BASIC, you could easily learn other programing languages, such as PASCAL, FORTRAN, C, or ALGOL.

You might want to pick up an introductory book on programming to give you the concepts of structured programming. Some of the concepts which you might tackle when you have learned your BASIC are: procedures, functions, local variables, and logical blocks.

As you read your BASIC books, be aware of the differences between S-BASIC and the BASIC used in the book. Major differences are:

* Variables must be declared, using the VAR statement.

* Line numbers are only needed where referenced by GOTO and GOSUB.

* Disk files will probably be different, but S-BASIC supports all major forms of disk files, so an analagous method of disk I/O will be found in S-BASIC.

AN INTRODUCTION TO S-BASIC FOR THE EXPERIENCED PROGRAMMER

S-BASIC is a structured language built on a BASIC syntax foundation. This leads to a clean and easy-to-use syntax.

There are six data types:

Single and double precision binary floating point
Fixed-point BCD
Integers
Characters (bytes)
Strings

There are three basing modes:

Data
Common
Run-time specified
(i.e., dynamically positioned in RAM programatically)

All major structure statements are implemented:

PROCEDURE
FUNCTION
BEGIN AND END
IF THEN ELSE
WHILE
REPEAT
CASE
Local variables

There are procedures, functions, and logical blocks.

The language is extendable, using procedures and functions.

Serial and random files are provided. Both can be ASCII or binary.

Random files have three access modes:

* Buffer-directed

* Record sequential with auto rewind to beginning of record on re-read of random record

* Record sequential without auto rewind

An ability to search and include from a source library at compile time is provided.

The TEXT statement is provided to facilitate the construction of menus and help text in a program.

The COMMENT statement allows large blocks of comment text without redundant REM statements.

There are many more unique features in S-BASIC. They are covered elsewhere in the manual.

GETTING STARTED

To program in S-BASIC, you will need four files from the CP/M S-BASIC diskette:

SBASIC.COM
OVERLAYB.COM
BASICLIB.REL
USERLIB.REL

The S-BASIC beginner's section fully explains how to set up your program diskette.

To start compilation of a program, enter: SBASIC <name>.BBX

This is also fully explained in the beginner's section.

When you write a program in S-BASIC, always store it on a diskette in drive B, and always give it the extension, BAS.

NOTATION

LINE NUMBERS

In general, an S-BASIC statement would be represented with a line number and a basic statement, but line numbers are optional; they need to appear only where a reference is necessary.

A valid line number is a digit, 0, 1, 2, ... 9, followed by ASCII characters that are not reserved. (Reserved characters are: ;:[]{}<>(),"=#-+*/%~)The following are valid line numbers:

```
0000   0025   0746   0test.routine
```

(Be sure to leave no space between the zero and "test.routine".)

If numbers only are used, they need not be in order.

STATEMENTS

A statement may be upper case or lower case ASCII characters, as the compiler will convert all lower case into upper case.

In this manual, when <statement> is called for, no line number is necessary.

When <statement> is called for, the following construction may be substituted in its place:

```
BEGIN
  <body>
END
```

Where BEGIN and END form a frame around <body>, <body> may be any number of basic statements, including nested BEGINS and line number references.

Key words (things that must be included in a statement) will be printed in capital letters.

Key words may not appear by themselves in any other way in the program, because they have special meaning.

Spaces, where given, must be included, so that a key word may be included in a variable name without being confused with the variable.

Angle braces < > enclose an item that must be included in the statement.

Brackets [ ] enclose an item that is optional.

Braces { } enclose an item that may or may not be included and/or repeated.

Three dots ... indicate an item that may repeat.

A slash mark / should be read as "or". Example: this/that = this or that

The words "run-time" refer to the compiled object code produced by the compiler.

Physical Length of Lines

The input buffers of the compiler are token-oriented. This means there is no limit to how long a physical line can be.

However. sometimes the physical line length must be restricted. A backslash (\) can be used to continue a logical line on to the next physical line.

Everything from the backslash to the end of physical line is ignored, i.e., you can put a comment after the backslash.

Example:

```
INPUT     VAL1, VAL2, VAL3\
          VAL4, VAL5, VAL6
```

or another use:

```
     PRINT     FIRST.NAME,          \PERSONS FIRST NAME
               LAST.NAME,           \PERSONS LAST NAME
               PHONE.NUMBER         \PERSONS PHONE NUMBER
```

KEY WORDS

	ECHO	LOCATION	SQR
$CONSTANT	ELSE	LOG	STEP
$INCLUDE	END	LOGIC	STOP
$LINES	EQV	LPRINTER	STR$
$LIST	EXECUTE	MID	STRING
$LOADPT	EXP	MID$	STRING$
$PAGE	FCB	NEXT	SUB
$STACK	FCB$	NOT	TAB
$TRACE	FFIX	NUM$	TAN
ABS	FILES	OF	TEXT
AND	FINT	ON	THEN
ASC	FIX	OPEN	TO
ASCII	FOR	OR	UNTIL
ATN	FRE	OUT	VAL
BASE	FUNCTION	PEEK	VAR
BASED	GO	POKE	VARIABLE
BEGIN	GOSUB	POS	WHILE
CALL	GOTO	PRINT	WRITE
CASE	HEX$	PROCEDURE	XLATE
CHAIN	IF	READ	XOR
CHR	IMP	REM	
CHR$	INITIALIZE	REMARK	CONTROL.C.TRAP
CLOSE	INP	RENAME	RECORD.SEQUENTIAL
COM	INPUT	REPEAT	
COMMENT	INPUT1	RESTORE	
COMMON	INPUT2	RET	
CONSOLE	INPUT3	RETURN	
COS	INSTR	RIGHT	
CREATE	INT	RIGHT$	
DATA	LEFT	RND	
DELETE	LEFT$	SGN	
DIM	LEN	SIN	
DIMENSION	LET	SIZE	
DO	LOCATE	SPACE$	
		SPC	

COMMENTS

Single line comments may be implemented using the REM statement:

REM/REMARK <text>

where <text> is a single line comment or remark.

Multiple comments may be inserted into the text as follows:

```
COMMENT
 <text>
END
```

where <text> may be any number of lines made up of any type of characters.

In both the REM and COMMENT statements, <text>, REM, COMMENT, and END are ignored by the compiler and produce no code. Thus, they may be as long and as frequent as desired without affecting the the operation or size of the program at run-time.

Examples:

```
REM This is a comment line
 REMARK This is another comment line
 COMMENT
  This is a comment line that can
  be any number of lines long and can
  contain all characters *#$%&() and so on ...
  a COMMENT is ended by a line that starts with the word END
  like so:
 END
```

DATA TYPES

There are six variable types from which the programmer can choose. This allows for the best selection between storage and accuracy. The six types are:

Type	Abbreviations Used in the Manual Only
REAL.DOUBLE	RD
REAL	RS
FIXED	F
INTEGER	I
STRING	S
CHAR	C

REAL.DOUBLE and REAL

This type provides the user with binary floating point numbers. There are approximately 6 digits of accuracy with the type REAL and 12 digits with the type READ.DOUBLE.

When these numbers are printed, the following rules apply:

* If the number will fit (meaning, 6 digits for REAL and 14 digits for REAL.DOUBLE), then the number will be printed with the decimal point 'floating'. For example:

  -23.003 5.023 .0002 -1000.1

* If the number will not fit, then scientific notation will be used to represent the number times a power of ten.

  5.123E12 6.321E-7

  Here the letter E reads as "times ten to the power of". The range of the exponent is approximately +38 to -38.

  In general, the forms for these types are represented below with the letter:

  S standing for the algebraic sign,
  D standing for a digit from 0 to 9, and
  E standing for the power of ten function, if needed.

RD	SD.DDDDDDDDDDESDD	or	DDDDDDDDDDDDD Floating '.'
RS	SD.DDDDDESDD	or	DDDDDD with the '.' anywhere

Constants may be entered into the compiler, and at run-time, using either the "floating point" or "scientific" form. If the sign is positive, it is printed as a space.

FIXED

Provides the programmer with a packed BCD number providing 11 digits of decimal accuracy.

This is decimal math, so there is no rounding problem as is inherent with floating point math. In general, for long summing operations, the fixed point format gives the best results. Indeed, it was designed for business use.

The form of a fixed point number is:

SDDDDDDDD.DDD

where there are 8 digits to the left of the decimal point and 3 digits to the right of the decimal point.

When a fixed point number is printed, it is automatically formatted so that all the decimal points will line up. This is done as follows:

1. .005 is added to the number. This is done for display purposes only.

2. A leading space is printed.

3. Leading zeros are replaced with spaces, and the sign is printed (right justified) as a space, if positive, and as a "-", if negative. The digits up to the "-" are then printed.

4. The decimal point is printed.

5. And the next TWO digits are printed. The effect is a printout in dollars and cents format.

Fixed point numbers may be entered into the computer at run-time or as constants to the compiler in a free format similar to that of type REAL. The main difference is that no E is allowed, and a check is made for a number too large or too small. If such a number is entered, then an error message is given. If the error is encountered at run-time, the user is asked to re-enter the number before the program continues.

When printing a fixed type number set >= 99999999.995, an overflow warning is given. This is because, before a fixed-point

number is printed, .005 is added to it for rounding. This addition does not occur in a PRINT USING statement.

INTEGER

Its range is +32767 to - 32767

An INTEGER is stored as a word (2 bytes) integer in twos complement form. There is no check made for overflow. The number simply wraps around.

Integer and character division truncates the answer. No rounding is performed.

Constants to the compiler may be entered as decimal numbers or as hex constants.

Hex constants must have as the first digit a decimal digit from 0 to 9, and the last digit should be an H; for example, 0A6FH or 80H.

When numbers are entered at run-time, they may be either decimal or hex.

When hex numbers are entered, it is not necessary for the first digit to be a decimal digit, and a sign is not allowed.

Otherwise, the rules are the same as when they are entered as constants.

STRING

A STRING variable may contain any type of ASCII character except for an ASCII null (00H).

The length of this string of characters may be from 0 to the maximum length of a given string variable. This maximum length is set using the VAR statement described below.

Constants should be entered enclosed in double quote marks.

```
"This is a string constant"
```

Strings may be entered at run-time either enclosed in quotes or terminated with a "return". For more information on entering strings, see the section on input and output.

There are two right string functions, RIGHT and RIGHT$. These functions are different.

CHAR

A CHAR (character) is a single ASCII character that may assume any ASCII or 8-bit value assigned to it.

A character constant is entered in the compiler as a decimal number, a hex number, or as a single character enclosed in single quotes. For example:

```
5  OCH   'a'
```

For the rules governing the input and output of characters during run-time, see the section on input and output.

When defining the data type, CHAR, the word, BYTE, may be used in its place. This is the same data type, but it will help to make clear the intended use of the variable.

When the symbol <type> is used, it refers to one of the six types described above. All six types have three different possible locations in the run-time package (i.e., the compiled code). They may be:

* located in the data storage area set aside by the compiler
* set into the common storage area where they will not be changed during chaining (See the section on chaining.)
* not assigned any storage or location. This will be done by the BASIC program at run-time.

When using characters as numeric values, or when mixing with integers, the values assumed are equal to, or greater than, Ø and equal to, or less than, 255.

Integer and character division truncates the answer. No rounding is performed.

SUGGESTED PROGRAM STRUCTURE

Common variables
Variables global to the total program
Functions and procedures
Variables global to main program only
Main body of program
Data statements

VARIABLES

In the following statements:

<name> refers to the ASCII name given to the variable.
The first letter of <name> must be a letter.
This may be followed by ASCII characters that are not reserved.
<name> may not be a reserved word, though it may contain one.

To declare variables, the variable statement is used:

```
VAR <name>{,<name>}{/[;<comment><new line>}]=<type>[:size]
```

For example:

```
VAR                 ;This is a comment.
     First.name     ;Another comment
     Last.name      ;Another comment line
     X, Y, Z        ;Some more variables
= Integer

VAR A, B    ;A comment line of text
    C, D    =Byte
```

Note: Byte is the same as CHAR.

Variables must be declared as to their type and location in memory. The following statements are used for this purpose.

Data Storage Area

The following statement sets variables into the data storage area:

```
VAR <name>{,<name>}{/;<comment><new line>}=<type>[:<size>]
```

If the <type> is string, a maximum length may be given, using the :<size> argument.

If <size> is omitted for <type>=STRING, then a default length of 80 characters is used for the length of the string.

<size> may range from 1 to 255 characters. For example:

```
FIRST.NAME,LAST.NAME=STRING:30
```

This creates two string variables: FIRST.NAME and LAST.NAME.

The maximum length that each of these two strings can reach is 30

characters.

When an integer constant is called for, a compile-time symbolic constant can be used. Its use is to specify the <size> of a string symbolically when used with $INCLUDE. For further use of this, see section on $CONSTANT.

The programmer should be aware of a string's internal format:

<length><ASCII characters>[<Ø byte>]

<length> is the maximum length the string can obtain.

<length> is a byte value in memory.

<ASCII characters> is the string of characters that make up the string, one byte per character.

<Ø byte> flags the end of the string in memory IF the maximum length has NOT been reached.

Thus, from the above VAR statement, FIRST.NAME would look like this in memory if 'tom' was stored in it for 3Ø bytes:

<3Ø><t><o><m><0><0><0>...

When a base-located string is positioned in memory, the <length> byte "follows" the string around in memory. The based statement sets up the string length only. The string variable should not be used until the variable is positioned in memory. Failure to do so may destroy other data in memory.

Some more examples:

```
VAR X,Y,Z=REAL
VAR ACCOUNT=INTEGER
VAR BALANCE,DEBIT,CREDIT=FIXED
VAR RESPONSE=CHAR
```

These statements allocate storage into the data storage area. They create the real type variables, X, Y, and Z. They also create an integer named ACCOUNT, three fixed type numbers called BALANCE, DEBIT, and CREDIT, and a variable called RESPONSE of type char.

Common Storage Area

Variables may be allocated to the COMMON STORAGE area with the statement:

```
COM/COMMON <name>{,<name>}=<type>[:<size>]
```

The operation of this statement is the same as for the VAR statement.

Base-located Area

Variables that are assigned no storage or location are called "base located". They are declared using the following statement:

```
BASED <name>{,<name>}=<type>[:<size>
```

The operation of this statement is the same as for VAR and COM. For example:

```
BASED X, Y = INTEGER
```

BASE <name> AT <expression>

<expression> is of type integer. This statement positions the variable <name> at the memory location given by <expression>. For example, this could be in a disk I/O buffer or memory-mapped I/O location. Any type of variable, not just strings, may be based. For example:

```
BASE X AT Z + 12
```

Arrays

Arrays must be declared using the following statement:

```
DIM/DIMENSION [COM/BASE] <type>[:<SIZE>} <name>(<size>{,<size>})...
```

If COM is given, then the array resides in the common storage area.

If BASE is used, then the array is assigned no location or storage.

If both are omitted, then the location is in the standard data area.

<type> specifies the type for each <name> in the statement.

<name> is the name given to the array, and the same rules apply for it as for <name> in a variable statement.

<size> gives the number of elements in each dimension (vector) of the array.

As before, <SIZE> refers only to strings. Some examples:

```
DIM BASE CHAR  VIDEO.DISPLAY(80,24)
DIM REAL X(5,5,7) Y(20) ALPHA.VECTOR(3)
DIM COM STRING:30 NAME(3)
DIM STRING:X+Y; FIELD (A+J,C)
```

The DIM statement (for the data field only) may be used to change the <size> argument(s). The same array name may be used more than once in several DIM statements. This does not create two arrays, but rather changes the size of the array. The number of size arguments (dimensions) must remain the same.

```
DIM <type> <name> (<size exp>{,<size exp>})...
```

When an array is redimensioned, its data contents are destroyed. If the <type>=STRING, then the syntax for the expression giving the maximum string length is:

```
STRING:  <Integer expression>;
DIM STRING:MAX.LEN+Q; NAME(X-1)
DIM COM STRING:32 COMMON.NAME(20)
```

{The semicolon (;) is not allowed for arrays that are BASE or COMMON.}

A base-located array may be positioned in memory, using the following statement:

```
LOCATE <name> AT <expression>
```

where:

<name> is the name of an array declared, using a DIM BASE... statement.
<expression> is an integer expression which specifies where in memory the array is to be positioned.

Experienced programmers may find the information in the following table helpful. Beginners may ignore it or find more information from other sources.

ARRAY CONTROL STRUCTURE
(SPEC)

Memory	Use
Word	Address of array data
Byte	# of dimensions
Byte	Size of each element
Byte	Reserved
Word 1	Dope vector (size of a dimension)
[Word2	Dope vector
.	
.	
.	
Wordx	Dope vector]

Finding a Data Structure at Run-time

The location of a data structure may be found at run-time with this statement:

```
LOCATION VAR/ARRAY/SPEC/FILE <name1>=<name2>
```

In this statement:

VAR refers to a variable,

ARRAY refers to an array, and

SPEC refers to the array control field set up by the DIM statement.

FILE refers to the location of a disk I/O buffer. For further information on file buffers, see the section on files.

<name1> is a variable of type integer that will be sent to the numeric location in memory of <name2>.

An example of the use of based data type is placing a character array on top of a string.

```
VAR X=STRING
DIM BASE CHAR CHARACTERS(80)
LOCATION VAR ADDRESS=X
LOCATE CHARACTERS AT ADDRESS
```

Note: Array index 0 is the <length> of the string.

Or the system's command line:

```
BASE X=STRING
BASE X AT 80H
```

In the above examples, variables are placed into memory either on top of existing variables or into default locations (80H). Care should be taken not to place a variable into a program.

The system load map which follows shows where based data types can be placed in memory.

SYSTEM LOAD MAP

Hex Address	Contents
0-FF	System defined
100	JMP <program>
103	Error code
105-108	System
109	Address of end of compiler code
10B	Common data or program
to ^109	End of compiled code
???	Beginning of BDOS

Location 109H has several uses.

On initial load, this word points to the end of the compiler-generated code.

This location is also used as a stack pointer (similar to SP of the 8080) for use by procedures and functions.

This stack builds to the 8080 stack. Consequently, NO memory in the TPA (transient program area) is safe.

But, if location 109H is changed BEFORE the use of a function or procedure, room can be made by increasing its value. For example, to make 1k of "safe" memory:

```
VAR         START.MEM,STOP.MEM=INTEGER
BASED       MEM=INTEGER
BASE        MEM AT 109H
START.MEM = MEM
START = MEM+1024
STOP.MEM = MEM
```

Future versions of the compiler will use location 109H for allocation of array space and file buffer space. To remain compatible, this location should not be changed until after all DIM and FILES statements have been executed.

Also see the section on files to find where based data types can be placed in memory.

Block Structures

A block structure is a group of <statements> that is treated as one logical statement. An example of this is the use of the BEGIN...END construction.

Remember that, when <statement> is called for, the BEGIN...END structure may be substituted.

Within this block, variables may be declared.

Only variables generated with the VAR statement may be used.

These variables are considered local to the block. By "local" we mean that the <name> given to the variable is only valid within the block structure.

The compiler allocates storage for local variables within the block. At the end of the block structure, the same data storage areas will be used for other variables that might be declared later in the program.

The <name> is discarded from the compiler's symbol table, so that the <name> may also be used in another area of the program.

Be sure to close block structures. Otherwise, at the end of a compilation, you may get a message something like: UNDEFINED LINE NUMBER(S), followed by some garbage. As the compiler generates code, it creates some internal symbols. If these symbols cannot be resolved later on, they are treated as bad line numbers. For example:

```
IF BOOL THEN BEGIN
  PRINT "THE VALUE IS ";BOOL
X = 5*Y
  and so on with the rest of the program...
```

This is incorrect, because there is no END to match the BEGIN statement.

Global and Local Variables

This leads us to the concept of global and local variables.

Variables declared outside of a local block structure are global, i.e., can be used by the entire program.

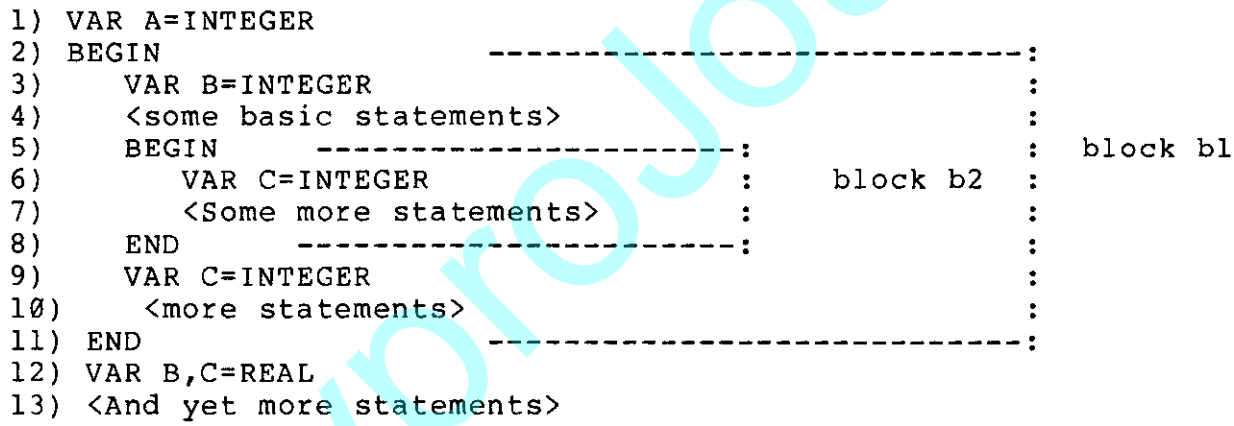
Variables declared inside a block (BEGIN...END and later functions and procedures) are local to that block and cannot be accessed outside the block.

Block Structures using Variables

Block structures can be nested to any depth, limited by memory.

Variables declared outside a block are considered global to the block. Variables declared inside a block are local to the block.

Some examples:



In line 1, variable A is created.

Line 2 is the beginning of a block structure that goes from line 2 to line 11.

In line 3, a variable called B is created. It is local to block b1.

In line 5, a second block, "nested" inside the first, is generated. In this block, the variable B created on line 3 is considered global to block b2. Both A and B are global to the block.

In line 6 and within block b2, a third variable is made.

At line 8, b2 is ended and, as far as the program is concerned, the variable C ceases to exist.

So, in line 9, a NEW variable C can be created. This C is local to b1 and IS NOT the same C used in block b2.

In line 11, the structure b1 is ended, and both variables (B and C), local to b1, are dismissed. Once again, their <name>s may be used for other purposes.

In line 12, this is done.

Once local block structures are declared, then the use of the statements DIM and COM is forbidden.

EXPRESSIONS

Determining the Type of an Expression

An <expression> has a property known as type.

The type of an expression refers to the data type of the variables in the expression. For example,

an expression could be of type REAL
or
if we are dealing with a person's name, it could be of type STRING.

How is the type of an expression determined? There are a few simple rules.

* If the <expression> is to the right of an assignment in a LET statement, its type is the same as the <name> (name=expression)

* If, in this manual, during the description of a statement or function, the type of an expression is given, then simply, that is its type.

* If an expression is by itself, for example, in an IF...THEN or PRINT or UNTIL statement, then the type of the expression is set by the first operand encountered.

* If the first operand is a variable, i.e., <name>, then the type of the expression is set to the type of <name>.

* If the first operand encountered is a constant, then:

If it is a:	then it is a type:
single quote	char
double quote	string
number	real

Variables and constants of a type different than the expression type are allowed in an expression.

In a MIXED type expression, variables and constants are converted to the expression type by the following rules:

1) String to char and char to string are done directly.
2) Char to integer and integer to char are done directly.
3) Fixed to integer are done directly.
4) Given typel to be converted into type2:
    a) Convert typel into real.double.
    b) Convert real.double into type 2.

Type conversions as defined in 4) above, consume much time.

It is best not to mix expression types unless necessary. Conversion errors, such as converting too large a number into an integer, for example, could cause problems. However, conversion by 1, 2, or 3 AND converting real and real.double to integer, and vice versa, need not be avoided.

In general, the conversions are provided to facilitate the use of arrays and functions. Some examples of expressions and their type:

```
VAR X,Y=REAL
VAR A=FIXED
VAR I=INTEGER
VAR S=STRING
X=5+Y          type is real, no conversions
A=X+.5         type is fixed X, convert to fixed
PRINT X        type is real, no conversions
PRINT 5+A      type is real.double, convert A to real.double
X=X=Y          type is real, no conversions
W=S+5          type is real, convert string to real
```

In the statement, IF I=A THEN PRINT, the expression type is INTEGER. Therefore, the variable A will be converted into an integer.

When the compiler encounters one of the following logical operators: NOT, AND, OR, XOR, IMP, AND EQV, it clears the expression type to undefined, and the following operators and argument types are defined, governed by the rules given previously. For example:

```
VAR R=REAL
VAR I=INTEGER
VAR S=STRING
VAR BOOL=INTEGER
```

BOOL= I>5 OR S="STRING"

The assignment above sets the expression type to INTEGER (the type of BOOL).

When the OR is encountered, the expression type is cleared.

When S is encountered, the expression type is set to STRING.

When the expression is finally evaluated, the result is converted to the appropriate type--in this case, INTEGER, the type of BOOL.

BOOL= R=5 AND S>="ABCDEFG"

BOOL sets the expression type to INTEGER.

When R is encountered, it is converted into an integer.

AND clears the expression type, and evaluation continues as in the above example.

If the first constant in an expression is a constant number, explore:

```
25 AND MASK
```

This will NOT do a bit-by-bit AND of 25 and MASK. 25 will be considered and processed by the compiler as a real. MASK, if it is not a real, will be converted into a real. A logical AND is done of the real 25 and MASK, producing a logical result.

The simplest expressions involve constants or variables. The statement below is used to assign a value to a variable:

```
[LET] <name>=<expression>
```

In its simplest form:

```
A=10
NAME.FIRST="Tom"
```

Or, using a variable:

```
A=B
```

String--Special Form

In addition to the LET statement, there is a special form for use with strings:

MID/MID$(<name>,<expl>,<exp2>)=<exp3>

<name> is a string variable, and part of it is replaced by <exp3>, a string expression. Put another way: <exp3> is inserted into <name>.

<exp2> and <exp3> are integer expressions.

Replacement starts with the character position given by <expl> and continues until either the end of <exp3> is reached or <exp2> characters have been transferred.

If <expl> is greater than the current length of <name>, then <name> will remain unchanged.

If <expl>=<exp2>, or length of <exp3> is greater than the maximum length of <name>, then the larger string will be created and truncated to the length of <name>.

FUNCTIONS

Functions can also be used in the <expression> as follows:

A=A+B*C

In this function, operators and operands are mixed to get the desired result from <expression>.

The order of execution of the operations is determined by their precedence.

TABLE OF PRECEDENCE

Those functions on the same line have equal precedence and are evaluated from left to right through the function.

1) subexpressions enclosed in parentheses
2) ^ or ** exponentiation
3) +, - unary negation and absolute value, i.e., +5=ABS(5)
4) *, / multiplication and division
5) +, - addition and subtraction
6) relational: < > or # and =, >, <, <=, >=
7) logical operators: NOT, AND, OR, XOR, IMP, and EQV

Defining Logical Functions

Logical functions are defined, using true/false states for the different variable types.

For types real.double, real, and fixed,

a value of zero is false;
a value not equal to zero is true.

If logical functions are used to return a value, the values are:

-1 for true
Ø is returned for false

With type integer, the same basic rules apply, but logical operations are done bit-wise instead of considering the number as a whole.

For integer, the test for true/false is done for the number as a whole.

When using integers, use the value of
-1 for true and
Ø for false.

String type and char type are the same except that, when considering a string, only the first character is looked at.

Simply, if the letter is of the set, (T,t,Y,y),
then the string or char is considered to be true,
else it is false.

Remember, integer is done bit-wise. For all other types , the variable as a whole is considered.

TRUTH TABLE FOR LOGICAL FUNCTIONS

Operator	Example	Meaning
NOT	NOT Q	The logical negative of Q. If Q is true, NOT Q is false.
AND	Q AND X	The logical product of Q and X. The function is true if, and only if, Q and X are true.
OR	Q OR X	The logical sum of Q and X. Function is false if, and only if, Q and X are both false.
XOR	Q XOR X	The logical exlusive OR of Q and X. Function is true if either Q or X is true, but false if both are true or both are false.
IMP	Q IMP X	The logical implication of Q and X. Function is false if, and only if, Q is true and X is false.
EQV	Q EQV X	Equivalent function. Has the value true if Q and X are both true or both false.

It is possible for a logical expression to be broken into several subexpressions.

TRUTH TABLE SUMMARY

T stands for true, and F stands for false.

Q	X	Q AND X	Q OR X	Q XOR X	Q EQV X	Q IMP X	NOT Q
T	T	T	T	F	T	T	
T	F	F	T	T	T	F	
F	T	F	T	T	F	T	
F	F	F	F	F	T	T	
T							F
F							T

Relational Symbols

Relational symbols test the relationship between two arguments. They return a value of true or false, depending upon whether the relation is true. The relational symbols are as follows:

Symbol	Example	Meaning
=	Q = X	Q is equal to X
<	Q < X	Q is less than X
>	Q > X	Q is greater than X
<>,#	Q <> X	Q is not equal to X
<=,=<	Q <= X	Q is less than or equal to X
>=.=>	Q >= X	Q is greater than or equal to X

CONTROL STATEMENTS

In this section, we will be concerned with the control and transfer of control in a program.

In Structured BASIC, normal statement execution continues from one source line to the next. This flow of execution can be controlled by the programmer, using several statements that either:

direct control to be passed to another line
or
direct a block or group of statements to be executed zero or more times.

The GOTO Statement

The simplest example of the transfer of control is the GOTO statement, using GOTO or GO TO. The following statement transfers control to the line labeled with <line number>.

GOTO <line number> / GO TO <line number>

Example: GOTO 57
GO TO ØALPHA

You may want to review the information on line numbers in the section on Notation.

The GOSUB and RETURN Statements

The GOSUB (GO SUB) statement:

GOSUB <line number> / GO SUB <line number>

passes control to a subroutine. The subroutine begins with the line number referred to in the GOSUB statement where there are one or more statements to be executed. It ends with a RETURN statement which returns control to the line following the GOSUB statement.

When the GOSUB is executed, control is transferred to <line number>. But, unlike a GOTO, it is possible to "return" to the statement that directly follows the GOSUB.

Example:

```
    GOSUB 100
    PRINT A
    .
    .
    .


100     REM Arithmetic Mod Z Subroutine
        A=(X+Y)
        A=A-(A/Z)+Z
        RETURN
```

In the above example, control is transferred to line number 100, where some statements are executed (the subroutine). Then, when the RETURN statement is encountered, control is returned to the statement right after the GOSUB that called the subroutine (such as the PRINT A statement in the example above).

The computed GOTO and GOSUB are as follows:

```
    ON x GOTO <line number>{,<line number>}
    ON x GOSUB <line number>{,<line number>}
```

"x" is an integer expression that should evaluate to a value from 1 to the number of <line number>s in the statement. Should x be out of range, then the next statement will be executed.

The CASE statement and the FOR...NEXT statement described later in this chapter place information on the run-time stack. This allows an optimization for speed and efficiency that would not be possible otherwise.

A program should not GOTO or GOSUB out of a FOR NEXT or CASE structure. Doing this will leave information on the run-time stack, thus accumulating garbage on the stack. Additionally, if a RETURN is executed within such a structure, the return address may be lost.

However, due to the advanced control structures of S-BASIC, it is a simple matter to avoid this. Please see the section on FOR ... NEXT for a general example of this.

The ERROR Statement

There are two types of errors:

Nonfatal

An example of a nonfatal error is when an error is made while entering a number in response to an INPUT statement. Nonfatal errors will only give a warning.

Nonfatal errors will not be trapped by the ON ERROR statement.

Fatal

An example of a fatal error is division by zero.

If the error is fatal, then an error message is printed, and control is passed to the operating system.

ON ERROR OFF/GOTO <line number>

With this statement:

the fatal error message is not printed,
and control is transferred to <line number>
or, if OFF is used, then normal error processing is resumed.

Both nonfatal and fatal errors generate a message and place a code in a location in memory.

Once an ON ERROR statement is executed, it is active throughout the program until another ON ERROR statement changes the <line number> destination of an error.

The run-time stack pointer is reset. This causes the loss of any data on the stack, i. e., FOR NEXT loop control information, returns for a GOSUB, procedures, functions, and any other statement that uses the run-time stack.

The ON ERROR GOTO statement resets the stack. It should not be executed in statements that use the stack. For example, do not place this statement within a FOR NEXT loop. Its execution, even staying within the loop, will corrupt the loop control information on the stack.

ERROR CODES AND MESSAGES (RUN-TIME)

When the following messages are printed on the console, they are followed by: ERROR

If line numbers are known, then 'IN LINE xxxx' is also added.

When the error is not fatal, then 'WARNING ONLY' is appended.

Code	Message
01	LOG <=0
02	CHAIN/EXECUTE OPEN
03	S-TYPE FILE NOT FOUND
04	R-TYPE FILE NOT FOUND
05	S-TYPE FILE CLOSE
06	R-TYPE READ
07	EXTENDING FILE
08	END OF DISK DATA
09	RANDOM RECORD
10	R-TYPE FILE NOT OPEN
11	NO MORE DIR SPACE
12	READ/WRITE PAST EOR
13	S-FILE WRITE
14	WRITE ON UNOPENED FILE
15	READ PAST EOF
16	READ ON UNOPENED FILE
17	<F> DIVISION BY ZERO
18	OVERFLOW/UNDERFLOW
19	OUT OF STRING DATA
20	OUT OF NUMERIC DATA
21	<RS> DIVISION BY ZERO
22	<RD> DIVISION BY ZERO
23	SUBSCRIPT OUT OF BOUNDS
24*	STRING INPUT
28*	NUMBER TOO LARGE/SMALL
29*	INSUFFICIENT INPUT
31*	TOO MANY CHARACTERS. MAX IS 255
33	*** OUT OF MEMORY ***
37	BAD CHANNEL NUMBER
38	INPUT FILE READ

The code is placed in memory. This location in memory is 103H. It can be accessed using the BASED data location type.

```
BASED ERROR.CODE=INTEGER
BASE ERROR.CODE AT 103H
```

Now, when an error is encountered, the coded value (each error has its own value) can be accessed, using ERROR.CODE.

This value will only be changed by the run-time if another error occurs.

This could be used when processing an ON ERROR statement
or after an INPUT statement
to see if the user had trouble when typing in the response.

The REPEAT UNTIL Statement

Please review the section on expressions.

Remember, when <statement> is called for, the block structure, BEGIN...END, may be used.

    REPEAT <statement> UNTIL <expression>

<statement> is repeated until <expression> is true. For example:

```
REPEAT
      BEGIN
            PRINT "STILL LESS THAN"
            X=X+1
      END
 UNTIL X>6
```

Note that the REPEAT statement may be broken up by a carriage return line feed sequence onto several lines. The point where this is permitted is after the REPEAT and just before the UNTIL.

In a REPEAT statement, the <statement> will be done at least once.

You may RETURN or GOTO out of this statement.

The WHILE DO Statement

Another statement for controlling the execution of a <statement> is:

    WHILE <expression> DO <statement>

The <expression> is checked for truth.
If it is true, then <statement> is done.
Therefore, <statement> may never be executed.

This statement may be broken up in the same manner as the REPEAT...UNTIL after the DO.

You may RETURN or GOTO out of this statement.

```
WHILE A>B OR C<D DO
BEGIN
     <some statements>
END
```

Use of the REPEAT statements guarantees that <BODY> will be done at least once. If this is not necessary, a WHILE loop could be used.

The CASE Statement

To select a single <statement> or block structure for execution from among many.

```
CASE <expression> OF
     <expr1>:<statement>
     <expr2>:<statement>
            .
            .
            .
     <exprn>:<statement>
END
```

In this statement, <expression> is evaluated and compared for equality with <expr1>.

If they are equal,
then the <statement> with <expr1> is executed, and
control is passed over the rest of the CASE statement,
past the END,
and to the rest of the program.

If they are not equal,
then the :<statement> is passed over
and <expr2> is checked for equality.

If this test is good,
then <statement> is executed,
and control is passed out of the CASE structure.

This continues until a match with one of the <exprn> is found or the END statement is reached.

<expr1>, <expr2>, <exprn> are all expressions of type equal to <expression>. For example:

```
          CASE RESPONSE OF
"YES":     BEGIN
               <the yes branch>
           END
"NO":     <the no statement>
          END
```

You cannot RETURN or GOTO from within a case statement to outside a case statement.

An otherwise clause can be made by setting <exprn>=<expression>

The IF...THEN Statement

Another form of conditional testing and branching

```
IF <expression> THEN <statement 1> [ELSE <statement 2>]
```

If <expression> is true, then <statement 1> is done.

If the ELSE clause is present, then <statement 2> is done if <expression> is false.

The statement may be broken up onto several lines after the THEN and before and after the ELSE. For example:

```
IF A=B AND C>D THEN PRINT "TRUE" ELSE PRINT "FALSE"

IF RESPONSE THEN RETURN
     ELSE PRINT "NOT DONE"

IF Q<>B THEN
     BEGIN
          <some statements>
     END
ELSE
     BEGIN
          <some more statements>
     END
```

A short form of the IF...THEN uses a line number to replace <statement1>. This is the same as if <statement1>= GOTO <line number>. Example:

```
IF RESPONSE THEN 2037
```

Please note that, in the IF statement, it is "ok" to GOTO or RETURN out of a block.

The IF statement is sensitive to blank lines. The following will not process correctly:

```
IF <exp> THEN <statement>
  ELSE

<statement>
```

No else clause, a null statement (a blank line) is a statement to the compiler.

The FOR...NEXT Statement

To re-execute a group of statements a fixed number of times with a specific increment value.

The form of the statement is:

```
FOR <name>=<exp1> TO <exp2> [STEP <exp3>]
     <body>
NEXT [<name>]
```

In this statement <name> is an index variable that is used as the counter in the FOR...NEXT loop. Its type determines the type of the three expressions <exp1>, <exp2>, and <exp3>.

<exp1> is the starting value of the index or loop counter <name>.

<exp2> is the terminal value. That is to say, <name> shall not exceed <exp2> during the execution of the loop.

If STEP is given, it determines the value that is added to <name> for each repetition of the loop.
If this value is positive, then the loop counts up.

If the value is negative, then the loop counts down by step value.

When STEP is omitted, a default value of +1 is used.

<body> refers to a grouping of statements not unlike the grouping found in a BEGIN...END block. There may be any number of statements including more FOR...NEXT statements (nesting).

The key word, NEXT, functions like the key word, END, in that it forms a frame around the statements that are to be repeated or looped.

With the NEXT statement, the <name> need not be specified. The NEXT always refers to the next nest or loop. iname> can be

given, if the loop is long, and the programmer wishes to help indicate which FOR he is referring to. A check is not made to insure that <name> refers to its matching FOR. <name> is treated like a comment by the compiler.

It should be noted that <exp2> is evaluated once when the loop control is set up.

Some examples:

```
VAR LETTER=CHAR
VAR INDEX=INTEGER
VAR X.STEP=REAL

1)  FOR LETTER=FIRST.LETTER TO LAST.LETTER
      <body>
    NEXT

2)  FOR INDEX=1 TO 10
      PRINT INDEX
    NEXT INDEX

3)  FOR X.STEP=1 TO 10 STEP .01

4)  PRINT X.STEP
    NEXT

5)  FOR LETTER='A' TO 'Z'
      <body>
    NEXT LETTER
```

In the FOR...NEXT loop #1 above:

the variable, LETTER, takes on the value of FIRST.LETTER,

<body> is executed,

and LETTER is moved on to the next letter in the ASCII set until LAST.LETTER is reached.

In loop #2, a value called INDEX is stepped from 1 through 10, i.e., the loop is executed 10 times. With this statement, the numbers 1 through 10 would be printed on the console device.

In statement #3, a variable called X.STEP is started with a value of 1 and indexed to 10 by a step value of .01. This means that the loop will be executed 1000 times (10/.01).

For an explanation of statement #4, see the section on procedures.

In the FOR...NEXT statement starting on statement #5 above:

the variable, LETTER. is started at the value of 'A',

<body> is executed,

and the NEXT statement is reached.

LETTER is moved on to the next letter in the ASCII set until Z is reached.

<body> is executed then for each letter from A to Z with LETTER being set to the value of the letter during the looping.

In this statement:

```
FOR X.STEP=10 TO 1 STEP -.01
 PRINT X.STEP
NEXT
```

the function of this statement is to step through the loop 1000 times by a step value of .01. The main difference between this statement and the one above is that the counting is done backwards. The value X.STEP starts out at 10 and counts down to 1 by increments of .01.

Remember not to RETURN or GOTO out of a FOR...NEXT statement. If that type of construction is necessary (for example, when translating from another BASIC), it is possible to substitute a general construction. Thus:

```
FOR X = EXP1 TO EXP2 STEP EXP3
          <BODY>
NEXT
```

Becomes:

```
X=EXP1
REPEAT
BEGIN
   <BODY>
   X=X+EXP3
END
UNTIL X>EXP2
X=X-EXP3
```

Of course, much simpler constructions are possible for specific examples. The above is general for the purpose of demonstrating the technique.

CHAIN AND EXECUTE Statements

Often, it is desirable to load a BASIC program from disk under control of another BASIC program.

It is also desirable to preserve arrays and variables during the loading and executing of the other BASIC programs and maintain the symbolic reference in the program.

Such arrays and variables are created with the COMMON statement and the DIM COM... statement.

During the loading and executing (chaining) of BASIC programs, these variables remain intact with their run-time values unchanged.

(It should be noted that CHAIN will only work with BASIC programs produced by the S-BASIC compiler.)

COMMON variables need not be declared for two programs to chain each other in and out of memory.

When chaining, the number and type of COMMON variables and/or arrays must match exactly, i.e., they each must have the same number of COM and DIM COM statements, and be in the same order as each other. If they are not be in order, then a CHAIN error will occur. This is a fatal error.

CHAIN Statement

A BASIC program can load and execute another BASIC program using the following statement:

```
CHAIN <file name>
```

Where <file name> is the name of the file in a valid operating system format. <file name> may be an expression, and its type is string. For example:

```
CHAIN "LDF.COM"
CHAIN LOAD.FILE
CHAIN COMMAND.FILE+".COM"
```

Where the string variable, LOAD.FILE, holds the file name to be loaded and executed.

A program may chain back to the program that invoked it or to another BASIC program. CHAIN always starts execution of the program at the beginning.

When the end of program is reached, or a STOP statement is found, the chained program returns control to the operating system.

EXECUTE STATEMENT

To load and execute any .COM file and then regain control of the system from a BASIC program, the following statement may be used:

```
EXECUTE <expression1>[,<expression2>]
```

In this statement, <expression1> is of type string and should evaluate to an operating system file name with an extent of .COM.

If <expression2> is given, it is of type string and should evaluate to a valid operating system command line. This command line is executed AFTER the file specified by <expression1> is loaded, executed, and has returned control back to the operating system.

Even if the file specified by <expression1> is not found or an error is generated, the command line given by <expression2> is still executed so that control can be returned to your program in the event of an error. For example:

```
EXECUTE FILE.NAME,COMMAND.STRING
EXECUTE "A:SORT.COM"
EXECUTE "A:SORT.COM","B:MENU"
```

In the above examples, FILE.NAME and COMMAND.STRING are string variables.

The file specified by FILE.NAME will be loaded and executed.

Upon completion, the COMMAND.STRING will be executed as a system command.

The command string given by <expression2> is controlled through the SUBMIT facility of the operating system.

As such, it is restricted in the same way, i.e., the statement should be executed while the system is logged in on drive A.

For further information, see your operating system manuals.

EXECUTE does not preserve the common data structures.

CHAIN and EXECUTE look for the file as given in the argument. For example, under CP/M, if you do not append a .COM to the file name, it will try to load a file with a blank extent. This can be useful in preventing program modules from being run out of sequence. Also, CHAIN and EXECUTE allow lower case file names.

I/O Devices

Data quantities are entered into a variable at run-time, using the INPUT statement.

Data quantities are printed on a physical device, such as a CRT screen or printer, using the PRINT statement and the TEXT statement.

Below is a table of the physical devices connected to each I/O channel.

Channel #	Input	Output
Ø	Console	Console
1	Dummy	List
2	Dummy	Punch
3	Reader	Dummy
4	Console status	Dummy
5	Keyin	Keyout (Dummy)

Above, where Dummy is specified, it refers to a nonexistent device.

Input from the input devices is terminated with a RETURN or ESCAPE or having the carry bit set in the program status word.

If an ESCAPE is used to terminate a line, it will be stored as part of the line.

The device key-in will return the character typed if a key is down; otherwise, it will return a Ø, with the carry bit set in the PSW. The effect of this is that an input can be done to see if a key has been pressed.

Data quantities are entered into the run-time package produced by the compiler via these channels. The BASIC statement that directs this activity is the INPUT statement.

Designating an I/O Source

INPUT [#<exp>;]["<prompt>",/;]<name>{,<name>}[,/;]

Where <exp>, if given, is an integer expression designating one of the I/O channels listed above as the source of input data.

If <exp> is not given, then a default device of Ø will be used; this is the console.

The <prompt>, if included in the statement, will precede the input statement's request for data. The input statement requests data from the user with a ? prompt (see below for exception).

The <prompt> must be followed with either a comma (,) or a semicolon (;). If the semicolon is used, then the input request will be placed in the character position directly after the <prompt>. If the comma is used, then the input request will be placed in the next tab position.

Tab positions are set at every 14 character positions.

At least one <name> must be given.

The information entered from the device given by <exp> is converted into internal format according to the <name>'s type and stored at <name>.

<name> may then be followed by a comma and more <name>s to be entered.

This list of <name>s may be ended in one of three different ways.

* With a return-line-feed, which will continue to the next line of the device.

* With a comma, which will tab to the next tab setting.

* With a semicolon, which will do nothing (stay put).

Generally, the S-BASIC run-time prompts the user with a question mark, accepts the input, and terminates this with a return and a line-feed to the next line on the device. Both the question mark generation and the moving to the next line can be controlled by the INPUT statement. The key word, INPUT, is substituted with one of the below to give the desired effect:

	Generates a question mark prompt	Generates a return to the next line
INPUT	Yes	Yes
INPUT1	Yes	No
INPUT2	No	Yes
INPUT3	No	No

When more than one data item is to be entered (i.e., more than one <name> in the input list), then the items are separated with commas.

If the items being entered are strings which include commas, then they must be enclosed in double quotes.

Single characters must be separated with commas.

There are two special cases of the above:

1. If only one <name> is given to the input list, and its type is string, then everything the user types is entered into the string. No commas or quotes are needed, and, if typed, will be entered into the string.

2. If only one <name> is given as above, and its type is char, then only one key press will be required; an ESC or RETURN will not be necessary.

Some examples of the use of the INPUT statement:

```
VAR A=REAL
INPUT A
```

This will enter a number in real format and store it in A.

```
VAR B=REAL
INPUT A,B
```

Enter two reals separated by commas, and store at A and B.

```
VAR LETTER=CHAR
VAR READER=INTEGER
READER=3
INPUT3 #READER; LETTER
```

Get a byte from the reader device.

```
VAR CHECK.AMOUNT=FIXED
INPUT2 #0; "Enter amount of check $";CHECK.AMOUNT
```

Enter a dollar amount, and store it at CHECK.AMOUNT. Channel #0 is the default and need not be specified. It is done here for an example.

When variables are entered, using the INPUT statement, they are first entered into a buffer, then converted into the run-time format. While the text is being entered into the buffer, several editing functions can be used to make corrections before conversion. They are:

1. Control-U - to delete the current line typed in and begin anew.

2. Control-R - the computer prints what has been entered into the buffer.

3. If a terminal that will use ASCII BS is being used for input, an ASCII BS may be typed to delete the last character typed.

4. If the console does not use ASCII BS, then an ASCII DEL (rub out) may be used to delete the last character typed.

Echo

As each character of input is typed by the user, it is 'echoed'. This action can be turned on and off by using the statement:

```
ECHO ON/OFF
```

The INPUT statement generally will return to the operating system if a control--c is typed during type-in.
This feature can be turned on and off with the following statement:

```
CONTROL.C.TRAP ON/OFF
```

It is initially on.

Input Only

A special "input only" channel is provided for use when an SBASIC run-time file is loaded with a second file name. This channel is #9 and can be used to "read" the file specified in the command line. This is the file which is set up in the default FCBN at 5CH. The input only status and the way the file is used can be changed by entering changes into the I/O routine, "USERLIB".

```
REM DUMP TEXT FILE TO CONSOLE
VAR LETTER=CHAR
LETTER=ØDH
WHILE LETTER <>1AH DO BEGIN
   PRINT LETTER;
   INPUT #9;LETTER
END
```

To run this program, assume a name of DUMP.

```
A>  DUMP FILE
```

Output (Printing)

Data is sent to a device (printed), using the PRINT statement.

```
PRINT [#<exp>;][<expl>{,/;<expl>}[,/;]]
```

<exp>, if given, refers to an I/O channel from above; it is an integer expression.

If not given, then output is sent to the console (device #Ø). <expl> is an expression of any type. See section on Determining the Type of Expression.

This expression may be followed by a comma (,) or a semicolon (;).

A comma will cause the print position to be moved to the next tab stop.

A semicolon will cause the print position to advance to the next print position.

A PRINT statement without an expression list will generate a blank line, or if some commas are in the statement, will cause the print head to move to the next print tab stop.

Some examples:

```
VAR LIST=INTEGER
LIST=1
PRINT #LIST;A,B;
```

Will print reals A and B on the list device supressing the return to the next line.

```
PRINT #LIST;A,B,
```

Same as above, except moves to next tab stop after printing B.

```
PRINT #LIST
```

Generates a blank line on the list device.

Changing the Destination of the Output

A use for #<exp> is to change the destination of the PRINT statements output at run-time.

```
LIST      = 1
CONSOLE   = Ø
INPUT "HARDCOPY";RESPONSE
IF RESPONSE THEN DEVICE=LIST ELSE DEVICE=CONSOLE
PRINT #DEVICE; JUNK
```

If a device is not given (i.e., <exp> above), then the default print device is the console (device #Ø).

I/O statements set up the I/O channel once at the beginning of the statement to conserve code generation. For example, the PRINT statement sets up the channel to be used for output once--just before the first expression is evaluated. Thus, if one of the expressions to be printed is a call on a user-defined function that changes the I/O channel, from that point on, the PRINT statement will send its output to the wrong channel.

If the following statements are used, the default destination can be changed. The statement:

```
LPRINTER [<integer const>]
```

will cause the default print device to be changed to #1 if <integer const> is not present.

If <integer const> is present, then that will become the default print device, and will be used from then on when LPRINTER is encountered.

The statement:

```
CONSOLE [<integer const>]
```

can be used to change back to the console as the default print device.

<integer const> can be given to change the default setting of #Ø.

LPRINTER and CONSOLE are compile-time statements and have no effect at run-time.

In the statement:

```
IF A=B THEN LPRINTER
```

LPRINTER will "execute" regardless of the logical state of the IF expression A=B.

LPRINTER is a compile time statement.

You may change the I/O channel used at run-time by using the #<exp>; form of the PRINT statement. See above, starting "A use for #<exp>".

The TEXT Statement

When large blocks of text need to be sent to a device (for example, when giving instructions on how to run a program), the TEXT statement can be used to compose the text just as it will be printed without the bothersome PRINT statement. The form of this statement is:

```
TEXT <channel>,<delm><text><delm>[,/;]
```

In this statement, <channel> is an integer constant for the compiler that specifies where the text is to go.

<delm> can be any printing character and is used to frame <text>.

<text> can be any number of lines and any characters, not including <delm>.

The second <delm> is to be the same character as the first <delm> and tells the compiler that the end of the output text has been reached.

The comma and the semicolon, or lack of, function just like the print statement ending.

The following is an example of the TEXT statement.

```
TEXT Ø,&
```

A menu might look like this:

1. Do something
2. Do something else
3. Do nothing
4. I do not want to respond

And still more is possible. The need for a bunch of print statements is gone. Just one text statement and easy formatting is at your command. For, with the TEXT statement, you enter the text just as it will be displayed. No need to cope with the offset caused by the PRINT statement. And it's fast.

&

The character "&" is used as the <delm> and frames the <text> of the TEXT statement in this example. This example sends the <text> to the console.

The PRINT USING Statement

A PRINT statement generally outputs numbers and strings in a free format that is left justified. The exception to this is when printing fixed point numbers. Often, it is desirable to employ a specific format for the printout.

The PRINT USING statement is provided for this purpose.

The statement contains a format specification that controls the area in which printing takes place.

A format string is composed of two basic fields:

* a field to control the printing of numbers

* a field to control the printing of strings

Both fields need not be specified, if numbers only or strings only are to be printed by a statement.

Between these fields are literal data that is printed as found in the format string.

The form for printing, using a format string, is:

```
PRINT USING <string exp>;<print list>
```

With a general form for the PRINT statement:

```
PRINT [USING <string exp>;][#<integer exp>;][<print list>]
```

The format string is <string exp>.

This format string is scanned from left to right to seek a format field.

The type of field, string or numeric, is determined by the print list.

Characters that are found before encountering the desired format field are printed (literal data).

Therefore, the <string exp> is an image of the line to be printed, with the exception of the formatting characters, or field. This will be further explained after a discussion of the format fields.

String Fields

String fields control the printing of a string from the print list.

A one-character field, a fixed-length field, or a variable-length field can be specified.

!

An exclamation mark is a one-character format field. It specifies that the string from the print list is to be printed in one character position.

If the string is empty, then a space is printed.

If there is more than one character in the string, only the first character is printed.

```
PRINT USING "!. "; "John","Doe"
```

would output:

```
J. D.
```

The period and space in the format string are treated as literal data; they are not part of the format field.

When the end of a format string is reached, it is re-used.

/
The slash mark is used to format a fixed-length field, when printing a string.

It is specified by a pair of slash marks separated by Ø or more characters.

Both slashes count for a character position.

Thus /.../ would reserve 5 spaces for a string to be fitted into, when printed.

If the string is less than the length specified, it is filled with spaces on the right.

If longer, then the print string is cut off.

```
PRINT USING "/.../"; "JO","GILBERT"
```

would output:

```
JO   GILBE
```

&
The ampersand is a one-character format field.

It specifies that the string is to be printed as is, or in a variable-length field.

Numeric Fields

Numeric fields are used to control the printing of numeric data.

It is generally not necessary to format fixed point numbers, as they already print in a format style.

However, by using formatted print, it is possible to "see" the last digit which is normally not printed, due to the rounding of fixed point numbers when displayed. (See the section in Data Types starting F SDDDDDDDD.DDD)

In general, the format field specifies how many digits are to be printed before the decimal point and how many are to be printed after the decimal point. Also, floating dollar sign, trailing sign, asterisk fill, exponential format, and others can be specified.

#

The pound sign reserves one digit position for the number to be printed.

There may be one decimal point within these "#" to indicate where the decimal point is to be printed.
Leading zeros are replaced with spaces.

Format	Number	Printout
###	12	12
##.#	12	12.Ø
###.##	12.456	12.46
##	5ØØ	% 5ØØ

As can be seen, the number is rounded to fit within the formatted field.

If the number will not fit, it is printed with a "%" symbol.

,

A comma may be inserted one or more times before the decimal point.

The effect is to cause the printing of a comma every 3 decimal places.

Each comma takes up a digit position.

So, though one comma in the format field will cause comma insertion, it is best to use a comma for each occurrence, so as not to use up "#" digit positions from above.

+

The plus sign may be used at the beginning of a numeric format field or at the end of a numeric field. It indicates that the sign of the number is to be printed.

A plus sign is printed if the number is positive.

A minus sign is printed if the number is negative.

-

The hyphen may be used like the plus sign. The difference is that, if the number is positive, a space is printed.

**

Two asterisks, when placed at the beginning of a numeric field, specify that leading zeros are to be replaced with an asterisk. Normally, leading zeros are replaced with spaces.

$$

A pair of dollar signs placed at the beginning of a numeric field is used to generate a floating dollar sign. That is to say, the dollar sign is floated through leading spaces to the first character position before the number.

**$

The character sequence **$ at the beginning of a numeric field generates both asterisk fill and a floating dollar sign.

The *, which is located in the first character position before the number, is replaced with a $.

Exponential Format

^^^

One or more ^ following the format field will specify that the number is to be printed in exponential format.

Comma insertion is not allowed.

The decimal point may be fixed anywhere within the format field.

The number is left justified, and trailing spaces are filled with zeros.

The exponent is printed in the form "Esdd", where E stands for "times ten to the" and "s" is the sign (-,+). The "dd" stands for the digits of the power function. Thus, the printout 5E+12 would be read as "Five times ten to the 12th power".

Providing for the Escape of Characters from Format Control

When it is necessary to print one of the above characters, whether from the string type or numeric type, the character can be made to escape from format control. For example, to print a # before a number, the # would have to escape being part of the numeric format field.

The procedure is to type a backslash followed by the character to be escaped.

Some examples:

```
0001:                 var f=string
0002:                 var x=real
0003: 0000            input f
0004:                 input x
0005:                 print using f;x
0006:                 goto 0000
0007:                 end
0008:                 ****** End of program ******
```

```
? ######.##
? 12
     12.00
? ###.##
? 5.555
  5.56
? ##
? 5000
% 5000
? ##,###,####.##
? 12345
     12,345.00
? +###.##
? 12
 +12.00
? ###.##+
? 12
 12.00+
? ###.##+
? -12
12.00-
? ###.##-
? 12
 12.00
? ###.##-
? -12
 12.00-
? ********.##
? 12
******12.00
? $$######.##
? 12
       $12.00
? **$#####.##
? 12
*****$12.00
? **$#####.##
? -12
****$-12.00
? ##.##^^^^
? 123.45
12.35E+01
? Literal data ####.##
? 12
Literal data    12.00
? Literal data with escape  \#####.##
? 12
Literal data with escape #   12.00
? Pay **$,###,###.##+
? 123456
Pay ***$123,456.00+
```

```
0001:                 var f,x=string
0002: 0000            input f
0003:                 input x
0004:                 print using f;x
0005:                 goto 0000
0006:                 end
0007:                 ****** End of program ******
```

```
? Single character field !
? Print string
Single character field P
? Fixed length field /.../
? Gilbert
Fixed length field Gilbe
? Any length &
? Any string
Any length Any string
? /.../The end
? k
k    The end
? & after format
? Test
Test after format
? & after format &
? Test
Test after format
```

When printing is finished, what literal data is left in the format string is printed. This can be seen above in the last few examples.

Reading Data

Data need not come from INPUT statement. It can be included in the program and read during run-time. This is done by using one of three BASIC statements.

1. DATA <ASCII data>
   This statement stores ASCII data in the program for use at run-time.

   All data statements must appear together in the program.

   The ASCII data is saved in memory in a compacted form by removing all the spaces from the ascii data.

   The exception to this is data enclosed in double quotes.

   The programmer may include as many spaces and/or tabs as desired to improve readability without taking up room in the run-time package.

Data items in the ASCII data should be separated by commas just like data for an INPUT statement.

All data statements must be together.

Examples:

```
DATA  5,6,-204,line,data,"line with spaces"
DATA a,b,c,d,e,f,g,h,i,j,k,l
```

2. Data is read from the data list generated by the DATA statement, using the READ statement.

   Items are read from the beginning to the end of the list.

   Example:

   READ <name>{,<name>}

   Here <name> is a variable that is to be stored with the current value in the data field.

   The data should make some sense for the type of <name>.

3. As items are read from the data list, a pointer is moved through the list to determine what item is to be READ next. This pointer can be moved to point to the beginning of the data field or to a particular DATA statement.

   BEFORE data can be read, the pointer must be positioned, using the following statement:

   RESTORE [<line number>]

   where <line number> is a line number of a data statement at which the pointer is to be positioned.

   If <line number> is omitted, then the pointer is set to the first data statement in the program.

DISK FILES

This chapter is about the storage and retrieval of data from the disk.

Storage Area of Disk

A disk is divided into three different regions:

* operating system area
* directory with information on the file name, its location on the disk, its length, etc.
* data storage area

Data is written to and read from the data storage area under the control of the operating system, using the information in the directory to keep track of where the data is to go to or come from.

Disk Statement Types

There are three classes of disk statements in S-BASIC:

* One type deals with the setting up of memory buffers for the operating system. This is where data comes from or goes to during a disk transfer.
* Another set of statements manipulates the directory.
* The third group brings this all together to accomplish the final transfer of data to and from the disk.

Access to Data

There are two basic types of files and data transfers:

* Random access
* Serial access

All disk I/O (input/output) is done through file channels.

These channels are another set of channels from those used in the INPUT and PRINT statement. Each channel is defined as to its

type, random or serial.

Files

1. The files statement may use expressions of type integer for the <size> argument.

   FILES R/S (<SIZE EXPRESSION>){ .R/S(<SIZE EXPRESSION>)}

2. The OPEN statement may change the size of the disk buffer by adding an expression of type integer.

   OPEN #<channel>;<expression>,<size expression>

FILES R/S (<SIZE>) {,R/S(<SIZE>)}

where the first channel number is Ø.

In the statement above:

R - defines the file channel as random access.

   For a channel defined as random, <size> is an integer expression which determines the number of bytes per record.

   If <size> is a multiple of 128 bytes, a special optimization is possible that enhances the speed in which data moves to and from the random record on the disk.

S - defines a channel as serial, and <size> will determine the length in sectors of the disk I/O buffer for this channel.

Some examples of the FILE statement:

   FILES R(128),R(4),S(1),S(2),R(917)

   FILES R(RANDOM.BUFFER.SIZE),S(SERIAL.BUFFER.SIZE)

   FILES R(128),R(256),S(1)

In the first statement:

Channel #Ø is a random file with a record length and memory buffer length of 128 bytes.

Channel #1 is a random file with a record length of 4 bytes.

Channel #2 is a serial file with a buffer length of 1 sector.

Channel #3 is a serial file with a buffer length of 2 sectors.

Channel #4 is a random file with a record length of 917 bytes.

The maximum number of channels is 32.

One FILES statement per program is allowed.

The channels defined with this statement may be used more than once for different files, by use of the OPEN and CLOSE statements.

A FILES statement only defines the type of file for a given channel.

CREATE <expression>

Before anything can be done with a file, it must exist in the directory.

The CREATE statement will create an entry in the directory with the name given by the string <expression>. This does not affect the data area of the disk.

Note: If a file with the name given by <expression> is already on the disk, nothing will happen.

DELETE <expression>

The DELETE statement deletes the named file specified by the string expression <expression>. If there is data stored in this file, the data is lost.

RENAME <exp1> TO <exp2>

With this statement, both <exp1> and <exp2> are of type string.

The file specified by <exp1> is renamed to the name given by <exp2>.

If the file <exp1> does not exist, then no renaming takes place, and a file named <exp2> is not created.

In either case, the data in the file is not changed.

INITIALIZE

This statement initializes the operating system for a new disk. It MUST be used whenever a disk is changed.

For further information on these statements, see their counterparts in your operating system manuals.

Attaching and Detaching Files from a File Channel

For disk I/O to take place, a file from the directory must be assigned a file channel from the FILES statement. At the end of the disk I/O, the file must be detached from the channel or damage might occur to the data in the file.

A file is attached to a file channel with the following statement:

```
OPEN #<channel>;<expression>[,<size>]
```

Where <channel> is an integer expression specifying one of the file channels defined with a FILES statement.

<expression> is of type string and should be the same as a valid operating system file name.

<size> is an integer expression specifying the size of the disk I/O buffer and is optional.

The statement enables the file specified by <expression> to be used for input or output operations.

```
OPEN #RANDOM.CHANNEL;ACCOUNT
```

Reading Data from a File

There are two statements used to read data from a disk file:

If the file channel is serial:

```
READ #<channel>;<name>{,<name>}
```

Data is read from the file much like data being read using the READ statement.

The data is read sequentially from the beginning of the file to the end.

The data can be re-read by re-opening the file.

If the file channel is random:

```
READ #<channel>,<exp>;[<name>{,<name>}
```

Read <exp> is an integer expression which is used to read the record from the file into a buffer.

Note: No <name>s need be given for this to occur.

Thus, it is possible to do direct buffer I/O with the file. A common use would be to place based located data types into the buffer. As disk I/O progresses, their values will reflect the changes.

Data can be read from the record into variables much like the READ statement for the sequential file, except that only the data in the current record can be read. This is record sequential access.

Writing Data to a File

Data is written to the file much the same as it is read, except in reverse.
The statements for doing this are:

If the file is serial:

```
WRITE #<channel>;<name>{,<name>}
```

<channel> is the integer expression specifying the file channel to be used.

Data is written to the file one <name> at a time, from the beginning to the end of the file.

If the file is random:

```
WRITE #<channel>,<exp>;[<name>{,<name>}
```

Data is written to the file's record.

<exp> is an integer expression specifying the record number.

If no <name>s are given, then the buffer is written to the file.

Closing the File

Once writing to a file is complete, it must be closed to ensure that all the disk buffers have been cleared and the directory updated with any changes made to the file. This is done with the statement:

```
CLOSE #<channel>
```

Where <channel> is the file channel to be closed.

This disconnects the file from the file channel; this channel may now be used for another file.

THE USE OF RANDOM AND SERIAL FILES

Binary is used to store data on file and in memory.

All data types have a fixed length measured in bytes:

Real	4 bytes
Real.Double	7 bytes
Fixed	6 bytes
Integer	2 bytes
Char	1 byte
String	max.len+1

The type string has a length equal to the maximum length of the string + 1.

When information is stored on the disk file, each data item takes up its count of bytes, no matter what the value of the data item.

For example, the number Ø takes up as much storage as 456.

The Pointer

When a file is opened, a pointer is created for that file channel. The pointer indicates where the next data item will be read from or written to.

For example, if we open a file, the pointer is set to the beginning of that file.

If a real data type is read from the file, then the pointer is set to the fifth byte in the file after the read operation. This is the next location in the file where data will be read from.

Now, if the same file is opened on ANOTHER channel and used for writing, then the real that we read can be changed and written back onto the file right where it was.

Sequential Files

This is handy for updating large sequential files in that, to make changes, the file does not have to be copied into another file. It can simply be written onto itself, since the data field lengths are fixed.

Problems would occur if different data types were written, then read, or string lengths were changed .

For example, to read a number from a sequential file, the following statement would be used:

```
READ #FILE.CHANNEL;VAR.NAME
```

And the next value in the file:

```
READ #FILE.CHANNEL;VAR.NAME2
```

And so on.

Strings

When talking about string lengths, we do not mean the number of characters in the string, but rather the maximum length the string can obtain, since the string is padded to the maximum length.

Updating a File

The technique for updating a large file is to open the file on two different channels, then read from one channel and write to the other.

```
REM UP-DATE A SEQUENTIAL FILE
FILES S(1),S(1)
OPEN #0,"FILENAME"
OPEN #1,"FILENAME"
FOR X=1 TO EOF
     READ #0;VARIABLE.X
     REM SHOW THAT THE VALUE CAN BE CHANGED
     VARIABLE.X=VARIABLE.X+SOMETHING
     WRITE #1;VARIABLE.X
NEXT
CLOSE #1
```

As shown above, when a file is opened, it can be used for read or write mode. However, once an opened file is read from or written to, its mode cannnot be mixed.

Generally, it is not necessary to close a file that has been used for reading.

A file may be opened on a channel where a file is already open, but this action destroys any data in the old file's memory buffer.

S-BASIC will NOT automatically close a file; it must be closed with the CLOSE statement. This helps to prevent needless trips to the directory.

Random Files

A random file is separated into several parts called records. Each record can be thought of as a mini sequential file.

The same type of actions described above are possible for a random file, except that only a single record of the file is considered.

Remember, the size of a random record is defined in the FILES or OPEN statement.

READ and WRITE

Variables are read and written directly from and to a random record as follows:

```
READ #FILE.CHANNEL,RECORD;VARIABLE.1,VARIABLE.2
WRITE #FILE.CHANNEL,RECORD;VARIABLE.1
```

As can be seen above, a random file can be read and written to at the same time. These actions of read and write can be mixed in any order.

This is different from sequential files, as sequential files can only be read from or written to, but not both at the same time on the same channel.

This is possible with random files.

The mode of a random record is: read when reading data, and write for writing data.

A new record is fetched into the buffer or written onto the file when the mode changes or the record number is changed.

Therefore, it is possible, for example, to read some data from a random record in one statement and read some more data in another statement. This is called record sequential.

```
READ #3,R;X1
READ #3,R;X2
```

The above statements would read the first variable in record R and then the NEXT variable in record R. It will not read the same data into X1 and X2.

The record is treated like a mini sequential file.

The same type of action is possible for writing.

To get back to the beginning of the record, a different record could be read, or the same rcord could be fetched with a buffer direct statement (see below).

```
READ #1,R;X
REM CHANGE VALUE OF X
X=X+1
WRITE #1,R;X
READ #1,R;X
```

The above example would read X.

The mode at this point is "read", and the record number R.

Then, with the WRITE statement, mode is CHANGED to "write".

For this reason, X will be written into the record as the first data in that record.

In the following READ, mode is again changed so the buffer is fetched from the file and will reflect the change in the value of X.

When doing random file I/O, it is possible to read and write variables directly from the record. It is also possible to do direct buffered I/O. This is done by declaring variables to be based located.

Remember that a based located variable is not assigned a location by the compiler, but rather its location is assigned at run-time.

Also, remember that the location of a file's buffer can be found, using the LOCATION statement and specifying FILE.

Using these statements, variables can be placed in the disk file buffer.

These buffers can then be read from or written to the random record, using the READ # and WRITE # statements without any <name>s. The effect of this is that, when a READ # statement is executed, that record is fetched from the file and placed in the disk buffer.

If variables have been placed in the buffer, they will be reflect the new contents of the buffer.

Likewise, if data is stored into the buffer with assignment statements and the LET statement and followed by a WRITE #, then they will be written to the random record.

Not only simple variables of any type can be placed in the buffer, but arrays of any type can also be placed into the buffer. It is also permitted to overlay variables on top of one another. See the section on arrays.

```
FILES R(8)
BASED X,Y=INTEGER
BASED Z=REAL
LOCATION FILE ADDRESS=#Ø
BASE X AT ADDRESS
BASE Y AT ADDRESS+2
BASE Z AT ADDRESS+4
```

```
FILES R(128)
DIM BASE CHAR X(127)
LOCATION FILE ADDRESS=#Ø
LOCATE X AT ADDRESS
```

To get a buffer:

```
READ #CHANNEL,RECORD
```

To put a buffer:

```
WRITE #CHANNEL,RECORD
```

In the above examples, based located variables are placed into the disk file buffer.

A READ statement will read a disk buffer (record), and X, Y, and Z would be set to the value of the buffer.

A WRITE statement would write the buffer to the disk, and thus the values of the variables.

When reading and writing a random file using direct-buffered I/O, leave off the semicolon.

```
Wrong:              WRITE #1,X;
CORRECT:            WRITE #1,X
```

The first example uses simple variables. The second uses an array.

Variables and array(s) may be placed in the buffer in just about any way the programmer wishes, including on top of each other.

A possible example would be a character array on top of a string.

When strings are placed into a file buffer, be sure the <length> is not changed inadvertently by a disk read. See the section on the BASED statement.

The method to expand the size of a file is to append records to the end of the file, i.e., the last record +1.

Writing out farther than this will have unknown results.

ASCII FILES AND CHANNEL NUMBERS

S-BASIC has been expanded to allow channel numbers to be expressed as type integer expressions. This includes the statements READ, WRITE, OPEN, and CLOSE.

This section is concerned with the addition of ASCII files to the INPUT and PRINT statements. The expanded syntax for these statements concerns only the channel number. The expanded syntax is:

```
#<channel number>[,<record number>];
```

An example of use:

```
PRINT #CHANNEL, RECORD.NUMBER; A, B, C
```

Also, the syntax of the FILES statement has been expanded as follows:

```
FILES <R/S/RA/SA(<SIZE>)> /D {,<R/S/RA/SA(<SIZE>)> /D}
```

Example:

```
FILES   D, D, S(1), R(213), SA(1), RA(80+2)
```

where RA stands for Random ASCII and SA stands for Sequential ASCII files. D is used as a place marker for Device.

Before the above can be fully understood, the concepts governing disk I/O channels and device I/O channels must be presented.

Normally, disk and device channels are separate. Channel number 0, when used with the statements REAL and WRITE, refers to the disk. Channel number 0, when used with the statements PRINT and INPUT, refers to devices. In other words, disk and device channels are separate, accessed by different statements.

With the addition of ASCII files accessed through the statement INPUT and PRINT, the separateness of the channels is violated. In the above FILES statement, RA and SA are used to set up ASCII DISK channels. When an ASCII file channel is defined, the device channel of the same number is LOST! It is important to remember this. In the above example, FILES statement device channels number 4 and 5 are lost. In their place are now disk channels. INPUT and PRINT, when using channels numbers 4 and 5, will talk to the disk and not to a device. A consequence of this can be examined, using the following example:

```
FILES SA(1), SA(1)
```

In this example, ASCII files are defined for channels number Ø and 1. BUT, these are the channels used for console and print device! Remember, that, if an ASCII channel is defined, the corresponding device channel is lost. This statement destroys our ability to talk to the console and printer! For this reason, a dummy place marker has been defined to set aside channels for device I/O. The dummy marker is 'D'.

```
FILES D, D, SA(1), SA(2)
```

The device marker is treated as a comment by the compiler. It simply reserves a channel as a device channel. Its principle use is as a program-documenting feature. Please note the following example:

```
FILES S(1), S(1), SA(1), SA(1)
```

Here we have defined disk channel numbers Ø and 1. This files statement does NOT cause device channels Ø and 1 to be lost. Device channels are only lost when defining ASCII file channels. We would ENCOURAGE the use of the device marker, D, in the files statement. This is to make clear the use of channels. By using the device marker in the FILES statement, the appearance of merged disk and device channels can be given.

Note: ASCII file channels may NOT be accessed with the READ and WRITE disk statements. Also, direct-buffered I/O is not possible with ASCII files. For example, the following statement is not permitted:

```
PRINT #CHANNEL,RECORD.NUMBER
```

But the following statement is permitted to generate a return and line feed sequence:

```
PRINT #CHANNEL,RECORD.NUMBER;
```

Please note further that, when inputting from an ASCII file, the only input statement that can be used is INPUT3. This is because the disk is not an interactive device. You cannot send it a prompt, get data, then send it a return linefeed combination.

Also, the input prompt must be left out. Failure to abide by this will result in a fatal run-time error. An example of a proper input statement:

```
INPUT3 #C,R: A,B,C
     or
INPUT3 #C; A,B,C
```

When doing random ASCII disk I/O, it is permitted (provided record sequential is enabled) to write more than one logical line into the record. Therefore, a given record may contain more than one logical line. A logical line is all characters up to, but not including, an ASCII return. An ASCII linefeed is assumed after the return. The print statement will automatically generate the return linefeed sequence. Before data is accepted into a record (when writing), the record is filled with spaces, and a return linefeed sequence is appended. Therefore, a blank record is a line of spaces. As you print into the record, your printed characters are overlayed on top of the spaces in the record.

The characters sent to the disk by the print statement are identical to the character stream sent to a device. Commas are not inserted. What goes to the disk is a print image. Generally, this would cause problems when it came to reading back in the data with an INPUT statement. This problem has been avoided by expanding the power of the library routine that fetches values. Spaces, commas, plus sign, and minus sign are all treated as delimiters for numbers. Null values generate numeric values of Ø. It is therefore not necessary to insert commas. When printing numbers using the ";" delimiter, at least one space is inserted in place of the plus sign. When printing values using the "," delimiter, tabbing is done as with normal printing. Remember, what goes to the file is a print image. Example:

```
    PRINT #DISK; 1; 2; 3
```

generates:

```
    1 2 3
```

and can be read back in using:

```
    INPUT3 #DISK; A, B, C
```

If you wish to insert commas:

```
    PRINT #DISK; 1  ;',';  2  ;',';  3
```

generates:

```
    1, 2, 3
```

as before, to read in:

```
    INPUT3 #DISK; A, B, C
```

When inputting from an ASCII file, each input statement addresses one logical line. The process is as follows:

* The logical line is fetched from the disk and placed into a buffer.

* Then, each variable specified in the input statement is retrieved one by one from the buffer.

Since an input buffer is used, the MAXIMUM INPUT LINE LENGTH is 255 characters! If you printed a line longer than this, you will NOT be able to read it!

This buffer is used so that:

* The library routines that fetch variables from the console input buffer can be used for the disk input operation. This saves a considerable amount of run-time memory.

* Since a record may contain any number of logical lines, it is not necessary (as with other BASICs) to write all the data for the record with one PRINT statement.

* It is also not necessary to input all the data in a record with one INPUT statement (provided record sequential access is enabled).

* With S-BASIC, binary files are available for data storage. These files are MUCH faster, as there is no need to convert the ASCII data into the internal binary formats.

ASCII files have three principle uses:

* to hold parameter information. This parameter info can then be manipulated with a text editor.

* to collect printed output. By sending printed output to the disk, instead of the printer, a system utility can be used at a latter time to print the file.

* In moving data from one system to another in a standard ASCII format.

FUNCTIONS

There are two types of functions in Structured BASIC:

User-defined functions
Built-in or intrinsic functions

User-defined Functions

Functions are defined, using the following statement:

```
FUNCTION <name>[(<arg def>[;<arg def>])]=<type>
 <body>
END=<expression>
```

where <name> is the name given to the function.

It is generally a good idea to define all data structures at the beginning of a function or procedure.

When a function is called from an expression, it may require one or more arguments to be passed to it. The type of the argument and where it is to be stored in the function is defined by the argument list enclosed in '()'.

Within the braces is <arg def> for argument definition. The form of the <arg def> is the same syntax as a VAR statement, except that the key word, VAR, is not included.

There may be more than one <arg def> in the argument definition; these are separated with ';'. Thus, the argument list may have many different arguments of differing types.

A function need not have an <arg def> if no arguments are needed. An example of this would be a function that returned a value from a file or I/O device.

Then comes <type>. <type> determines the type of the result of the function.

<body> is the body of the function and is composed of several basic statements. There need not be any body, if the function can be expressed using one expression.

Next comes the END statement, which is the end of the function, and an <expression>, which will determine the result of the function.

The type of <expression> is set by the <type> of the function.

Expressions that appear in the <body> of the function need not be of this type. <body> may be considered a mini program within a program.

Within <body>, other functions may be called, or the function may call itself. Some examples of functions:

```
FUNCTION FAC(I=REAL)=REAL
 IF I=0 THEN 1l ELSE I=FAC*I-1)*I
END=I

To use:
X=FAC(Y+5)

FUNCTION QUB(I=INTEGER)=INTEGER
END=I*I*I

To use:
PRINT QUB(5)

FUNCTION CHAR.STRING(LETTER=CHAR;COUNT=INTEGER)=STRING
VAR COMPOSE=STRING
VAR I=INTEGER
COMPOSE=""
FOR I+1 TO COUNT
   COMPOSE=COMPOSE+LETTER
NEXT
END=COMPOSE

FUNCTION READER=CHAR
VAR BYTE.VALUE=CHAR
INPUT3 (3) TYPE.VALUE
END=BYTE.VALUE
```

Variables that are declared in the argument list or in the body of the function are local to that function.

Different variables with the same name could be declared later in the program.

Functions may also be declared inside of a function, and functions can be declared inside this, and so on and on.

Remember that the <body> of a function is like a mini program, so what you can do in a program, you can do in a function.

If the argument list is long, it may be continued on another line after the ';'. Like so:

```
FUNCTION <name>(<arg def>;
         <arg def>;<arg def>;
         <arg def>  and so on... )=<type>
```

When defining the result type of a function as string, it is not necessary to specify the result length. The result length is set during the evaluation of the expression.

PROCEDURES

Procedures can be thought of as subroutines that can have variables (arguments) passed to them when they are called. Generally, subroutines cannot call themselves without destroying their local data structures, but procedures can call themselves without destroying their local data structures.

When procedures or functions call other procedures or functions, the local data structures are saved on the run-time stack. This is called recursion.

When control is 'returned', these data structures are put back. Only those data structures defined up to the point of call are saved.

Procedures are defined like functions, except that there is no <type> of the procedure and no result <expression>.

The form of the procedure declaration is:

```
PROCEDURE <name.>[(<arg def>{;<arg def>})
 <body>
END
```

Once again, the argument list may be continued on another line after a ';'.

As with functions, the body of a procedure is like a mini program.

Variables can be declared within <body>, files manipulated, and so on.

Functions may be defined within a procedure. Procedures may be defined within a function. Procedures may call themselves or other procedures.

A procedure is invoked with its <name> as the first thing (token) on a line (note the procedure's name may be preceded with a line number).

After the procedure's name, expressions of the same type as the variables in the argument list should follow, separated by commas.

If there are no arguments, then there need not be any expressions following the procedure's name.

```
PROCEDURE NEW. PAGE
 PRINT #LIST;CHR$(OCH);
END

PROCEDURE SORT(KEY=INTEGER)
  <body of code to sort something>
END

PROCEDURE OUTPUT(DEVICE=INTEGER;ARGUMENT;STRING)
 PRINT #DEVICE; ARGUMENT
END

PROCEDURE PRINT. SUM(A,B,C=INTEGER)
 PRINT A+B+C
END
```

How they would be used:

```
NEW PAGE
SORT SOMETHING
OUTPUT LIST,"A JUNK LINE"
PRINT.SUM NUMBER1,NUMBER2,NUMBER3
```

SCOPE OF RECURSION

By scope we are referring to those variables that will not be changed during a recursive call. That is to say, a function may call itself, and it appears as if each invocation of the function has created anew its own local set of variables. For example, consider the following:

```
FUNCTION FAC(I=REAL)=REAL
  IF I=0 THEN I=1 ELSE I=FAC(I-1)*1
END =1
```

This function finds X! (If X=3, X!=3*2*1). Each time FAC calls itself, a new I is created. When FAC ends and returns a result to itself, the value of I appears to have been unchanged; only the result is returned.

The scope, or variables, that are created anew are important to the design of a program. Without a knowledge of the rules of recursion in S-BASIC, some strange things can seem to occur.

Variables will refuse to change values or will appear to change values at random times. Even if our program does not do explicit recursion, the compiler still follows the rules. This prevents unwanted crosstalk between local variables and helps exclude program code interactions (one function or procedure interfering with another).

The rules governing recursion are designed to limit the amount of variables that must be created with each invocation. This saves memory and increases speed of execution. The program following this discussion shows a program that demonstrates these rules. The program is composed of procedures, but the same rules apply to functions and to procedures and functions mixed.

Recursion occurs if:

* The function or procedure calls itself. In this case, any variables declared within the function or procedure, including parameters, are created anew.

* The function or procedure is declared within another function or procedure (nested) and calls a function or procedure that is also declared (nested) within the same function or procedure. This includes the function or procedure calling itself or the function or procedure within which it is nested.

  Only those variables declared within the function or procedure doing the calling are recursive.

In all cases, only those variables declared in the source code up to the point of the invocation are recursive.

In the following program, these rules can be seen in action:

In procedure P1, line 5, a call is made on P0. This call is not recursive. No variables are created. In line 6 of P1, a call is made on P1 itself. This call is recursive, and variable X1 is the variable recreated.

Procedure P2 is an example of nested procedures.

In P3, line 12, procedure PØ is called. Since P0 is outside the scope of P2, there is no recursion.

Procedure P4, line 2Ø, is an example of a nested procedure calling another nested procedure; this call is recursive for X4 only. P4 does not create new variables X2 and X3. That operation is saved for the future, should program flow require it. This helps conserve memory and program execution speed.

In line 25, P2 calls itself, and variables X2, X3, and X4 are created anew. This is the simple rule that all variables declared within the procedure or function up to the point of the recursive call are recreated. Although it serves no obvious purpose, it does help in the allocation of variables and in the prevention of interactions.

```
0001:00 PROCEDURE P0( X0=INTEGER)          Procedure #0, Scope 1
0002:01 END of P)
0003:00
0004:00 PROCEDURE P1( X1=INTEGER )         Procedure #1, Scope 2
0005:01         P0 X1                      No Recursion
0006:01         P1 X1                      Recursion, X1
0007:01 END of P1
0008:00
0009:00 PROCEDURE P2( X2=INTEGER )         Procedure #2, Scope 3
00010:01
0011:01       PROCEDURE P3( X3=INTEGER )   Procedure #3, Scope 3
0012:02               PO X3                No Recursion
0013:02               P2 X3                Recursion, X3
0014:02               P3 X3                Recursion, X3
0015:02       END of P3
0016:01
0017:01       PROCEDURE P4( X4=INTEGER )   Procedure $4, Scope 3
0018:02               P1 X4                No Recursion
0019:02               P2 X4                Recursion, X4
0020:02               P3 X4                Recursion, X4
0021:02               P4 X4
0022:02       END of P4
0023:01
0024:01       P0 x2                     No Recursion
0025:01       P2 x2                     Recursion, X2 X3 X4
0026:01       P3 X2                     Recursion, X2 X3 X4
0027:01
0028:01 END of P2                       End of P2 and end of
                                           Scope
```

QUICK-REFERENCE LIST OF

INTRINSIC FUNCTIONS

Structured BASIC provides several built-in functions for commonly-used algebraic and string operations.

Functions are of the form:

<name>(<argument list>)

where <name> is the name of the function
and <argument list> is a list of one or more expressions separated by commas and enclosed in braces.

The type of each expression in the <argument list> is defined by the abbreviations: RD, RS, F, I, S, and C, as described in chapter 2.

For example, the random number generator is a function of the form- RND(RS),
where RS means an expression of type real.

ABS(RS)

This function returns the absolute value of the real expression.

If RS is positive, then RS is returned.
If RS is negative, then RS is returned as positive.

ASCII(S) / ASC(S)

Returns an integer (type integer) that is equal to the first character of the string argument.

ATN(RS)

Returns the angle whose tangent is RS (in radians).

CHR(I) / CHR$(I)

Returns a one-character string, having the ASCII value of I.

COS(RS)

Returns the cosine of RS. RS is in radians.

EXP(RS)

Returns e (2.71828) to the power of RS.

FCB$(S) / FCB(S)

Returns a string equal to a valid format for an FCB.

FFIX(F)

Returns the integer part of the fixed type expression.

The return is a type fixed number.

FINT(F)

Returns the next lowest integer <=F.

Returns a type fixed result.

FIX(RS)

Returns the integer of RS.

This integer is of type RS
and is generated by truncating the fractional part.

FRE(I)

If I is false (I=Ø), returns amount of free memory by a comparison of the 8Ø8Ø stack and the procedure/function stack (memory location 1Ø9H).

If I is true, returns the number of blocks used on the current drive.

It is preferable to use this function to SIZE(*.*), because FRE looks at the bit map in memory, where SIZE looks at the directory on the disk.

HEX$(I)

Returns a string of 4 characters.

Each character position is set to the hex character of I,
i.e., a string of hex characters equal to I.

INP(I)

Does an 8080 input instruction from port I.
The result is returned as an integer.

INSTR(I<S1,S2)

Searches for S2 within S1, starting at the Ith character of S1.
The result is an integer =0, if S2 is not found in S1,
or equal to the character position in S1 where S2 was found.

INT(RS)

Returns the next lowest integer <= RS. The result is of type real.

LEFT$(S,I)

Returns the leftmost I characters of S as a string result.

LEN(S)

Returns an integer with a value equal to the number of characters in S.

LOG(RS)
Returns the natural log (Ln or log to the base e) of RS.

MID(S,I1,I2) / MID$(S,I1,I2)

Returns a substring of S, as a string result. Characters are taken, starting at the I1th character position in the string, for a count of I2 characters. Mid is used for insertion. It is covered in the chapter on expressions.

NUM$(RS) / STR$(RS)

Returns a string that is composed of characters representing the value of RS, as if it was to be printed.

PEEK(I)

Returns an integer equal to the memory location I.

POS(I)

If I is a positive integer,
POS(I) returns an integer
whose value is equal to the print position on output channel I.

If I is a negative integer,
it is converted to a positive integer and
the current line count for that channel is returned.

The print position is set to 1 by an ASCII CR.

The line count is set to 1 by an ASCII FF.

Since there is no -Ø to return the console line count, 255 is used.

RIGHT(S,I)

Returns the characters of the string, S,
starting at the Ith character through the last character of the string, S.
The result is a string.

RIGHT$(S,I)

Returns the rightmost I characters of S as a string result.

RND(RS)

If RS is true (<>Ø), returns a random number between Ø and 1.

If RS is false (=Ø), returns last number generated.

SGN(RS)

If RS>Ø, returns 1.
RS=Ø returns Ø.
RS<Ø returns -1.

SIN(RS)

Returns the sin of RS. RS is an angle measured in radians.

SIZE(S)

Returns the size of a disk file in blocks.

Wild card file names can be used as described in the system manuals.

The file name is specified by the string argument.

SPACE$(I) / SPC(I)

Returns a string result that is composed of I number of spaces.

SQR(RS)

Returns the square root of RS.

STRING(I1,I2) / STRING$(I1,I2)

Returns a string result that has a length of I1 and is composed of characters whose ASCII value is equal to I2.

TAB(I)

Returns a string of spaces that will move the print head from the current print position to print position I.

The channel used is the last one accessed.

TAN(RS)

Returns the tangent of the angle RS.
RS is in radians.

VAL(S)

Returns a RS that is equal to the numeric value of S, as if the number was entered using an INPUT statement.

XLATE(S1,S2)

The result is a string. This string is the result of a translate function.
S1 is the source string, and S2 is the table string.

Each character from S1 is used as an index into S2, and this character is placed into the result. This process continues character by character until S1 is exhausted.

COMPILER OPERATION

From a basic program, S-BASIC produces zero, one, or two files.

One file is a .PRN file, which is a print file of the source with line numbers and error messages, if any.

The other file is a .COM file that can be executed directly in the TPA.

S-BASIC is a true compiler, in that the .COM file is Z80 code.

The compiler is initiated by typing:

```
SBASIC filename
```

or

```
SBASIC filename.parameters
```

For both cases, filename refers to the basic source file and is ASSUMED to have an extent of .BAS.

When SBASIC loads and executes, it prints the following sign-on message on the console:

```
SBASIC Compiler Version x
```

where x refers to the version number of the compiler.

When the compiler is initiated by the command, SBASIC filename, one file is generated, called filename.COM.

This is the execution or run-time file for use in the TPA.

If errors are detected by the compiler, the file, filename.COM, is not generated.

Depending on when the error is found, if filename.COM existed before compilation, it may be deleted.

A copy of the source file with line numbers and error messages is sent to the console during compilation.

The second part of the command allows the user to specify parameters to the compiler. These parameters direct where the files, if any, are to go. The form is:

```
SBASIC filename,a1a2a3
```

where, as before, filename is assumed to have an extent of .BAS. The arguments a1, a2, and a3 designate the following:

a1 designates from which disk drive the source or .BAS file is to come:
A for A drive, B for B drive, and so on.

a2 designates on which disk drive the object or .COM file is to be placed. A is for A drive, B for B drive, and so on. The use of the letter, Z, specifies that the generation of the object file is to be skipped.

a3 This argument specifies where the print or .PRN file is to go. A capital letter specifies the drive where the file is to be placed. If the letter, X, is used, the print output is sent to the console. If the letter, Y, is used, the print output is sent to the list device. the letter, Z, suppresses the generation of the listing.

Some examples:

1. SBASIC TEST
2. SBASIC TEST.BBX
3. SBASIC TEST.BZY
4. SBASIC TEST.BAB

In example 1:

The basic source file, TEST.BAS, is compiled from the current drive.

If no errors were found, the file, TEST.COM, is placed on the current disk.

A listing is sent to the console during compilation.

In example 2:

TEST.BAS is compiled from drive B.

If no errors were found, the object file, TEST.COM, is placed on drive B.

A listing is sent to the console.

In example 3:

The source file comes from B, and there is no generation of TEST.COM.

The listing is sent to the list device.

In example 4:

The source file is on drive B.

The object file is to be placed on A.

The print file, TEST.PRN, is sent to drive B.

Listings sent to the list device are headed with the SBASIC sign-on.

Page ejects (form feeds) are used to separate pages.

Files Used During Compilation

When the compiler is running, it uses several files which must be on the currently-selected drive. These files are SBASE.COM, OVERLAYB.COM, BASICLIB.REL, and USERLIB.REL.

Commands Used During Compilation

During compilation, several commands may be given to the compiler via the $<command> statement included in the source file. The commands are:

$LINES Command

Under normal operation, in the case of a run-time error, the compiler emits code that enables the printing of line numbers.

This production allows control to be returned to the system during program execution by typing a control--C.

This production takes up memory in the object program.

This feature can be disabled, using the LINES command.

The LINES command changes the state of a compiler toggle from:
true (produces line number references) to
false (does not produce line number references)
and back again.

So, when debugging is finished, this statement can be used to suppress the generation of line number references. Then, if a run-time error is encountered, the error message is printed, but no line number is given.

As the compiler generates code, source line information is included so that run-time errors can print the line number that the error occurred on. Also, by typing a control--t at the console, a run-time trace can be produced. This code generation also checks for a control--c as a panic out of program execution. This code production is normally on. It can be toggled on and off by the statement:

```
$LINES
```

This takes about 7 bytes per source line. By turning this production off, some memory can be saved. Also, by turning this production off, the program may run faster.

$TRACE Command

This command works with the LINES command. When line number references are being generated, it is possible to have the line numbers printed on the console at run-time. This traces the flow of the program as it executes.

Line number references are generated for every <statement> given in this manual. They are also generated when a <line number> is given. For example:

```
0516:     123       IF A THEN PRINT
0517:               IF Q THEN RETURN
```

For line 516, three line numbers will be printed:

One for the 123
Another for the statement, IF....
A third for the PRINT statement within the IF.

In line 517, two would be generated:

One for the IF... statement
One for the RETURN.

This feature allows not only the execution of a program to be followed, but for the internal flow of instructions to be monitored.

For example: an IF...THEN need not transfer control, and with $TRACE in effect, it is possible to determine if the <expression> of the IF statement was true or false. This follows for other statements as well.

A trace for the above program could be:

```
[0516] [0516] [0516] [0517] [0518]
```

This trace would indicate that statement 516 was executed.

A was found to be true, and the PRINT statement was executed.

Then statement 517 was executed, Q was found to be false, so the RETURN was not executed.

Program execution continued on to line 518 (not shown).

The $TRACE command statement works like the $LINES statement, in that a compiler toggle is changed from false to true or true to false with each occurrence of $TRACE in the source file. This command can be used to frame the portion of basic source file for which a trace is needed, but avoid tracing program steps that need not be traced.

A run-time trace can be turned on and off by typing a control--t at the console.

$PAGE Command

This statement causes an ASCII form-feed to be sent to the listing device, causing a skip to the next page.

$CONSTANT Command

This command is used to define an integer symbolic constant at compile time. The form of the statement is:

```
$CONSTANT <NAME>=<INTEGER VALUE>
```

This will create a symbol named <NAME> and will substitute <INTEGER VALUE> each time it is used. This constant can be used whenever an integer constant is called for. For example:

```
$CONSTANT KEY.LENGTH=8
VAR FIRST.KEY,LAST.KEY=STRING;KEY.LENGTH
```

It is important to remember that this constant is a compile-time constant and cannot be changed at run-time.

It can be used in an expression as any other integer constant would be.

$INCLUDE Command

This command is used to access a library file.

The form of this statement is:

```
$INCLUDE <FILE NAME> [<MODULE NAME>
```

The <NAME> and <MODULE NAME> should not include reserved characters.

The foregoing statement is used with these statements:

```
$MODULE <MODULE NAME>
  <body of module>
$END.MODULE
```

The simplest form is when <MODULE NAME> is not given.

With this form of the statement, the file <FILENAME>.BAS is inserted into the source program and compiled.

When the end of <FILE NAME>.BAS is reached, the compiler returns to the main source program.

<FILE NAME> may be an S-BASIC program to be included in the main source, or it may be a library of S-BASIC programs or modules.

When <FILE NAME> is a library file, composed of basic program modules, the idea is to pluck out only those modules that are needed. This is done by giving the <MODULE NAME> in the $INCLUDE statement that is needed. Some examples will follow.

Modules are defined within the library file by the use of the $MODULE and $END.MODULE commands to frame the beginning and the end of the module.

```
$MODULE EXAMPLE
  REM THIS IS AN EXAMPLE MODULE
  PRINT EXAMPLE
$END.MODULE
```

Thus, to pluck out module EXAMPLE from above (assuming a library file called EX-LIB.BAS), we would use:

```
$INCLUDE EX-LIB EXAMPLE
```

Naturally, there may be many modules per library file. The nesting of $INCLUDE (i.e., having an included file include another file) is limited only by memory. Each nest uses about 160 bytes of memory.

The module's name is not a symbol to the compiler, so functions, procedures, and variables can use its name.

An error within an included module aborts the compilation within the module.

$LIST Command

This command is used to toggle the listing of source from the compiler off/on. Listing is at first on. When $LIST is encountered in the source program, the list toggle is flipped. Thus, $LIST is used to change the listing of a program, first off, then, on, then off, then on...

This statement is often used with $INCLUDE to prevent the listing of included source.

```
$LIST
$INCLUDE PROGR
$INCLUDE ISAM INSERT
$INCLUDE ISAM HASH
$LIST
```

$LIST [ON/OFF]

$LIST can also be followed by one of the words, ON or OFF.

ON turns listing on. OFF turns the listing off.

$STACK Command

The $STACK command is used with the $LOADPT command (given below) to set the run-time stack location when the stack location must be set by the programmer. Its form is:

```
$STACK <INTEGER CONSTANT>
```

$LOADPT Command

This command is used with the $STACK command to change the location of the compiled code from 100H to another location in memory. Its form is:

    $LOADPT <INTEGER CONSTANT>

where <INTEGER CONSTANT> is a memory address where the run-time code is to execute.

STATEMENTS FOR S-BASIC

In the following, a ^ indicates where the statement may be broken with returns.

BEGIN ... END

Where a BASIC statement is called for, the following structure may be used instead:

```
BEGIN
  (some basic statement, including more BEGINs)
END
```

```
          i.e., IF TRUE THEN
                    BEGIN
                      (some basic statements)
                    END
```

The statements:

CALL <exp>
This statement is used to execute assembly language routines. <exp> is an integer expression that determines the memory address to be called.

CHAIN <file name>
This statement allows an S-BASIC program to load and execute another S-BASIC program, where <file name> is the file name to be loaded and executed.

EXECUTE <file name> [,<expression>]
This statement loads and executes a .COM file.
<file name> is the file name of the desired .COM file.
<expression>, if used, is a string-type valid operating system command line, which is executed AFTER the .COM file has been loaded, executed, and control has been returned back to the operating system.

DATA <ascii data>
This statement stores the <ascii data> in the program for use at run-time. All DATA statements must appear together in the program, separated by commas. (See READ, RESTORE.)

READ <name> {,<name>}
This statement reads from a data list of a DATA statement.
Items are read from the beginning to the end of the list.
<name> is a variable to be stored with the current value of data.
As items are read from the data list, a pointer is moved through
the list to determine which is the next item to READ. You can
position the pointer with the RESTORE statement.

RESTORE [<line number>]
Positions the pointer at <line number> of a DATA statement.
If <line number> is not used, then the pointer is set to the
first DATA statement in the program.

OUT <I1,I2>
This statement sends data to an I/O port.
<I2> is sent to <I1>. <I1> is an 8080 I/O port number.

POKE <I1,I2>
This statement sends data to an I/O port.
<I2> is sent to <I1>. <I1> is a memory address.

CONTROL.C.TRAP ON/OFF
When a program is executing and waiting for an input to the INPUT
statement, typing a CTRL-C will return to the operating system.
This feature can be turned ON and OFF with the CONTROL.C.TRAP
statement. It is intially ON.

RECORD.SEQUENTIAL ON/OFF
This statement can turn sequential access of a random record ON
and OFF. If it is ON, you can read or write part of a record,
branch off and execute some statements, and return to continue
reading or writing within the record from where you left.

However, you cannot go back and re-read or re-write the same
record. If you try, this will cause a read or write past the end
of record. If it is OFF, then all the variables in the random
record must be read or written with one statement. It is
initially ON.

REPEAT <statement> UNTIL <expression>

This means that <statement> will be executed until <expression>
is true.

WHILE <expression> DO <statement>

If <expression> is true, then do <statement> and repeat.

IF <expression> THEN ^<statement> ^[ELSE ^<statement>]

If the <expression> is true, then do <statement>.
If the ELSE is used, then the ELSE <statement> is done if
<expression> is false.

VAR <name1>[,<namen>]=<type>[:length]
COM " " " "
BASED " " " "

Declare variables and their type.

If the <type> is string, then a length may be given. The default value is 80.

VAR assigns storage in the data storage field.

COM assigns storage in the common field which remains intact during CHAINING of programs.

BASED is not assigned a location by the compiler, but may be assigned a location in memory at run-time using the BASE...AT statement.

DIM [COM/BASE] <type> <name>(<arg>{,<arg>}) ...

Declare an array of variable type <type>. The <COM> or <BASE>, if given, determines where the array is stored. <arg> is the size of each dimension of the array.

Some examples:

```
DIM STRING:30 NAMES.FIRST(50) NAMES.LAST(50)
DIM REAL X(10,10,20,5,3,10) Y.CORD(10,3)
DIM COM REAL ARRAY.X(10,10)
```

BASE <name> AT <integer expression>

The variable <name> that was created by a BASED statement is located at the memory location given by <integer expression>

LOCATE <array name> AT <integer expression>

The array that was created with the DIM statement, using a location of BASE, is located at <integer expression>

LOCATION VAR/ARRAY/FILE <integer type var>=<name>

Sets <integer type var> to the location of the based located variable or to the location of a FILE buffer. ARRAY returns the location on the array data field. VAR of a simple data variable.

END

Return to the system.

STOP

Same as END

PRINT <expression list>

Print the value of the expressions.

There may be from Ø to as many expressions as will fit on a line.

Expressions may be separated by a comma (,) or a semicolon (;).

A comma will tab to the print field.

A semicolon will continue in the next character position.

PRINT USING <string exp>;<expression list>

Print <expression list> formatted by <string exp>.

ECHO ON/OFF

Enable/Disable the echoing of characters during input.

INPUT ["prompt string";] <variable list>

Enter the values typed at the console into the <variable list>.

TEXT <channel number>,<delm> <text> <delm>[,/;

The <text> is sent to <channel number> (Ø for console).

<text> may be any number of lines and may contain any characters except <delm>, which is used to frame the <text>.

A comma (,) or semicolon (;) may follow the <delm> and functions the same as a print statement.

LPRINTER
CONSOLE

Changes back and forth the default print device from console to the list device. This is a compiler directive and does not function with transfer of control at run-time.

COMMENT
  <any number of comment lines>
END

Allows the insertion of comment lines in the source file. These comment lines are ignored by the compiler.

REM <a comment>
REMARK <a comment>

One line of remark. It is passed over by the compiler.

ON ERROR GOTO <line number>

In case of a run-time error, control is transferred to <line number> and not to the system.

GOTO (line number>

Transfer control to <line number>

GOSUB <line number>
RETURN

Transfer control to <line number>.
RETURN returns control to the statement just after the GOSUB.

ON x GOTO <line number list>

ON x GOSUB <line number list>

Goto/Gosub line number indexed by x.

```
FUNCTION <name>(<var statement>{;<var statement>})=<type>
        <any number of statements>
END=<expression>
```

Define a function <name> with arguments in the <var statement>.

The <var statement> is the same as for VAR, except that the key word, VAR, need not be given.

<type> is the result type of the function and is the type of the result <expression> that is returned.

Both the argument list <var statement> and VAR statements that are inside the function are local to the function.

Example:

```
    FUNCTION SQR.PLUS.ONE(ARG.X=REAL)=REAL
    VAR Y=REAL
    Y=X
    END=X*Y=1

    FUNCTION SUM(VALUE1=REAL;VALUE2=INTEGER)=REAL
    VAR COUNTER=INTEGER
    FOR COUNTER=1 TO VALUE2
      VALUE1=VALUE1=VALUE1
    NEXT COUNTER
    END=VALUE1
```

```
PROCEDURE <name>(<var statement>[;<var statement>]
<body of procedure>
END
```

Same structure as functions except no value is returned, and the procedure is called with the form:

```
    <name> <argument list>
```

Example:

```
    PROCEDURE LPRINT$(A=STRING)
      PRINT #1; A
    END

    LPRINT$ "THIS IS A TEST LINE"
```

FILES R(x),S(n)...

Define files.

R stands for random.
S stands for serial.

With R, the X (an integer constant) refers to the number of bytes per random record.

For S, n specifies the number of sectors in the files buffer.

This statement sets file channel #0 for random file and file channel #1 for serial.

Up to 32 channels may be defined.

OPEN #n;<string expression>

Opens the file <string expression> on file channel #n

CLOSE #n

Closes the file on channel #n

READ #n,x;<var list> or READ #n;<var list>

WRITE #n,x;<var list> or WRITE #n;<var list>

Read and Write from a random file or a serial file.

INITIALIZE

Reset disk system. Used when disks are changed.

DELETE <string expression>

Delete a file from the disk.

RENAME <string expression> TO <string expression>

Rename a file.

```
CREATE <string expression>
```

Create a file.

```
CASE <expression> OF
<expression1>:<statement>
<expression2>:<statement>
          .
          .
          END
```

Case statement transfers control to the <expression x> that is equal to <expression>.

```
FOR <var>=<expression1> TO <expression2> [STEP <expression3>]
<any number of basic statements>
NEXT
```

Loops starting at <expression1> until <var> is greater than or equal to <expression2>.

A step value may be given.

APPLICATION NOTES

(Ap-Notes)

(The material in this section is used with permission of Digital Research, holder of the copyright. CP/M is a registered trademark of Digital Research.)

The EXECUTE statement allows for the loading and executing of any binary file. However, it does not provide for the setting up of parameters.

In many instances, the binary file which is loaded and executed expects to find parameters at tfcb and tbuff. It is easy for an S-BASIC program to set up these areas before the EXECUTE statement is executed. The following program will demonstrate how this can be done.

In general, binary files to be loaded and executed provide functions such as sorting, merging, and other system utilities.

As to parameters, binary files need:

In general, tfcb (at 5CH) is set up as a file control block with the ASCII name installed and the extent number cleared.

The first byte represents the drive number to be used: one for A: two for B:, and so on.

A value of zero signifies that the currently-selected drive is to be used.

In normal processing of a command line by the CCP, if a third file name is present, the ASCII representation will be placed at tfcb+16. Additionally, the entire command line should be present at tbuff.

You should consult the operator's manual for a particular utility file to determine which, if any, of these fields need to be set up. Experience has shown that some utilities only require the first file name at tfcb to be set up, while others only use the image of the command line stored at tbuff. It should be noted that tbuff also serves the purpose of a disk I/O buffer for some utilities.

The EXECUTE statement is able to regain control of the system after the utility gives control back to the CCP. This is done through the use of the SUBMIT facility. For this to function

properly, the EXECUTE statement must be executed while the system disk (A:) is selected. Please note that the EXECUTE statement will nest the submit file.

```
0001:     VAR FILE.NAME1, FILE.NAME2, COMMAND LINE = STRING
002:      VAR DRIVE.CODE = CHAR
0003:     FILE.NAME1="X.ZOT"
0004:     FILE.NAME2="Y.ZAP"
0005:     COMMAND.LINE=" X.ZOT Y.ZAP"
0006:     DRIVE.CODE=0
0007:
0008:     Based File.control.block, Second.file.name =String
0009:     Based Command.buffer +String
0010:     Based Drive.number, Buffer.length =Char
0011:     Based Extent.number, Record.number =Char
0012:
0013:     Rem The file control block at 5CH
0014:     $Constant tfcb  = 80H
0017:
0018:     Base File.control.block at tfcb
0019:     Base Second.file.name at tfcb+16
0020:     Base Command.buffer at tbuff
0021:     Base Extent.number at tfcb+12
0022:     Base Record.number at tfcb+32
0023:     Base Drive.number at tfcb
0024:     Base Buffer.length at tbuff
0025:
0026:     Rem Place file name into FCB.
               Note: FCB$ sets up ASCII only.
0027:     File.control.block = Fcb$(File.name1)
0028:     Second.file.name   = Fcb$(File.name2)
0029:     Rem Zero extent number and record number
0030:     Extent.number =0
0031:     Record.number =0
0032:
0033:     Rem The byte at 5CH is the drive number.  This byte was
0034:     Rem also the count byte of the string File.control.block
0035:     Driive.number = Drive.code
0036:
0037:     Command.buffer = Command.line
0038:     Buffer.length  = Len(Command.line)
0038:
0040:     ****** End of program ******
```

In addition to the default fcb which is set up at address tfcb, the CCP also constructs a second default fcb at address tfcb+16 (i.e., the disk map field of the fcb at tbase).

Thus, if the user types:

```
    PROGNAME  X.ZOT  Y.ZAP
```

the file, PROGNAME.COM, is loaded to the TPA, and the default fcb at tfcb is initialized to the filename, X, with filetype, ZOT.

Since the user typed a second file name, the 16-byte area beginning at tfcb + 16[10[is also initialized with the filename, Y, and filetype, ZAP. It is the responsibility of the program to move this second filename and filetype to another area (usually a separate file control block) before opening the file which begins at tbase, since the open operation will fill the disk map portion, thus overwriting the second name and type.

If no file names were specified in the original command, then the fields beginning at tfcb and tfcb + 16 both contain blanks (20H). If one file name was specified, then the field at tfcb + 16 contains blanks.

If the filetype is omitted, then the field is assumed to contain blanks. In all cases, the CCP translates lower-case alphabetics to upper case to be consistent with the CP/M (R) file naming conventions.

As an added programming convenience, the default buffer at tbuff is initialized to hold the entire command line past the program name. Address tbuff contains the number of characters, and tbuff+1, tbuff+2, ..., contain the remaining characters up to, but not including, the carriage return. Given that the above command has been typed at the console, the area beginning at tbuff is set up as follows:

tbuff:

+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11	+12	+13	+14	+15
12	b	X	.	Z	O	T	b	Y	.	Z	A	P	?	?	?

where 12 is the number of valid characters in binary, and b represents an ASCII BLANK.

Characters are given in ASCII upper case, with uninitialized memory following the last valid character.

Again, it is the responsibility of the program to extract the information from this buffer before any file operations are performed, as the FDOS uses the tbuff area to perform directory functions.

In a standard CP/M system, the following values are assumed:

boot:	0000H	bootstrap load (warm start)
entry:	0005H	entry point to FDOS
tfcb:	005CH	first default file control block
tfcb+16	006CH	second file name
tbuff	0080H	default buffer address
tbase	0!00H	base of transient area

MERGING AND USING ASSEMBLY LANGUAGE ROUTINES

In this section, we shall discuss merging assembly language routines with the the compiler's output, and then accessing those routines, using the S-BASIC CALL statement.

The basic technique is to utilize the common storage area as a safe location within the compiled code where the assembly language routine may be located. By referring to the System Load Map given elsewhere in this manual, it can be seen that the common storage area directly precedes both the program storage area and the data storage area. Therefore, if some common variable is declared before all other common variables, its position will remain constant in the code production, regardless of changes in the program and data structures. If this common variable is an array of type CHAR (a byte value), then the size of reserved space is directly related (in bytes) to the size of the array. For a one-dimensional array of type CHAR, the absolute address where the common data field can be found is 11AH. Additionally, the location of any common structure may be found programatically by utilizing the LOCATION statement. It should be noted that the common storage area is not disturbed during CHAINing operations.

There are essentially two methods by which the assembly language routine may be placed into the character array.

1. Assemble the assembly language routine to produce a HEX file with its source at 11AH or whatever the determined value is. The compiled code or COM file is loaded into memory under control of DDT or some other system resident debugger. The HEX file is then overlayed on top of the COM file in memory (i.e., where the common data field of the array is). Exit DDT, and save the memory image.

   We now have the assembly language routine bonded with the compiler-produced code. Thus, whenever the COM file is executed, the assembly language routine will be resident and ready for use.

2. As the assembly language routine will actually reside within a character array, it is possible to load it from a disk into the array at run-time. First, the assembly language routine is assembled to produce a HEX file. Then, under control of the debugger, it is read into memory and moved, so that the first byte of the routine is at 100H (the base address of files saved under CCP). Exit the debugger, and save the memory image. What we now have is our assembly language

routine as a binary image in a disk file. It is from this file that the assembly language routine will be drawn at run-time.

In general, the run-time procedure is as follows:

1. A sequential file channel is used.

2. Our disk file containing the binary image is opened on that sequential file channel.

3. Bytes (using a variable of type CHAR) are read from the sequential disk file and placed one by one into the character array. Remember that, in this context, the terms, character and byte, are interchangeable.

   It is vital, during this operation, to know the number of bytes to read from the disk file and place into the array, so as to avoid not reading all of the routine or generating an end-of-file error.

This method allows different routines to be loaded at runtime, as needed.

The following programs are used to demonstrate these techniques. The basic idea of these programs is to read an ASCII text file and produce a listing on the system list device. The assembly language program, STRINGIN, is used to capture text lines from the text file and place them into an S-BASIC simple string variable. The S-BASIC program calls STRINGIN to get each string. It then prints these strings on the list device, adding a title and page number.

You may find the following publications helpful:

Digital Research manual, CP/M INTERFACE GUIDE, section 3.2 "File Control Block Format", pages 10 and 10a

AN INTRODUCTION TO CP/M FEATURES AND FACILITIES, section on the SAVE command

DDT USER'S GUIDE

```
                ;ROUTINE TO READ AN ASCII SOURCE FILE INTO
                ;STRINGS USED BY THE S-BASIC COMPILER
                ;AN ASCII SOURCE FILE IS ASSUMED TO BE IN THE FOLLOWING FORMAT
                ;     ASCII SOURCE LINE
                ;     CARRIAGE RETURN & LINE FEED
                ;   REPEAT ABOVE AS NEEDED FOR EACH LINE OF TEXT
                ;     END OF FILE GIVEN BY A CONTROL-Z OR
                ;     PHYSICAL END OF FILE
                ;
                ;THIS ROUTINE RESIDES IN A COMMON ARRAY OF TYPE CHAR (BYTE)
                ;IT IS CALLED WITH THE BASIC 'CALL' STATEMENT.
                ;
                ;THE FILE CONTROL BLOCK AT 5CH IS USED AS THE PROGRAM'S FCB.
                ;IT IS ASSUMED THAT THE FILE NAME HAS BEEN PLACED IN IT
                ;AND THAT THE PROPER DISK NUMBER IS INSTALLED.
                ;THE DISK I/O BUFFER IS AT 80H <THE DEFAULT BUFFER>
                ;
                ;THE ROUTINE IS CALLED WITH THE FOLLOWING VALUES:
                ;
                ;          A = 0      THE FILE NEEDS TO BE OPENED
                ;          A <>0      THE FCB HAS BEEN OPENED
                ;          HL =       THE LOCATION OF THE STRING TO BE FILLED
                ;                     WITH TEXT FROM THE DISK FILE
                ;A CALL TO OPEN RETURNS NO DATA
                ;
                ;RETURN WITH THE FOLLOWING VALUES:
                ;
                ;          A = FUNCTION RETURN
                ;               0 = OK
                ;               1 = EOF
                ;               3 = NO SUCH FILE
                ;          DE = NUMBER OF CHARACTERS TRANSFERRED

0014 =      READF    EQU  20          ;CP/M READ NEXT RECORD
000F =      OPENF    EQU  15          ;CP/M OPEN A DISK FILE <FCB>
001A =      SETDMA   EQU  26          ;SET DMA DISK BUFFER ADDRESS
0000 =      GOOD     EQU  0           ;GOOD READ FUNCTION RETURN
0001 =      EOF      EQU  1           ;END OF FILE
0002 =      BAD      EQU  2           ;READ PAST EOF
00FF =      BADOPEN  EQU  255         ;BAD OPEN FUNCTION
005C =      FCB      EQU  5CH         ;USING THE DEFAULT FCB
0080 =      IBUFF    EQU  80H         ;INPUT DMA BUFFER FOR DISK I/O
0005 =      BDOS     EQU  5           ;ENTRY POINT TO DOS
001A =      EOT      EQU  1AH         ;CONTROL-Z
000D =      RETURN   EQU  0DH         ;ASCII RETURN
                ;
011A                 ORG  11AH        ;LOCATION OF COMMON DATA
                ;
011A B7     STRINGIN:ORA  A           ;DOES THE FILE NEED TO BE OPENED
011B C23701          JNZ  GETSTR
011E 3E80            MVI  A,128       ;SET UP INDEX SO GETSTR
0120 329B01          STA  INDEX       ;WILL START WITH A DISK READ
```

```
0123 0E0F              MVI   C,OPENF      ;OPEN THE FILE AT FCB
0125 115C00            LXI   D,FCB
0128 E5                PUSH  H            ;SAVE STRING POINTER
0129 CD0500            CALL  BDOS
012C E1                POP   H
012D FEFF              CPI   BADOPEN      ;OPEN OK?
012F 3E03              MVI   A,3          ;FUNCTION RETURN TO BASIC
0131 110000            LXI   D,0          ;NO CHARACTERS TRANSFERRED
0134 C8                RZ                 ;RZ = BAD OPEN
0135 AF                XRA   A            ;A = 0 =>OPEN OK
0136 C9                RET
                ;
                ;FILL UP THE STRING IN MEMORY
                ;TRANSFER CHARACTER TILL CR/LF IN SOURCE
                ;OR MAX STRING LENGTH REACHED
                ;IF THE MAX STRING LENGTH IS REACHED ASSUME CR/LF SEQUENCE

0137 56      GETSTR:   MOV   D,M          ;PICK UP MAX STRING LENGTH
0138 1E00              MVI   E,0          ;NUMBER OF CHARACTER TRANSFERRED
013A 23      L1:       INX   H            ;PAST COUNT BYTE / TO NEXT CHAR
013B CD6101            CALL  GETBYTE      ;GET A CHAR FROM FILE
013E FE1A              CPI   EOT          ;CONTROL-Z?
0140 CA5201            JZ    L2
0143 FE0D              CPI   RETURN       ;END OF TEXT LINE?
0145 CA5901            JZ    L3
0148 77                MOV   M,A          ;SAVE CHAR INTO STRING
0149 1C                INR   E            ;UP COUNT OF CHARS TRANSFERRED
014A 7B                MOV   A,E          ;PHYSICAL END OF STRING
014B BA                CMP   D
014C DA3A01            JC    L1
014F AF      BYE:      XRA   A            ;END OF CALL RET=0 (OK)
0150 57                MOV   D,A          ;DE=COUNT OF CHARS XFERED
0151 C9                RET
0152 3600    L2:       MVI   M,O          ;END OF STRING MARKER
0154 1600              MVI   D,0          ;DE=COUNT OF CHARS XFERED
0156 3E01              MVI   A,EOF        ;END OF FILE
0158 C9                RET
0159 CD6101  L3:       CALL  GETBYTE      ;WAIST LINE FEED
015C 3600              MVI   M,O          ;END OF STRING MARKER
015E C34F01            JMP   BYE
                ;
                ;THIS ROUTINE GETS BYTES SEQUENTIALLY FROM THE DISK FILE
                ;
0161 E5      GETBYTE:PUSH    H            ;SAVE STRING POINTER
0162 D5                PUSH  D            ;SAVE COUNTS
0163 3A9801            LDA   INDEX        ;INDEX INTO DMA BUFFER
0166 FE80              CPI   128          ;128 BYTES PER BUFFER
0168 C28C01            JNZ   ABYTE
0168 118000            LXI   D,IBUFF      ;INPUT BUFFER
016E 0E1A              MVI   C,SETDMA
0170 CD0500            CALL  BDOS
0173 115C00            LXI   D,FCB        ;READ A SECTOR
```

```
0176 0E14              MVI     ;CREADF
0178 CD0500            CALL    BDOS
017B B7                ORA     A
017C CA8C01            JZ      ABYTE
017F 218000            LXI     H,IBUFF     ;END OF FILE.  FILL BUFFER
0182 0680              MVI     B,128       ;WITH CONTROL-Z
0184 361A      G1:     MVI     M,EOT
0186 23                INX     H
0187 05                DCR     B
0188 C28401            JNZ     G1
018B AF                XRA     A           ;NEW INDEX VALUE
018C 4F        ABYTE:  MOV     C,A
018D 3C                INR     A           ;BUMP COUNTER
018E 329801            STA     INDEX
0191 0600              MVI     B,0         ;INDEX INTO IBUFF TO GET CHAR
0193 218000            LXI     H,IBUFF
0196 09                DAD     B
0197 7E                MOV     A,M
0198 D1                POP     D           ;RECOVER COUNTS
0199 E1                POP     H           ;RECOVER STRING PONTER
019A C9                RET
               ;
019B 80        INDEX:  D8      128         ;INDEX INTO BUFFER
               ;
0082 =          LENGTH  EQU     $-STRINGIN

019C                   END
```

```
0001    $LINES
0002:   COMMENT
0003:        CODE.SIZE IS THE SIZE OF THE ASSEMBLY LANGUAGE ROUTINE
0004:        GET STRING IS SET TO POINT AT THE LOCATION OF THE COMMON
0005;        DATA FIELD.  THIS FIELD WILL NOT CHANGE POSITION PROVIDED
0006;        NO OTHER COMMON DATA ITEMS ARE CREATED BEFORE IT.
0007:   END
0008:
0009:   $CONSTANT CODE.SIZE    = 81H
0010:   $CONSTANT GET.STRING   = 11AH
0011:   $CONSTANT LINE.LENGTH  = 132
0012:   $CONSTANT TBUFF        = 80H
0013:   DIMENSION COMMON CHAR ASSEMBLY LANGUAGE ROUTINE <CODE SIZE>
0014:
0015:   FUNCTION STRING IN <HL = INTEGER> = INTEGER
0016:         VAR A PSW, DE, BC = INTEGER
0017:         IF HL=0 THEN A.PSW=0ELSE A.PSW = OFF00H
0018:         CALL < GET STRING, HL, DE, BC, A.PSW >
0019:         END = A.PSW / 256
0020:
0021:   VARIABLE LINE = STRING  :LINE LENGTH
0022:   VARIABLE TEXT.LINE, PAGE = INTEGER
0023:   VARIABLE BUFF.LENGTH = CHAR
0024:   VARIABLE FILE.NAME = STRING:14
0025:   BASED BUFF.NAME = STRING
0026:
0027:   LOCATION VAR TEXT.LINE = LINE
0028:   BUFF.LENGTH = PEEK<TBUFF>
0029:   BASE BUFF.NAME AT TBUFF
0030:   POKE TBUFF, BUFF. LENGTH
0031:   FILE.NAME = BUFF.NAME
0032:
0033:   REM ASSUME FAB SET UP BY THE CCP.  FIRST OPEN FILE
0034:   IF STRING. IN<0><>0 THEN
0035:     BEGIN
0036:       PRINT "FILE NOT FOUND."
0037:       STOP
0038:     END
0039:
0040:   REM PRINT THE FILE TO THE SYSTEM LIST DEVICE
0041:   LPRINTER
0042:   PAGE = 1
0043:   WHILE STRING.IN<TEXT LINE>=0 DO
0044:   BEGIN
0045:     IF POS<-1>>60 OR PAGE=1 THEN
0046:       BEGIN
0047:         IF PAGE>1 THEN PRINT CHR<0CH> ELSE PRINT
0048:         PRINT :FILE:"; FILE NAME, TAB<70>; "PAGE:'; PAGE
0049:         PRINT
0050:         PRINT
0051:         PAGE = PAGE + 1
0052:       END
```

```
0053:       PRINT LINE
0054:     END
0055:     ****** END OF PROGRAM ******
```

```
0001:     REM CODE SIZE IS THE CODE SIZE -1 <ARRAY INDEX STARTS AT 0>
0002:     $CONSTANT CODE.SIZE = 81H
0003:     $CONSTANT ICHANNEL = 0
0004:
0005:     DIMENSION COMMON CHAR ASSEMBLY LANGUAGE.ROUTINE <CODE.SIZE>
0006:     VARIABLE MODULE = STRING
0007:     FILES S<1>
0008:     MODULE = "STRINGIN"
0009:
0010:     REM INCLUDE ROUTINE TO READ MODULE INTO THE ARRAY
0011:     $INCLUDE CODEGET
0012*     REM CODE SIZE IS THE SIZE OF THE ASSEMBLY LANGUAGE ROUTINE
0013*     REM ICHANNEL IS THE SERIAL CHANNEL TO USE FOR READING
0014*     REM MODULE IS THE FILE NAME TO READ
0015*
0016*     OPEN #ICHANNEL, MODULE, 1
0017*     VARIABLE BYTE  =CHAR
0018*     VARIABLE INDEX =INTEGER
0019*
0020*     FOR INDEX = 0 TO CODE SIZE
0021*          READ #ICHANNEL; BYTE
0022*          ASSEMBLY.LANGUAGE ROUTINE<INDEX> = BYTE
0023*     NEXT INDEX
0024*
0025:     REM THIS PROGRAM IS JUST TO SHOW HOW TO USE CODEGET
0026:     ****** END OF PROGRAM ******
```

```
DDT VERS 2.Ø
-@ FIRST METHOD 1
?
-@ READ IN THE COMPILER PRODUCED
?
-@ CODE "PRINT.COM"
?
-IPRINT.COM
-R
NEXT PC
1ØØØ Ø1ØØ
-@ NOW GET THE hex FILE
?
-ISTRINGIN.HEX
-R
NEXT  PC
1ØØØ ØØØØ
-@ EXIT AND SAVE
?
-^C
A>SAVE 15 PRINT.COM


DDT VERS 2.Ø
-@ NOW METHOD 2
?
-@ PREPARE MEMORY
?
-F1ØØ,2ØØØ,Ø
-@ FIND DISPLACEMENT
?
-H1ØØ,11A
Ø21A FFE6
-@ GET THE FILE
?
-ISTRINGIN.HEX
-RFFE6
NEXT  PC
Ø182 ØØØØ
-@ EXIT TO CCP AND SAVE
?
-^C
A>SAVE 1 STRINGIN
A>
```

INDEX FOR BEGINNER'S SECTION

INDEX FOR REFERENCE SECTION