GC28-8303-2

Systems

VS BASIC Language

IBM

Third Edition (December 1976)

This edition replaces the previous edition (numbered GC28-8303-1) and its technical newsletter (numbered GN26-0804) and makes them obsolete.

This edition applies to Release 3 of VS BASIC, program number 5748-XX1, and to any subsequent releases unless otherwise indicated in new editions or technical newsletters.

Significant changes to the VS BASIC language are summarized under "Summary of Amendments" following the list of figures. Technical changes made are indicated by a vertical bar to the left of the change. Editorial changes that have no technical significance are not noted.

Information in this publication is subject to significant change. Any such changes will be published in new editions or technical newsletters. Before using the publication, consult the latest *IBM System/370 Bibliography,* GC20-0001, and the technical newsletters that amend the bibliography, to learn which editions and technical newsletters are applicable and current.

Requests for copies of IBM publications should be made to the IBM branch office that serves you.

Forms for readers' comments are provided at the back of the publication. If the forms have been removed, comments may be addressed to IBM Corporation, P.O. Box 50020, Programming Publishing, San Jose, California, 95150. All comments and suggestions become the property of IBM.

Preface

This book has two parts. Each contains a comprehensive description of the VS BASIC language—the parts differ in the way that the language is presented.

In Part I, VS BASIC concepts are discussed (for the remainder of the publication, the terms "VS BASIC" and "BASIC" will be used interchangeably). Simplest concepts are presented first, along with those that are essential to any BASIC program. More complex features of the language are then gradually introduced— with the aid of many examples. Once the fundamentals of the language have been discussed, the presentation is modular. Thus, someone not interested in a particular advanced feature of BASIC (for example, matrix multiplication, or record-oriented files) can skip the section describing that feature.

If you are new to programming, or if you have never used BASIC or a similar high-level programming language, you should start reading with the first chapter of Part I.

Part II presents the BASIC language in reference fashion and should be used like a dictionary or a book of rules. BASIC statements are arranged in alphabetic order; in some cases, statements that are related to each other are discussed together, and the alphabetic entry for one statement may direct you to another statement for the discussion. Each statement description includes discussions of syntax and statement action, rules for coding the statement, and examples of coded statements. Little or no emphasis is placed on explaining examples or concepts; instead, there are references back to sections of Part I for instructional discussion of a subject.

If you have used BASIC or another high-level language before, you will probably want to proceed to Part II immediately to acquaint yourself with the VS BASIC statements. It is likely that you will find yourself, guided by the cross-references in Part II, returning to sections of Part I for detailed discussions of certain language features. Once you've mastered the language, you will probably need to refer only to Part II and even then only occasionally, say for a quick reference to a coding rule.

In several places in this book, you will find references to a *Terminal User's Guide* and a *Programmer's Guide*. These publications describe how to use VS BASIC in different environments — VSPC, CMS, TSO, DOS/VS, and OS/VS. Each of these publications is intended to supplement the information presented in this language manual to provide a complete guide to using VS BASIC in a particular environment. Each guide contains a master index containing entries to this manual as well as entries specifically discussed in the Guide.

The publications are:

- *VS BASIC: TSO Terminal User's Guide,* SC28-8304
- *VS BASIC: CMS Terminal User's Guide,* SC28-8306
- *VS BASIC for VSPC: Terminal User's Guide,* SH20-9060
- *VS BASIC: OS/VS and DOS/VS Programmer's Guide,* SC28-8308

Note to Batch Users: In some places in this book, particularly in the discussions of the INPUT and PRINT statements, you will find references to terminals and terminal operations. In a batch environment, substitutions should be made for these terms, as follows:

terminal (input function) —	system input device, or user data file
terminal (output function) —	system output device, or user data file
print line —	system output record, or user data record
carrier return —	end of output record
carrier position, line position —	output record position

Contents

Figures

Summary of Amendments

Number 3

Release 3

INPUT FROM Statement

This new statement permits the user to obtain input data normally retrieved from the user terminal, from a user data file.

PRINT TO Statement

This new statement permits the user to direct output data, normally printed at the user terminal, to a user data file.

Record-Oriented Files Extension

VS BASIC support of record-oriented files is extended to include the support of relative record accessing of record-oriented files and the ability to reuse a record-oriented file.

Error Handling During Program Execution

The OPEN FILE, CLOSE FILE, and RESET FILE statements now accept error clauses, similar to those supported by the GET and PUT statements.

A new statement, the ON statement, together with new (read-only) internal variables, provides for additional information relative to errors occurring at program execution time.

Buffered-Ahead Terminal Input

This new facility allows entry of input data to satisfy the request of more than one INPUT statement, on a single input line.

Alphabetic Equivalent Operators

This extension allows for the use of alphabetic equivalent operators for the relational and logical operations in addition to the special characters now supported for this purpose.

OPTION Statement

This new statement permits the user to specify certain options to apply to a given BASIC program that override normal VS BASIC actions.

DATA Statement Extension

This extension permits character data to be included in a DATA statement without surrounding single or double quotation marks, similar to that now supported by the INPUT statement.

FORM Statement Extension

The PIC clause of the FORM statement is extended to accept character data and to include new trailing data form specifications for identifying a credit (CR) and a debit (DB) condition. In addition, a literal enclosed in quotation marks is allowed in a FORM statement.

PRINT USING Image Statement Extensions

These extensions provide the user with the following capabilities:

- The ability to combine the output of a PRINT USING Image statement with the output of any other PRINT USING statement onto a single output print line.
- The ability to reuse the Image statement format specifications for the creation of a single line of output when the output list of the PRINT USING statement contains more scalar expressions than the number of format specifications in the corresponding Image statement.

New Intrinsic Function

A new intrinsic function, CHR, is provided which converts an arithmetic expression into an equivalent character form.

Program Listing Comments Extension

This extension permits comments to be added to most BASIC statements through the use of the REM keyword. This optional comment, consisting of the keyword, followed by the comment, is appended to the end of a BASIC statement.

Number 2

Date of Publication: March 26, 1976

Form of Publication: TNL GN26-0804 to GC28-8303-1

VSPC Support of VS BASIC

New: Program Feature

VS BASIC will now operate under VSPC (VS Personal Computing) as well as under CMS (Conversational Monitor System) and TSO (Time Sharing Option). VSPC operates in three VS operating systems: OS/VS1, OS/VS2, and DOS/VS. It offers the user both time sharing and batch processing. The following changes have been made:

- In VSPC the length of the character string used to pass a value in a CHAIN statement to a chained program can be as many as 255 characters.
- Users can access VSAM record-oriented files but must have the data processing installation personnel define them. A VSPC user can define his own VSPC record-oriented files.
- Appendix A. "Implementation Considerations" has been expanded.

Number 1

Date of Publication: July 1974

Order No. of Publication: Revision GC28-8303-1

New Appendix and Glossary

New: Documentation Only

Appendix B, which describes the collating sequence of the VS BASIC character set, and a glossary have been added. The former Appendix B is now Appendix C.

Internal Constants and Pseudo-Variables

New: Documentation Only

Descriptions of internal constants and pseudo-variables have been added to Part I. In Part II, the values in the table of internal constants have been extended to 7 and 15 significant digits.

Implementation Considerations

Maintenance: Documentation Only

The list of specific considerations for using VS BASIC in the CMS, TSO, DOS/VS, and OS/VS environments has been expanded. Especially significant are the file-naming conventions for each environment.

File Information

Maintenance: Documentation Only

Additional information about file usage has been included in Part I; for example, repositioning files, the use of generic keys, the size of rewritten records in record-oriented files, and opening files with the OUT keyword.

Arrays

Maintenance: Documentation Only

The discussion of arrays in Part I has been expanded to include defining arrays by context, the distinction between MAT and LET, and additional examples. In Part II, additional examples have been added to the array assignment statement.

Trailing Characters in FORM Statement

Maintenance: Documentation Only

Changes have been made in Parts I and II to clarify the discussion of the FORM statement, and in Part II to identify the use of the symbols +,-, and $ as trailing characters.

Floating-Point Numbers

Maintenance: Documentation Only

The discussion of floating-point numbers in Part I has been clarified.

Delimiters for Input Data

Maintenance: Documentation Only

Sections have been added to Part I which explain the use of the slash and the comma to delimit input data, and the use of the comma during input as a continuation character.

Additional Rules for BASIC Statements

Maintenance: Documentation Only

Rules have been added to these BASIC statement specifications in Part II: DEF, DELETE FILE, FORM, GET, OPEN, PRINT, PUT, READ FILE, REREAD FILE, RESET, REWRITE FILE, and WRITE FILE.

Print Zones

Maintenance: Documentation Only

The discussion of print zones in Part I has been expanded.

Intrinsic Functions

Maintenance: Documentation Only

The discussion of intrinsic functions in Part I has been clarified. In Part II, the description of the DAT, JDY and RND functions in the table of intrinsic functions has also been clarified.

Additional Examples Throughout

Maintenance: Documentation Only

More examples have been included in both Part I and Part II.

Part I. Using VS Basic

- Introduction
- Your BASIC Program
- Testing and Controlling Program Data
- Using Arrays
- Functions and Subroutines
- PRINT USING Image and FORM
- Program Chaining
- Stream-Oriented Files
- Record-Oriented Files
- Program Error Handling

Introduction

Chances are you want to use a computer because there's something you want to do—fast. It may be a problem to be solved, a report to be written, some information to be tested or analyzed. You're reading this manual because you want to do that something in the easiest possible way—using the BASIC language.

Essentially, all the things you can do with BASIC and the computer fall into one of these categories:

- Getting information into the computer
- Moving information around inside the computer
- Performing calculations on numeric information—adding, subtracting, multiplying, etc.
- Testing information to see if it meets certain requirements.
- Getting information (usually your result) out of the computer.

BASIC, like any other programming language, gives you many ways to do each of these things. Some are very simple and straightforward, others slightly more complex. Each method has its own particular advantages. In the rest of Part I, we're going to show you all the simple, straightforward ways first; then, if you want to learn the more complex methods, you can go on to read about them. You'll probably find that once you've mastered the easy things you'll want to go on, for the sake of getting your work done even faster and more efficiently.

Your BASIC Program

No matter what it is you want to do on the computer, you'll have to have a logical plan for getting it done. You can call this logical plan your "program." If you want to solve a simple problem, such as figuring out someone's bowling average, your plan would look like this:

1. Collect all that person's bowling scores.
2. Add up all the scores.
3. Divide the sum by the number of scores.
4. Display the result.

If you wanted to print a report, your plan might look like this:

1. Print a heading that identifies the report.
2. Set up the format of the report, deciding how many columns it should have and what should go where.
3. Print headings for your columns.
4. Collect all the information that is to appear in the report.
5. Put all the information in the proper order.
6. Print the information.

Each of the steps of your plan can be transformed into one or several BASIC statements. The collection of statements is called a BASIC program. Writing your program means putting all the necessary statements together, in the proper order. When you've done that, you are ready to *execute,* or *run,* the program—that is, have it do what it was written to do. You can run your program right after you've written it, or you can save it and run it at a later time. Theoretically, you write your program once, but can run it many times, as often as you like.

In this book, we're going to concentrate on telling you how to write your BASIC program. Your *Terminal User's Guide,* or *Programmer's Guide,* will give you exact information on how to run and/or save it. But we will be mentioning in later sections some of the things that can happen when you're running your program.

Now, let's see what a BASIC statement looks like:

```
00010    LET A = 3 REM ASSIGN VALUE TO A
```

This is actually a BASIC statement line. The five-digit number on the left—00010—is the statement number. Every BASIC statement has to have one. The number can't contain more than five digits, and it has to be different from every other statement number in the program. Zeros on the left don't make a number unique—00010 and 10 are considered to be the same.

The statement number is always the first thing on the statement line; then the statement itself begins. In the example we showed, LET A = 3 is the BASIC statement.

The phrase following the BASIC statement LET A = 3 is a comment and has no effect on what your program is doing. The word REM can be used to put comments on most BASIC statements. A comment serves as a reminder when you look at your program, of what the statement is doing. Your comment or

remark can say anything you want it to say, as long as it fits on one line with the statement.

Getting Data into the Computer

There are several different ways that you can get a piece of information into the computer.

The Assignment Statement

Earlier, we showed the statement:

```
LET A = 3 REM ASSIGN VALUE TO A
```

This statement is an assignment statement—it puts the value 3 into the computer and names that value A. In the statement, the number 3 is a *constant* —it can never have any value other than 3. The name A, however, is a *variable* —it can have any number of values, but only one at a time. Right now it has the value 3, but that value can be changed by other statements during a program. For example, the assignment statement:

```
LET A = 4
```

gives the value 4 to A.

You can also use the assignment statement to give a value to several different variables. For example:

```
LET A,B,C = 15
```

Here, A, B, and C are all given the value 15.

When you write an assignment statement, you don't have to include the word LET. The statements we've shown could instead be written:

```
A = 3
A = 4
A,B,C = 15
```

There are other things you can do with the assignment statement, such as performing calculations on data. We'll be talking more about them later. Right now, let's look at some more ways to get data into the computer.

The READ, DATA, and RESTORE Statements

To assign ten numbers, say the numbers 1 through 10, to ten variables, you could use 10 assignment statements:

```
100    LET A = 1 REM ASSIGN VALUE TO A
200    LET B = 2
300    LET C = 3
       etc.
```

Using ten assignment statements can be tedious; another way to enter these numbers is with one DATA statement:

```
200    DATA     1,2,3,4,5,6,7,8,9,10
```

The DATA statement causes numbers to be put into a data table in the computer. You can use one or several DATA statements to do this. The following set of statements would have the same effect as the single DATA statement above:

```
200   DATA    1,2,3
201   DATA    4,5,6
202   DATA    7,8,9,10
```

Once the numbers are in the table, you use the READ statement to assign them to variables. Here's an example:

```
200   DATA    1,2,3,4,5,6,7,8,9,10
210   READ    A,B,C,D,E,F,G,H,I,J
```

The READ statement reads the values in the data table and assigns them (in order) to the variables—the number 1 to the variable A, 2 to B, 3 to C, etc.

You don't have to read all of the values in the data table at one time. For example:

```
200   DATA    1,2,3,4,5,6,7,8,9,10
300   READ    A,B,C
```

will result in the first three values in the table being assigned to A, B, and C, respectively. Another READ statement will take up where the last left off. Thus:

```
310   READ    D,E,F,G
```

will read the values 4, 5, 6, and 7 into D, E, F, and G.

You must be careful, though, not to try to read more values than the table contains. For example, still another READ statement:

```
320   READ    H,I,J,K
```

would be requesting values for four variables when only three numbers (8, 9, and 10) are left in the table.

If you want to, you can use the values in the data table more than once. At any point in your program, you can ask that values be read from the beginning of the table again, even if you haven't read all the values in the table yet. To go back to the beginning of the table, use the RESTORE statement:

```
100   RESTORE
```

Say you want to read the values 1, 2, and 3 into three variables A, B, and C, in that order. Later on in the program, you want to read the same values into D, E, and F. These statements will do just that:

```
 30  DATA  1,2,3,4,5,6
       .
       .
 50  READ  A,B,C
       .
       .
100  RESTORE   REM READ FROM START OF DATA TABLE
105  READ  D,E,F
```

It's important to remember, when using READ and DATA statements, that no matter how many DATA statements you put in your program, only *one* data table is created, before any READ statement is executed. The table is created from all the DATA statements in your program, regardless of where they appear—at the beginning, at the end, or scattered throughout. Each of the three sets of statements here have the same effect:

```
200   DATA  1,2,3
210   DATA  4,5,6
220   READ  H,I,J,K,L,M
```

```
200   READ  H,I,J,K,L,M
210   DATA  1,2,3
220   DATA  4,5,6
```

```
200   DATA  1,2,3
210   READ  H,I,J,K,L,M
220   DATA  4,5,6
```

The INPUT Statement

Both the assignment statement and the DATA statement use constants—unchanging data items that are part of your program—to assign values to variables. You have to know, at the time you're writing your program, what numbers you want to assign.

The INPUT statement allows a little more flexibility. This statement names the variables that are to receive values, but allows you to wait until you are running your program to actually supply the values. For example:

```
50    INPUT  X,Y,Z
```

means that you will supply values for X, Y, and Z when your program is run. You'll know when it's time to supply the values because a question mark will be printed at the terminal:

```
?
```

When you see this, you should enter your values, one for each variable in the INPUT statement—in this case, three. The numbers are entered all on one line, separated by commas. Thus, when you've typed the information, the line should look like this:

```
?
185, 205, 191
```

By entering these numbers, you've assigned 185 to X, 205 to Y, and 191 to Z.

You have to be certain, when typing your input line, to enter exactly the same number of values as there are variables in the INPUT statement in your program. If you try to enter too many or too few, the line won't be accepted and you'll have to try again.

Using the Comma as a Continuation Character

If your input is too long to fit on one line, a comma entered after the last item on the line acts as a continuation character. Data on the next line is read as a continuation of the input from the preceding line. For example, the input for this statement:

```
100   INPUT A,B,C,D,E,F,G,H,I,J
```

could be entered like this:

```
?
53095, 10033, 76332, 41329, 12498, 29875, 35608,
81421, 73190, 30042
```

The comma appearing after the last item on a line informs the computer that additional data is to be read from the next line.

Using the Slash (/) to Delimit Input Data

The slash can be used to end an input line and, in effect, skip the remaining variables in the INPUT statement. For example, if you were using statement 100 above, but wished to enter only five values, you could enter:

```
?
53095, 10033, 76332, 41329, 12498/
```

The computer assigns the values to A, B, C, D, and E, and ignores F, G, H, I, and J.

Using Consecutive Commas as a Null Delimiter

Two consecutive commas appearing in the data are treated as a null entry; that is, the corresponding variable in the INPUT statement is unchanged. For example, if you did not wish to enter an item for B in this statement:

```
70   INPUT A,B,C,D
```

you could enter this data:

```
?
4,,3,9
```

The computer assigns the value 4 to A, leaves B unchanged, assigns the value 3 to C, and assigns the value 9 to D.

Prompting Your Input

Since a lot of time can elapse between the time you write a program and the time you run it, you may have difficulty remembering exactly how many values you have to enter. This is especially true when your program contains more than one INPUT statement. Then you have to keep track of which one comes first.

You can have your program keep track for you by reminding you what has to be entered. All you have to do is put a PRINT statement immediately before the INPUT statement in your program. For example, if your program is averaging bowling scores, you could use these statements:

```
45   PRINT 'ENTER THREE BOWLING SCORES'
50   INPUT  X,Y,Z
```

Then, at run time, instead of just a question mark appearing when it's time to enter your values, these lines will be printed:

```
ENTER THREE BOWLING SCORES
?
```

When you've entered your numbers, the lines will look like this:

```
ENTER THREE BOWLING SCORES
?
185, 205, 191
```

You can put any reminder message that you want in the PRINT statement, as long as you put quotation marks around it. You can use single or double quotation marks—this PRINT statement is effectively the same as the one shown above:

```
45   PRINT "ENTER THREE BOWLING SCORES"
```

Of course, you also have to remember that the PRINT statement has to fit entirely on one line. If your message is so long that it doesn't fit, you might consider using several PRINT statements in a row:

```
45  PRINT 'TYPE IN TWELVE AVERAGE TEMPERATURES'
46  PRINT 'FOR THE MONTHS JANUARY TO DECEMBER'
50  INPUT  M,N,O,P,Q,R,S,T,U,V,W,X
```

Bear in mind that you should use this method only when a long message is absolutely essential. As a general rule, you should keep your messages as brief as possible.

Buffered-Ahead Terminal Input

Another facility to allow entry of data values for more than one INPUT statement is called "buffered-ahead terminal input." This facility enables you to enter, on a single line, data values to satisfy more than one INPUT request. Each data value group entered is equivalent to a single line of input data, and is separated by a semicolon. For example, you could have written PRINT and INPUT statements for the system to prompt you as follows:

```
ENTER BOWLING SCORES - TEAM A
?
185,201,191
ENTER BLOWING SCORES - TEAM B
?
133,188,244
```

Using the buffered-ahead terminal input facility, the above could be entered as:

```
ENTER BOWLING SCORES - TEAM A
?
185,201,191;133,188,244
ENTER BOWLING SCORES - TEAM B
```

As you can see from the second example, the second question mark does not appear because the second input request was satisfied with the data values following the first question mark.

Remember, with the INPUT statement, the number of data values you enter should not exceed the width of the terminal input line.

Different Kinds of Data

So far we've learned that data can be entered into the computer in two forms: as constants within a program, or as input during program execution. Most of the data we've shown in our examples has consisted of numeric values. This *arithmetic data* is one of the two types of data that BASIC can handle. The other is *character data*. We've seen an example of character data in the PRINT statement used to prompt input—the prompting message enclosed in quotes. Now let's look at each data type in detail.

Arithmetic Data

VS BASIC allows several forms of arithmetic data—*integer, fixed-point,* and *floating-point. Integer* data is the form we've been using in our exampies. It consists of whole numbers, positive or negative:

```
12
+356
-4
24657
```

Notice that you can't use commas in an integer; in other words, the last number can't be expressed as:

```
24,657
```

Fixed-point data consists of a number containing a decimal point. Like integers, fixed-point numbers can be positive or negative:

```
1.59
.0054
34.0
+2.1212
-48.
```

The *floating-point* form of arithmetic data gives you a way of concisely expressing extremely large or extremely small values. (Unless you're working with math or science problems, you'll probably never need it.) In science, a very large value is often expressed as a small number multiplied by 10 raised to a power (for example, 3 million—or 3000000—is expressed as 3 times 10^6). Similarly, a very small value is expressed as a large number multiplied by 10 raised to a negative power (for example, .000003 is expressed as 3 times 10^{-6}).

A floating-point number looks like an integer or a fixed-point number, followed by the letter E, followed by an *exponent*—the power of 10. The number 3000000—or 3 times 10^6—is written in floating point as 3E+6; or it can be written in any of these equivalent ways:

```
3.0E+6
3E6
+3E06
```

Likewise, the number .000003—or 3 times 10^{-6}—can be written in floating point as 3E-06. Some other ways of expressing it are:

```
3.0E-6
3E-06
+3E-6
```

Here are some things to remember about floating-point numbers:

- The number preceding the E can be either integer or fixed-point format—that is, it may or may not contain a decimal point.
- A positive exponent can be preceded by a plus sign, or it can be written with no sign.
- A negative exponent must be preceded by a minus sign.
- The largest absolute value that can be expressed is approximately 10^{75}.
- The smallest absolute value that can be expressed is approximately 10^{-78}.
- The exponent can be expressed as one or two digits.

There's a limit to the number of digits that you can have in integer or fixed-point numbers as well. BASIC allows numbers to be expressed in *short-form* or *long-form*. You can specify the precision you want your program to have in an OPTION statement, or in the command you use to execute your program. If you use OPTION statement, it must be the first statement in your program.

Short-form arithmetic data can contain as many as seven significant digits; long form data as many as fifteen. If a number in your program contains more digits than allowed for the precision you have chosen, only the first seven or fifteen significant digits are used.

When dealing with whole numbers, one can always expect seven places of accuracy. However, when dealing with fractions, such accuracy cannot always be obtained. This is because of the way VS BASIC represents numbers internally during execution. Numbers are converted to hexadecimal for computation, and there are some decimal fractions which do not have an exact hexadecimal equivalent, that is, there are repeating hexadecimal numbers.

These maximum precisions apply to the integer or fixed-point portion of a floating-point data item, as well as to integers and fixed-point numbers.

Character Data

The other form of data, as we said, is character data. To the computer, it is merely a string of characters—letters, digits, punctuation marks, etc., strung together in any combination. Such a *character string* can be a single character, or it can be as many as 255 characters.

Depending on the way in which it is used, character data may have to be enclosed within quotation marks—either single or double. For example, quotation marks are needed for character data used in an assignment statement (LET statement). When character data is used in a DATA or INPUT statement, quotation marks should be used when the data contains any of the following:

- commas or slashes (or semicolon-INPUT only)
- leading or trailing blanks
- leading or trailing single or double quotation marks
- an initial integer immediately followed by an asterisk

It is never an error to use the enclosing quotation marks, so if you are not sure whether or not the quotation marks are needed in any given case, you can always use them.

These quotation marks are often referred to as delimiters, because they show the *limits*—the beginning and end—of a character data item.

The *length* of a character data item is the number of characters in it, not counting the delimiting quotation marks. But everything between the delimiters must be counted, even blanks. Here are some character constants and their lengths:

Constant	Length
'ABCD'	4
"TWO AND TWO"	11
'1234567'	7

Notice that the last constant, except for the quotation marks, looks just like an integer. Be careful not to confuse the two. An integer has an arithmetic value and, as you'll see later, can be used in arithmetic operations. A character string, even when it is made up of all numeric digits, cannot be added to or subtracted from. It is only a string of digits.

Uses of Character Data

You might want to use character data for a number of reasons. You've already seen it used to create messages. You can also use it to put a heading on a report, or to label your results in a program doing arithmetic calculations. You may want to read some names and account numbers, perhaps test them (you'll learn how to do this later), and print them. An account number or a serial number is a good example of a character data item that is all digits.

Some Rules about Delimiters

We've already said that a character data item can be *bounded by*—enclosed in—single or double quotation marks. While you have a choice, you cannot use both types to delimit a single character data item. An item that starts with a single quote must end with a single quote; likewise for double quotes. This has special significance when you want to use quotation marks as part of your character data item.

Let's say you want to use this character value:

```
IT IS THE DOG'S BONE
```

We haven't shown any delimiting quotes here yet; the single quote within the constant is needed as a punctuation mark. If you tried to use single quotes to delimit this value:

```
'IT IS THE DOG'S BONE'
```

the computer would recognize the punctuation mark as a delimiter, and you would wind up with the value:

```
IT IS THE DOG
```

The remaining characters—S BONE'—would cause an error.

One way of avoiding this problem, in this instance, is to use double quotes to delimit the constant:

```
"IT IS THE DOG'S BONE"
```

Since the opening delimiter is a double quote, only a double quote is recognized as the closing delimiter, and the entire value is accepted. If for some reason you're unable to use double quotes, BASIC allows this convention to solve the problem:

```
'IT IS THE DOG''S BONE'
```

Using two single quotes in succession here indicates that you want one single quote to be part of your character value, and that more characters may follow. This way, the entire constant is recognized.

All of what we've said applies when double quotes are needed as part of a constant:

```
A "FLOATING-POINT" ITEM
```

This value can be expressed in either of these ways:

```
'A "FLOATING-POINT" ITEM'
"A ""FLOATING-POINT"" ITEM"
```

To restate the convention—two consecutive occurrences of the delimiting character in a character constant are interpreted as one occurrence of that character within the character value. When this happens, by the way, the double occurrence of the character counts as only one character in determining the length of the constant. The constant just shown has a length of 18, regardless of the way it is expressed.

Rules for Forming Variables

Now that you know about the different kinds of data you can use in your program, you'll have to follow some rules to form the variables that data can be assigned to.

Arithmetic Variables

You may have noticed that all of the variables we've used in our examples involving arithmetic data were single letters of the alphabet—A, B, C, and so forth. Actually, BASIC allows *arithmetic variables* to be any of the following:

- A single letter of the *extended* alphabet
- A single letter of the *extended* alphabet followed by one of the digits 0 through 9

By the *extended* alphabet, we mean any of the 26 letters of the English alphabet, plus 3 additional characters: the commercial "at" sign (@), the number or pound sign (#), and the currency or dollar sign ($). These three characters are often referred to as the *alphabet extenders.*

Here are a few examples of valid arithmetic variables:

```
X
B3
#
$1
Q0
```

Character Variables

Character variables must be different from arithmetic variables. They are formed from a single letter of the extended alphabet, followed by a currency symbol ($). For example:

```
A$
@$
$$
```

Just as a character data item has a certain length, a character variable, too, has a length associated with it. The length of a character variable is the

number of characters that can be assigned to it. Without any action on your part, all character variables are given a length of 18. If you want any or all of your character variables to have a different length, you can define these lengths by including a DIM statement in your program:

```
200    DIM B$6, C$12
```

Here, we've said that the character variables B$ and C$ are to have lengths of 6 and 12, respectively. This DIM statement must appear before any other references to B$ or C$ in your program. If there are no other DIM statements in the program, all other character variables used have a length of 18.

You can put as many character variables in a DIM statement as you wish, provided that a length is specified after each variable, and the entries are separated by commas and will fit on one line. If you want to specify more variables than will fit on one line, you can use additional DIM statements. (The DIM statement, by the way, has another use that we'll cover later.)

Now let's look at the effect that the length of a variable has when character values are assigned to it:

```
30     LET A$ = 'DATA STRING'
```

Since there is no DIM statement specifying the length of A$, its length is 18. Any value it is assigned must be 18 characters long. Since the character value being assigned here is only 11 characters long, 7 blanks are added to the end of it to fill out the length of 18. To say this another way, it is *padded* on the right with 7 blanks.

If this DIM statement were to be in the program:

```
100    DIM A$8
```

the character value DATA STRING would be *truncated* on the right—only the first eight characters would be assigned to conform to A$'s length of eight. Now the value of A$ would be:

```
DATA STR
```

Mixing Arithmetic and Character Data

All of the methods for getting arithmetic data into the computer—the assignment statement, the DATA and READ statements, and the INPUT statement—can be used for character data as well. In fact, character and arithmetic variables can be interspersed in the same READ or INPUT statement:

```
30     INPUT A,B,C,M$
```

Three arithmetic values and one character value must be supplied, in that order, when the program is run. For example:

```
?
77, 85, 83, "DAILY TEMPERATURES"
```

Similarly:

```
10     DATA 2507, JOHN DOE, 33,"2/76"
20     READ A, B$, X, X$
```

Note that in both these cases, arithmetic and character data must be arranged in the same order as the arithmetic and character variables. The following statements would cause execution of your program to fail:

```
10     DATA 2507, JOHN DOE
20     READ B$, A
```

Expressions and Calculations

In an assignment statement, only a variable can appear to the left of the equal sign. Both variables and constants can appear to the right of the equal sign. In fact, constants and/or variables can be combined with operators. Here are some examples:

```
10    LET A = B
20    LET H = 300 + 278 + 312 + 218
40    J = 71 + 14 - 34 / X
50    LET X = X + 2
```

The last example adds 2 to X. If you know algebra, you will see a difference between the LET statement and an algebraic equation. A LET statement assigns a value to a variable and does not imply that the items on the left and right of the equal sign are mathematically equal. This LET statement says that 2 is to be added to the value of X and the new value is to be assigned to X. If the old value of X was 10, it becomes 12.

That part of the assignment statement to the right of the equal sign is called an *expression.* The expression specifies the value to be assigned to the variable on the left of the equal sign. An expression can be very simple, involving no calculations, or it can be quite complicated, involving many variables and arithmetic operations. Some examples of arithmetic expressions are:

```
3E6
71 + 14 - 34
A1
-6.4
X + Y + Z
X3 / (-6)
```

The symbols used in expressions to specify mathematical operations are called arithmetic *operators.* There are two kinds of arithmetic operators, *unary* and *binary.*

The two unary operators are the symbols + and -. The following examples show how unary operators are used:

```
+5
-10
-C1
```

The first example states that 5 has a positive value. (It does not mean that 5 is to be added to some other number.) The second example states that 10 has a negative value. The third example illustrates a negative variable. When a negative variable is encountered, the sign of the variable is reversed. That is, if C1 contains the value +5, the value of -C1 is -5; if C1 contains -10, the value of -C1 is +10.

There are five binary operators:

Symbol	Meaning	Example
+	addition	5 + 3
-	subtraction	5 - 3
*	multiplication	5 * 3
/	division	5 / 3
↑ or **	exponentiation	5 ** 3 (5 raised to the third power: 5 * 5 * 5)

Evaluation of Expressions

Arithmetic expressions are evaluated according to the priorities of the operators involved. Operations with higher priorities are performed first. Those at the same priority level are performed as they are encountered, from left to right.

The order of priority is as follows:

1. Exponentiation is performed first; thus it is said to have the highest priority.
2. Unary operations have the second priority.
3. Multiplication and division have the third priority.
4. Finally, addition and subtraction have the lowest priority.

Let's look at this example (assume A = 4, B = 10, and C = 5):

```
-A ** 2 + B / C * 2.5
```

The evaluation process follows this sequence:

1. A ** 2 is evaluated first. (Result = 16)
2. The unary minus sign is applied to the result of A ** 2 (that is, the sign of A ** 2 is reversed). (Result = -16)
3. B is divided by C. (Result = 2)
4. The result of B / C is multiplied by 2.5. (Result = 5)
5. Finally, the result of item 4 is added to the result of item 2. (Result = -11)

Parentheses may be used in an expression to alter the order in which the expression is evaluated by the computer. Any part of an expression enclosed in parentheses is evaluated before any other part of the expression. For example, the expression

```
A - B / C
```

is evaluated as follows: divide B by C and then subtract the result from A. Spacing is ignored; that is, even if the expression were written:

```
A-B / C
```

the division operation would be performed before the subtraction operation. Assuming values of 15, 10, and 5 for A, B, and C, the result would be 13. However, by using parentheses, the order of evaluation can be altered:

```
(A - B) / C
```

Now the expression is evaluated as follows: first subtract B from A and then divide the result by C. Assuming the same values for A, B, and C, the result here is 1.

Note that A - (B / C) is the same as A - B / C.

Thus, the use of parentheses in expressions is similar to the use of parentheses in algebra; that is, parentheses group operations and indicate which operations should be performed first. As a general rule, use parentheses whenever you are in doubt about the way an expression would otherwise be evaluated.

As a further illustration, evaluation of (2**3)**2 or 2**3**2 gives 64, whereas, evaluation of 2**(3**2) gives 512.

To summarize arithmetic expressions, there are two unary operators: +(plus) and -(minus). There are five binary operators. In order of priority, they are: ** (exponentiation), * (multiplication) and / (division), and +(addition) and - (subtraction). An expression can consist of a constant, a variable, a function reference, a subexpression contained in parentheses, or any combination of these separated by suitable operators. (In addition, an expression can contain an array reference, which we have not yet discussed.)

Character Expressions

A character expression can consist of a character constant, a character variable, a character-valued function reference, a character array member—not yet discussed—or any combination of these, separated by binary character operators and parentheses.

The following are examples of character expressions:

```
'NEW YORK"
A$
'ABC' || 'DEFG'
```

The concatenation operator, symbol | |, causes two character expressions to be joined together. In the last example, the character string DEFG is concatenated with ABC forming the character string ABCDEFG. The concatenation operator can also be written using the alphabetic equivalent symbol .CAT. instead of the symbol | |.

When two or more character strings are concatenated the length of the resulting string is the sum of the lengths of the individual strings.

For example:

```
"1234" .CAT. "5"
```

The length of the resulting character string is 5.

Intrinsic Functions

Functions are available in VS BASIC which simplify mathematical and character string operations. They include mathematical functions, such as sine, square root, and natural logarithms; and character functions, which perform such actions as extracting a portion of a character string and providing the time of day. These functions are called *intrinsic functions.* Conceptually, you can think of an intrinsic function as a group of instructions that performs a calculation. For example, the sine function consists of instructions to obtain the sine of an angle. To save you the trouble of having

to write these instructions into your program, these functions are made available as part of VS BASIC.

To use a function, you call it by its name. For example, to perform the sine operation, you refer to SIN. The name given to the square root function is SQR, to the natural logarithm, LOG, and to the time of day function, CLK. A complete list of intrinsic functions available in VS BASIC is given in Part II of this publication. Some examples of their use are shown below:

```
 70  LET V = SIN(Y)
 80  LET Z = 1 + SQR(X**3)
 90  W = 1 - SQR(COS(A))
100  T$ = CLK
```

The quantity in parentheses immediately following the name of the function is an argument (for example, X**3 in statement 80). An *argument* is merely an expression representing a value that the function is to act upon. The expression can be as simple or as complicated as any of the expressions we've encountered so far, and it is evaluated according to the same rules. Thus, in the second example, if the value of X is 4, then the value of X**3 is 64, and the value of SQR(X**3)—or the square root of 64—is 8. The third example shows nested function references. In such cases, the expression within the innermost set of parentheses is evaluated first; the expression within the next innermost set is evaluated next, and so on until the outermost level is reached. Thus, the cosine of A is found first and the square root of that cosine value is found next. The last example shows the CLK function, which has no argument,

Two functions allow you to locate or refer to portions of character strings, or *substrings.* With the IDX function you can locate a group of characters within a character string; with the STR function, you can extract or display a group. The following example illustrates the use of these functions:

```
00   DIM S$1,C$2
10   L$ = 'HTNSBRIMODXGYAVFLKQCZUWEJP'
20   S$ = 'A'
30   N = IDX(L$,S$)
40   C$ = STR(L$,5,2)
```

Statement 30 finds the position of the character 'A' in a scrambled alphabet. N will receive the arithmetic value 14 because the string 'A' (in variable S$) is at position 14 of the alphabetic string (variable L$). Statement 40 extracts two consecutive letters starting at position 5 of the scrambled alphabetic assigns then to C$. Thus, C$ will be set to 'BR', which occupies positions 5 and 6 of string L$.

When you use a function in a statement, you are making a *function reference.*

Internal Constants

An *internal constant* is an arithmetic constant having a predefined value. Internal constants, such as the value of π, the square root of 2, and the number of centimeters to an inch, help to simplify mathematical operations. By specifying the name of an internal constant in a BASIC statement, you make it unnecessary to define the value yourself. A complete list of internal constants available in VS BASIC is given in Part II of this publication. Some examples are shown below:

```
30   A = &PI * R ** 2
40   B = &INCM * I
```

Assuming variable R is a radius, statement 30 calculates the area of a circle by multiplying the square of the radius by the internal constant &PI, which contains the value of π. Assuming variable I represents inches, statement 40 converts inches to centimeters by multiplying I by the internal constant &INCM, which contains the number of centimeters to an inch.

Pseudo Variables

A *pseudo variable* is a special use of an intrinsic function as a variable. It can be used to receive a value in any place that an ordinary variable can be used, for example, in an INPUT statement, or on the left side of an equal sign. There is only one pseudo variable in VS BASIC. Its name, data type, and format are the same as those of the STR intrinsic function. Here is an example:

```
00   DIM A$10
10   A$ = 'ABCDEFGHIJ'
20   STR(A$,7,2) = 'XX'
```

The pseudo variable in statement 20, which is written to the left of the equal sign, indicates where the character string 'XX' should be assigned in A$. This statement assigns the string to the two character positions starting at position 7 of string A$. The result will be to give A$ the value ABCDEFXXIJ.

Internal Variables

Internal variables are used by VS BASIC to record information about your program as it executes. For example, during program execution, the internal variable &LINE contains the program line number of the VS BASIC statement currently being executed. You can find out the value assigned to an internal variable by specifying the name of the variable in a VS BASIC statement.

Internal variables are sometimes referred to as "read-only" internal variables because, while you can "read" them (that is, find out what the value of the variable is), you cannot write them (that is, assign new values to them that would write over the value VS BASIC has assigned to them).

One use of the internal variable in a VS BASIC program is to help identify and locate those errors that can occur during program execution.

A list of internal variables and their uses is given in Part II of this publication.

Getting Data Out Using the PRINT Statement

The simplest way to get data out of the computer is by using the PRINT statement. The PRINT statement is the output counterpart of the INPUT statement.

As you recall, the INPUT statement enters data in the following manner:

```
300  INPUT A, B, C, D, E, F, G
```

The set of variables in the statement is known as a *list.*

The PRINT statement prints the values of an output list in the same manner as the INPUT statement enters values for an input list:

```
350  PRINT A, B, C, D, E, F, G
```

The PRINT statement can print both constants and variables, and can be used to print character and arithmetic data on the same output line. Here are examples:

```
400  PRINT A, B, C, 5, 6, 7
```

This statement prints the values of the arithmetic variables A, B, and C, followed by the arithmetic constants 5, 6, and 7.

```
450  PRINT 'THE VALUE OF A IS:', A
```

This statement prints the character constant enclosed in single quotation marks, followed by the value of the arithmetic variable A. If A had the value 56, the printed line would look like this:

```
THE VALUE OF A IS: 56
```

A PRINT statement can also include an expression made up of variables and operators. For example:

```
500  PRINT A, B, A*B
```

The values of A and B will be printed, along with the product of both values.

In general, each PRINT statement causes the computer to begin a new line. Thus, a useful technique for entering blank lines between lines of output is to write a PRINT statement with no list; the computer prints a blank line.

Horizontally, the output page is divided into *full print zones,* each zone having 18 print positions. Assuming that the left-hand margin is set at position 1 on the terminal, the zones would begin in positions 1, 19, 37, 55, 73, etc. A comma is used in the PRINT statement as a signal to the computer to move across the page to the next full print zone. For example, if we use the following statement:

```
400  PRINT   A, 5, 'PING ', 'PONG'
```

the computer would start at the left edge of the page and print the value of the variable A. Then it would skip over to print position 19 and print the value 5. The character constant PINGb (the b indicates a blank character) would be printed beginning in print position 37, followed by the character constant PONG, beginning in print position 55.

If the variable A contained the value —12345, the printed line would look like this:

Print Position			
1	**19**	**37**	**55**
-12345	b5	PINGb	PONG

Note that a positive number is preceded by a blank, and a negative number is preceded by a minus sign.

It is possible to increase the number of print zones on a line. A semicolon or a *null delimiter* (a blank or no separation at all between data items) in the PRINT statement indicates to the computer to use a *packed print zone* rather than a full print zone. The size of the packed zone for arithmetic data is determined by the length of the field to be printed. An arithmetic value between 2 and 4 characters in length has a packed zone of 6 print positions. A value between 5 and 7 characters has a zone of 9 positions. Longer values have longer print zones; Table 4 in Part II of this publication summarizes print zones.

Note that the minimum arithmetic value is considered to be 2 characters rather than 1, because the first print position is always reserved for the arithmetic sign, either a blank for a positive number, or a minus for a negative one. Thus, the value 5 in the example above would be considered a two-character value, having a packed zone of 6 positions. The other arithmetic value in the example, -12345, has a zone of 9 positions.

The size of the packed zone for a character variable or expression is the length of the data item minus any trailing blanks. For a character constant, the size of the packed print zone equals the length of the character string that is enclosed in quotes.

If PRINT statement 400 were rewritten to specify packed print zones:

```
400  PRINT A; 5 'PING ';'PONG'
```

it would cause this line to be printed:

```
Print
Position
1          10     16       21
-12345     ъ5     PINGъ    PONG
```

A Simple Program

At this point, let's put together some of the statements we've been using, and construct a simple program.

Suppose we average those bowling scores we talked about earlier. First, we have to enter the scores (six of them, in this case), so that the computer can act on them. We can use an INPUT statement to do that. Then we have to tell the computer to add all the scores and then divide them by 6. We can use one or two LET assignment statements for that. Then we print the average so we can see the result. The PRINT statement can be used for that.

Here's our program:

```
 50  PRINT 'ENTER SIX BOWLING SCORES'
100  INPUT A, B, C, D, E, F
200  LET X = A + B + C + D + E + F
300  Y = X / 6 REM COMPUTE AVERAGE
400  PRINT 'THE BOWLING AVERAGE IS', Y
500  END
```

Statement 500, the END statement, tells the computer that the end of the program has been reached. Every program must have an END statement, and it should be the highest numbered statement in the program. If any statements appear after the END statement, they are ignored.

When we run the program, the computer will signal us that it is ready to accept the input by printing a question mark at the terminal. We respond by entering the six numbers to be entered into the variables A through F. When the computer is finished processing, it will print out the result using the format specified in PRINT statement 400. Here's what the output would look like at the terminal:

```
ENTER SIX BOWLING SCORES
?
175, 173, 181, 184, 181, 172
THE BOWLING AVERAGE IS 177.6667
```

Comments In Your Program

You will probably find it useful to put comments throughout your program, particularly if it is a complex one, or one that you do not use often enough to remember exactly how it is run. BASIC provides several ways to include comments in your program.

- Include comments on BASIC statements using the REM keyword.
- Include comments on certain BASIC statements without using the REM keyword
- Include comments using the REM statement

Earlier examples showed how to include a comment on a BASIC statement using the REM keyword. The REM keyword can be used with all BASIC statements except the DATA statement (described earlier) and the Image statement (to be discussed later). Recall the examples:

```
010   LET A=100 REM ASSIGN THE VALUE 100 TO A
020   RESTORE REM READ FROM START OF DATA TABLE
```

For BASIC statements consisting solely of a keyword, BASIC allows comments to be appended with or without the REM keyword. Statements such as RESTORE, END, and other statements not yet described (STOP, PAUSE, FNEND, and RETURN when used with GOSUB) do not require the REM keyword. The second example above could have been written:

```
020   RESTORE READ FROM START OF DATA TABLE
```

BASIC also provides a special statement used specifically to include remarks in your program. It is the REM statement. Like other BASIC statements, it is preceded by a statement number. The following are examples:

```
40    REM THIS  A COMMENT
10    REM USE THIS PROGRAM TO COMPUTE BOWLING AVERAGES
```

In the last example, the REM statement is used to describe the purpose of the program, computing bowling averages.

Testing and Controlling Program Data

Now that we can write simple programs, let us explore different ways to make our programs more efficient.

Suppose you wanted to print out each number from 1 to 50 together with its square. You could do it simply enough by writing the following statements:

```
10  PRINT 1, 1**2
20  PRINT 2, 2**2
30  PRINT 3, 3**2
40  PRINT 4, 4**2
```

and so on, ending with

```
490 PRINT 49, 49**2
500 PRINT 50, 50**2
510 END
```

Although this technique works correctly, it is very time consuming and tedious. In writing out a number and its square for all numbers from 1 to 50, what we are really doing is performing the same operation repeatedly, but using different numbers each time. Calculations that are to be repeated can generally be done efficiently by a simple programming device known as a *loop.*

Loops

Here's a concise method of performing the same operations shown above:

```
10  LET X = 1
20  PRINT X, X**2
30  LET X = X +  1
40  GO TO 20
50  END
```

Here we have created a loop in statements 20 through 40. When the program is run, the PRINT statement will be executed once each time the value of X increases by 1. The statement that makes the loop possible is the GO TO statement. It alters the normal sequence of execution by directing the computer to execute a different statement. It does this by referring to the number of that statement. The statement GO TO 20 directs the computer back to statement 20, which prints the value of X and its square. Statement 30 then increases the value of X by 1, and statement 40 is executed again, "branching" the program back to statement 20.

There is one problem with the loop we have shown here: there is no provision for ending the loop. Consequently, not only will we get results for values from 1 to 50, but also for 51, 52, and so on, unless we take some action to stop the execution. In this program, we want the loop to end after we reach the value 50, or, put another way, we want the loop to continue as long as X is less than or equal to 50. To provide this action, we should build into the loop a test for some condition, so that when the condition is met, the loop would end automatically.

Using the IF Statement

An IF statement says it quite concisely:

```
IF X <= 50 GOTO 20
```

This IF statement says that if X is less than (<) or equal to (=) the value 50, the program is to branch to statement 20. Here we have incorporated the GOTO statement into the IF statement. Let's put this new statement into the program and see what happens.

```
10  LET X = 1
20  PRINT X, X**2
30  LET X = X + 1
40  IF X <= 50 GOTO 20
50  END
```

As long as X satisfies the condition "X less than or equal to 50," execution will loop back to the PRINT statement. However, when X no longer satisfies the condition—when X is greater than 50—the loop will end automatically and the execution will "fall through" the IF statement to the next statement, which in this case is an END statement signifying the end of the program.

The IF statement has many applications, some of which can be quite sophisticated, depending on the condition tested in the statement. For example, conditions such as the following can be tested:

```
160 IF A = 0 GOTO 60
170 IF A = 0 THEN 60 ELSE 70
180 IF B - X / Y < Z**2 GOTO 80
190 IF Z>Y THEN PRINT Z ELSE PRINT Y
```

The first example is quite simple: if the value of the variable A is equal to 0, branch to statement number 60. The second statement tests the same condition as the first statement, but substitutes the word THEN for GOTO. In the IF statement, THEN and GOTO have exactly the same meaning. This IF statement also introduces the ELSE clause, which provides an alternative action if the condition is not met. This ELSE clause causes a branch to statement 70 whenever A does not equal zero. The third statement makes a test between two sets of expressions. The first expression evaluates the result of B-X/Y. The second expression evaluates the result of Z**2. If the result of the first expression is less than (<) the result of the second expression, then the program is to branch to statement 80.

The last example illustrates the use of an IF statement to perform actions other than branching. It states that if the value of Z is greater than (>) the value of Y, Z is to be printed, otherwise Y is to be printed.

These symbols <, >, and = are part of a set of operators called *relational operators.* Relational operators are used to test the relationship between two expressions. It is important to note that relational operators do not perform any arithmetic operations. They simply test whether a condition is or is not satisfied. For example, in the statement:

```
300  IF X=50 GOT 350
```

The equal sign does not mean that X is to be given the value 50, it tests whether the value already assigned to X equals 50. If a condition is satisfied (if X does equal 50 in this example), then the condition is considered "true." If a condition is not satisfied (if X does not equal 50), the condition is considered "false." Thus, a relational operator says that if the condition being tested is true, the action specified is taken; otherwise, the action is not taken. Reviewing this concept using the example IF A=0 GOTO 60, if the condition

is true (A does equal 0), then the branch to statement number 60 is made; otherwise the branch is not made. Instead, the program continues with the next statement in sequence.

The relational operators and their definition are:

Operator	Meaning
= or .EQ.	Equal to
<> or .NE. or ≠	Not equal to
> or .GT.	Greater than
< or .LT.	Less than
>= or .GE. or ≥	Greater than or equal to
<= or .LE. or ≤	Less than or equal to

The special characters and their alphabetic equivalents can be used interchangeably in relational expressions. Here are some examples:

```
IF A = B
IF "PRINT" < "PRIZE"
IF A$ .NE. D$
```

In the first example, a test is made between the values contained in the arithmetic variables A and B. The second example illustrates comparison of character data. For character data, a comparison is made according to the EBCDIC collating sequence of each character in corresponding positions in the constant. (Refer to Appendix B. "Collating Sequence of the VS BASIC Character Set.") In other words, the first character of one constant is compared to the first character of the other, the second compared to the second of the other, etc. In this example, the first three letters of the constants compare equal, but when the letter N is compared to Z, they compare unequal. The letter N, occurring before the letter Z in the alphabet, registers "less than" in the collating sequence. At this point, the condition tested would be met, that is, the character string PRINT is indeed less than PRIZE.

In the third example, character variables are compared. Let's assume that the variable A$ contains the value ON and the variable D$ contains ONLY. The first two characters match but when the letter L is compared to a blank, which is assumed for comparison purposes, they do not match. Thus, the result in this case would also be true, since the value of A$ is not equal to the value of D$. If, however A$ and D$ did contain matching strings, say both contained the characters ONLY, then the test results would be false—A$ and D$ would be equal, thereby not satisfying the condition of the test.

Logical Operators

The example IF A=B tests the relationship between two expressions. Suppose, however, that you wish to take action if *more* than one relationship is true. For example, suppose that not only must A equal B but also X must equal Y. You could make these comparisons this way:

```
40 IF A=B THEN 50 ELSE 60
50 IF X=Y THEN 100
60 ...
```

In statement 40, if the values of A and B are equal, then statement 50 is executed; otherwise, a branch is made around statement 50. In statement 50, if the values of X and Y are equal, then statement 100 is executed; otherwise, program execution continues with statement 60.

A more concise way of making this test is by using the And logical operator, written as & or .AND.:

```
40 IF A=B & X=Y THEN 100
60 ...
```

Statement 40 now says that if A equals B *and* X equals Y, then statement 100 is executed. If only one comparison, or neither comparison, is true, program execution continues with statement 60.

The IF statement can specify two logical operators:

Operator	Meaning
& or .AND.	And
| or .OR.	Or

The And operator states that *both* conditions of a test must be true in order for the entire expression to be true; the Or operator, that *either* condition must be true.

If you wanted to branch to statement 100 if either A equalled B or X equalled Y, you could write this statement:

```
40   IF A=B | X=Y THEN 100
```

Here are other examples of the And and Or operators:

```
70   IF C$> D$ | J$=K$ THEN 50 ELSE 150
80   IF A1≠A2 & J$ > "CAT" GOTO 300
90   IF A1.NE.A2 & J$ .GT. "CAT" GOTO 300
```

The first example tests an Or condition using character variables. It says that if either the value in the variable C$ is greater than that in D$ or the values in J$ and K$ are equal, then a branch is made to statement 50; if neither condition is true, then a branch is made to statement 150.

The second example tests an And condition using mixed variables. It says that if both the value in the arithmetic variable A1 is not equal to that in A2 and the value in the character variable J$ is greater than the character string CAT, then the program is to branch to statement 300; otherwise, program execution is to continue with the next sequential statement.

The third example illustrates the use of the alphabetic equivalents in place of the symbol in a BASIC statement. It tests the same conditions as the second example.

The Computed GOTO Statement

The computed GOTO statement is a version of the GOTO statement that gives you the ability to branch to different statements during various stages in a program.

A computed GOTO could look like this:

```
100  GO TO 30, 40, 50 ON J
```

A branch is made to statement 30, to statement 40, or to statement 50, based on the value contained in the variable J. J may contain a valid value of from 1 to 3. If J contains 1, a branch is made to the first statement shown in the list, statement number 30. If J contains 2, the branch is to the second statement, number 40. If J contains 3, the branch is to the third statement, number 50. If J contains any other value, program execution "falls through" to the statement following the computed GOTO statement.

The expression determining the branch to be made can be a simple variable, such as J above, or a more complicated expression, say, (A+B)/2. If such an

expression were used, its computed value would determine the branch to be made. Consider this example:

```
50   GOTO 200, 220, 100, 240 ON (A+B)/2
```

The expression (A+B)/2 is evaluated, and a branch is made to statement number 200, 220, 100, or 240, depending on whether the value is 1, 2, 3, or 4, respectively. Note also that the statement numbers shown in the list do not have to be specified in sequential order; that is, statement number 100 can be the third number in the list even though it is a lower number than the others.

More About Loops—Using FOR and NEXT Statements

A still more concise method of specifying a loop is by using the FOR and NEXT statements. For example, our program for finding and printing the square of the numbers from 1 through 50 could be further simplified to look like this:

```
10   FOR I = 1 TO 50
20   PRINT I, I**2
30   NEXT I
40   END
```

The FOR statement identifies the beginning of the loop; the NEXT statement identifies the end of it. In between is the statement, or sequence of statements—we only need one for this example—that will be executed repeatedly until the specification in the FOR statement has been satisfied.

In our example, the FOR statement specifies that the statement in the loop (the PRINT statement) will be executed repeatedly for successive values of I from 1 through 50 (an increment of 1 is added to I for each execution of the PRINT statement). When the value of I exceeds 50, execution of the loop is ended, and control is passed to the next logically executable statement following the NEXT statement. In this case, the following statement is an END statement denoting the end of the program. However, other statements could precede it, or the NEXT could be the last statement prior to the END.

The specification "I = 1 TO 50 " defines the set of values over which the loop will be executed. As we've seen, the set in our example is 1, 2, 3,..., 50. The increment is always 1 unless it is explicitly stated to be otherwise; for example:

```
10   FOR I = 1 TO 50 STEP 2
```

This FOR statement explicitly states an increment (or step) of 2. Thus, the statement(s) in the loop will be executed once for every odd value of I from 1 to 50 (that is, the range is 1, 3, 5,...49). When the value of I exceeds 50 (that is, when it reaches 51), execution of the loop will end. The value of I will be set back to 49 before the next logically executable statement is executed.

If you wanted to execute the loop once for every even value of I from 1 to 50 (that is, 2, 4, 6,...50), you would say the following:

```
10   FOR I = 2 TO 50 STEP 2
```

Again, when the value of I exceeds 50 (in this case, when it reaches 52), execution of the loop will end. The value of I will be set back to 50 when the next logically executable statement is executed.

As with expressions appearing in assignment statements and in the body of PRINT statements, the specifications in FOR statements can be quite complicated. For example, the following FOR statements are permitted:

```
30    FOR I = A TO B
40    FOR J = 8*M+Y TO A**3
50    FOR K = SQR(B) - C TO 550 STEP A/B**2
```

The first example states that the initial value of I is to be taken from the variable A and that the loop is to be executed repeatedly until the value exceeds the value of B. The second example states that the initial value of J is the value of the expression 8*M+Y, and the loop is to be executed until this value exceeds the value of A**3. The third example states that the initial value of K is to be the square root of B minus C; the loop is to be executed until the value 550 is exceeded, and each time through the loop the value of K is to be increased by the value of the expression A/B**2.

You can also use more than one set of FOR/NEXT statements together in a program, by "nesting" one loop within another one. Let's look at a program that computes compound interest and uses nested FOR loops in the process.

The mathematical formula to compute compound interest is:

$$A = P\left(1 + \frac{R}{100}\right)^{t}$$

where A is the amount to be calculated, P is the principal, R is the rate of interest, and t is the time period.

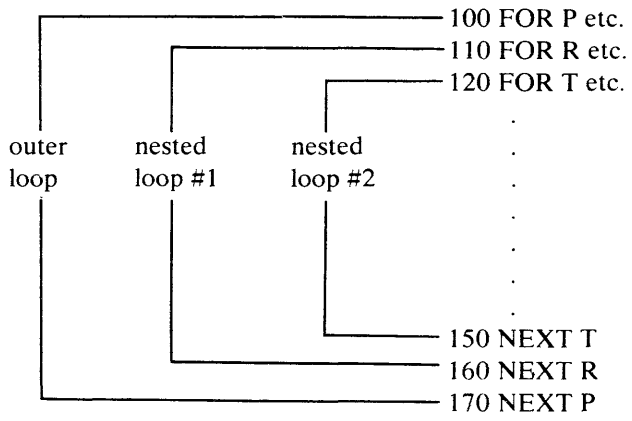
The program below shows how you can enter any amount as the principal (P), compute interest on it using interest rates from 1% to 20%, for each of ten years, and print out all the amounts—a total of 200 values.

```
 90   PRINT 'ENTER PRINCIPAL'
100   INPUT P
105   PRINT 'TIME', 'RATE', 'AMOUNT'
110   FOR T = 1 TO 10 REM VARY THE TIME
120   FOR R = 1 TO 20 REM VARY THE RATE
130   LET A = P*(1 + R/100)**T REM COMPUTE THE AMOUNT
140   PRINT T, R, A
150   NEXT R REM USE THE NEXT RATE
160   NEXT T REM USE THE NEXT TIME
170   END
```

Statement 130 duplicates, in VS BASIC terms, the compound interest formula. The FOR statement numbered 120 and the NEXT statement numbered 150 delimit one loop. The first time through the loop, the value of R, the rate variable, is set to 1. When NEXT R is reached, R is incremented by 1 and the statements are executed again with the new value of R. Each time through the loop the PRINT statement prints time, rate, and amount values. This process continues until R reaches 20 and the loop is ended.

However, this loop is enclosed, or *nested,* within the loop delimited by the FOR and NEXT statements numbered 110 and 160. This outer loop changes the value of T, the time variable, from 1 to 10. Each time the value of T changes, the inner loop cycles through 20 times changing the value of R. Since T changes value 10 times, the loop changing the value of R is executed 200 times. Each time, the PRINT statement prints new values.

A nested loop is one that is enclosed by another loop. That is, the FOR/NEXT statements of one loop occur between the FOR/NEXT statements of another loop, as illustrated:



Using Arrays

An array is a simple way to keep together data items that are related. For example, if you wanted to keep the average temperature for each month of the year, you could construct an array having twelve data items. The DIM statement can be used to define an array:

```
10    DIM T(12)
```

This statement defines an arithmetic array, T, containing 12 items, or *members*. The DIM statement is the same statement we used earlier to define character length, using B$6 and C$12 for character variables of length 6 and 12. The computer recognizes an item as an array by the appearance of parentheses. The parentheses are used to define the number of items in the array.

Arrays can be arithmetic or character. For example:

```
20    DIM T$(12)
```

This statement defines a character array having twelve members.

A DIM statement can specify the length of the members of a character array at the same time it is defining the array:

```
20    DIM T$10(12)
```

Here each of the members of array T$ is assigned a length of 10; without the length specification, each member, like other character variables, would be assumed to be 18 characters long. All members of a character array have the same length.

Naming Arrays

Character arrays are named in exactly the same way as character variables, that is, the name must consist of a single alphabetic character (including the alphabet extenders) followed by the currency symbol. Thus, the name A$ can name either a character array or character variable. Arithmetic arrays are named in almost the same way as arithmetic variables. An arithmetic array name may consist *only* of a single alphabetic character (including the alphabet extenders); you may recall that arithmetic variables can also be named with a character followed by a digit. Thus, the name A can be used for either an arithmetic array or arithmetic variable, but the name A1 can be used only for an arithmetic variable.

Defining Arrays

Defining an array in a DIM statement is known as an *explicit* array declaration. There is another way to define an array through *implicit* declaration; that is, by referring to a member of an array in a program statement without having defined it first in a DIM statement. When you refer to an array member without explicitly declaring it in the DIM statement, the computer will recognize that you are working with an array and will automatically allow space for ten members. To refer to a particular member of an array, you specify it by its location in the array. For example, T(1) refers to the first member of the array named T, T(2) refers to the second member, T(3) refers to the third member, and so on. Each number giving the location of a particular member is called a *subscript.* If the following statement appeared in the program:

```
40    LET T(9) = 69
```

only the ninth member of T would be assigned the value 69; all other members would remain unchanged.

Remember that an array defined implicitly is assumed to have ten members. So in order for our array T to contain twelve members, we must explicitly define it. If an array has very few members, (for example, two or three), it would be wise to use a DIM statement, such as:

```
10    DIM A(2), B(3)
```

The DIM statement, in addition to defining the number of members in the array, also defines the number of *dimensions* in the array.

So far, we have discussed only one-dimensional arrays. In BASIC, you can also have arrays of two dimensions. Assume that values have been assigned to our array T, such that:

```
T(1)  is  31
T(2)  is  43
T(3)  is  42
T(4)  is  57
T(5)  is  64
T(6)  is  73
T(7)  is  79
T(8)  is  79
T(9)  is  69
T(10) is  58
T(11) is  44
T(12) is  39
```

Let us assume that these values represent the average temperature for the month; T(1), represents January's average, T(2), February's, etc.

For various reasons, another programmer might want to consider the year as divided into four quarters of three months each; he could define his array (call it M) as a two-dimensional array, as follows:

```
15    DIM M(4,3)
```

In this statement, the array M is defined as a two-dimensional array containing 12 members (the product of 4 and 3), just like the array T. The difference is that the members of M are distributed over two dimensions, whereas in T they are distributed over only one dimension. Conceptually, the two dimensions of M can be thought of as rows and columns—four rows and three columns. The first value would be identified as being in the first row and the first column, or as M(1,1); the second value would be in M(1,2), the first

row, the second column; the third in M(1,3), the first row, third column. The fourth item is M(2,1), or the second row, the first column, the fifth item, M(2,2), would be in the second row, second column, and so on.

Assuming that the same temperatures assigned to T are assigned to M, notice the difference in the way each item is referred to:

Array T	Temperature	Array M
T(1)	31	M(1,1)
T(2)	43	M(1,2)
T(3)	42	M(1,3)
T(4)	57	M(2,1)
T(5)	64	M(2,2)
T(6)	73	M(2,3)
T(7)	79	M(3,1)
T(8)	79	M(3,2)
T(9)	69	M(3,3)
T(10)	58	M(4,1)
T(11)	44	M(4,2)
T(12)	39	M(4,3)

Two subscripts are needed to refer to a particular member of M; for example, M(3,1) refers to the temperature for July, the first month in the third quarter.

Note the difference between a subscript and the array dimension specification. A subscript *refers* to a particular member of an array. It can be any valid arithmetic expression (for example, a numeric constant or an arithmetic variable). In the example above, the expression is truncated to the value 1, 2, 3, or 4, depending on the value of the member of the array. The dimension specification *defines* the number of members of an array. The dimension specification can appear only in a DIM statement and it must be indicated by unsigned integers only. An array name cannot appear in a DIM statement if the array has already been defined—either implicitly by usage or explicitly by already being defined in a previous DIM statement.

As you can with a one-dimensional array, you can implicitly define a two-dimensional array by using it in a program statement without defining it in a DIM statement first. You would do this by referring to a particular member, using two subscripts. For example, A(4,3) would refer to the item in the fourth row, the third column of the array A. A two-dimensional array defined implicitly will be assigned the dimensions (10,10), or 100 members altogether. If the value of either dimension is to exceed ten, however, you must use a DIM statement to define the array as you would for a one-dimensional array that exceeds ten members. Remember that DIM statements to define arrays must appear in the program *before* you refer to the array.

Still another way to define an array is by declaring it implicitly by context, that is, by using it in an assignment statement without defining its number of members or dimensions. For example, this statement:

```
180  MAT  A  =  (5)
```

assigns the value 5, enclosed in parentheses, to all members of the array A, identified as an array by the keyword MAT. MAT is used in an assignment statement to indicate that an operation is to be performed on an entire array, in this particular case, to assign the value 5 to all members. If A has not been previously declared, its occurrence in statement 180 would declare it by context, and it would be implicitly declared as a two-dimensional array, with ten members in each dimension, for a total of 100 members.

Placing Values into Arrays

Initially the system sets all arithmetic arrays to zero and all character arrays to blanks. Arrays can be given other values through LET, READ, and INPUT statements, just like other variables. The assignment statement can assign values to individual array members or to all the members of the array. Here are some examples:

```
300   LET A(4,5) = 10
320   MAT A = (15)
330   LET P$(4) = "PHILADELPHIA"
```

The first example assigns the value 10 to the member in the fourth row, fifth column of the two-dimensional arithmetic array A. In the second example, the keyword MAT identifies A as an array, and the value 15 is assigned to all the members of the array. (This is a special form of the assignment statement and is known as the "array assignment statement.") In the third example, the value PHILADELPHIA is put into the fourth member of the one-dimensional character array P$.

When specifying values by means of READ and INPUT statements, you must remember that every array member that is to receive a value must be represented in the statement, and a value must be supplied for each member specified. Let's look at these statements:

```
10    DIM T(12), T$(12)
15    PRINT 'ENTER 3 TEMPERATURES, THEN THREE MONTHS'
20    INPUT T(1), T(2), T(3), T$(1), T$(2), T$(3)
```

The DIM statement defines the arithmetic array T and the character array T$, each with 12 members. The INPUT statement states that values will be supplied at execution time for the first three members of each array. Execution of the INPUT statement at a terminal causes the computer to print the question mark (?) at the terminal. A valid response would be:

```
?
31,43,42,JANUARY,FEBRUARY,MARCH
```

The first three values are entered into T(1), T(2), and T(3), respectively. The next three values are entered into T$(1), T$(2), and T$(3), respectively.

The following statement can be used to enter values for the arithmetic array A, consisting of four rows and three columns:

```
30    INPUT  MAT  A(4,3)
```

When this statement is executed, the computer prints a question mark at the terminal, and you can enter the three values for the first row, followed by a carrier return:

```
?
1,1,1
```

For each succeeding row, the computer prints a double question mark, after which you enter three more values followed by a carrier return:

```
??
2,2,2
??
3,3,3
??
4,4,4
```

Another way of assigning input values to arrays is through use of a FOR/NEXT group in conjunction with the READ and DATA statements.

For example, if you wanted a list of 15 numbers assigned to an array named B, you could write:

```
10    DIM B(15)
20    FOR I = 1 TO 15
30    READ B(I)
40    NEXT I
50    DATA 2,3,5,7,11,13,17,19,23,29,31,37,41,43,47
```

The subscript I is used to step through the values in the data table.

Redimensioning Arrays

Once an array has been dimensioned, either explicitly using a DIM statement, or implicitly through usage, it cannot be explicitly dimensioned again. But it can be *redimensioned;* that is, the array can be given new dimensions. A one-dimensional array can be redimensioned into a two-dimensional array, or it can be redimensioned into a one-dimensional array with a different number of members. Similarly, a two-dimensional array can be redimensioned into a one-dimensional array or a two-dimensional array having a different number of members in either or both dimensions. The rule to remember when redimensioning an array is that the total number of members in the new array may not exceed the total number in the original array. For example, the array M(12,10) has 120 members, the product of 12 and 10. It can be redimensioned as long as the new array does not contain more than 120 members (it can contain fewer). Thus, M(12,10) may be correctly redimensioned to M(40,3), or M(100), but not to M(40,4).

One way to redimension an array is by stating its new dimensions right after the array name in an array assignment statement. For example, if you want to redimension the array C(5,5) to C(3,4), you could use the array assignment statement:

```
MAT C(3,4) = (0)
```

The word MAT is used to indicate that operations are to be performed on the entire array, or matrix. This statement changes the array dimensions to (3,4) and assigns the value zero to each member of the newly dimensioned array.

Difference between MAT and LET

It is important to note the distinction between the array assignment statement, identified by the word MAT, and the LET assignment statement. The following example shows a sequence of assignment statements along with the output from each one.

`10 DIM C(2,3)`	Array C is initialized to	$\begin{bmatrix}000\\000\end{bmatrix}$
`20 LET C(2,1) = 1`	Array C is	$\begin{bmatrix}000\\100\end{bmatrix}$
`30 MAT C = (5)`	Array C is	$\begin{bmatrix}555\\555\end{bmatrix}$
`40 MAT C(3,2) = (8)`	Array C is	$\begin{bmatrix}88\\88\\88\end{bmatrix}$
`50 LET C = 9`	Variable C is 9	

Statement 10 defines arithmetic array C as a 2x3 array and initializes each member to 0. Statement 20 assigns the value 1 to the member in the second row, first column of the array. Statement 30 assigns the value 5 to all members of the array. Statement 40 redimensions the 2x3 array into a 3x2 array and assigns the value 8 to all members. Statement 50 does not refer to an array but to an arithmetic variable, C, and assigns the value 9 to it. VS BASIC allows you to use the same name to represent both an array and a simple variable in the same program.

The array assignment statement can also assign the values of an array to another array, as long as both arrays have identical dimensions. Let's look at this example.

```
100 DIM Y(4), Z(4)
     .
     .
     .
150 MAT Y = (A*B)
     .
     .
     .
180 LET Y(3) = 15
     .
     .
     .
200 MAT Z = Y
```

Statement 150 assigns the value of the expression A*B to all the members of the array Y. The expression must always be enclosed in parentheses. Statement 180 assigns the value 15 to the third member of Y. Note the difference between the LET statement and the MAT statement. Statement 200 assigns the values in array Y to array Z. If the only change made to array Y between statements 150 and 200 was the assignment made in statement 180, array Y will contain the values A*B in members 1, 2, and 4 and the value 15 in member 3. Array Z will be assigned these values in the corresponding members.

In order for the values of one array to be assigned to another, both arrays must have identical dimensions. For example, if Z had the dimension (5), or (2,2), it would have to be redimensioned to the dimensions of Y before it could receive Y's values.

Array Operations

A number of different operations can be performed on arrays. Arithmetic arrays can be used in simple arithmetic operations, such as adding or subtracting the values of members in different arrays, and in true mathematical matrix operations such as matrix multiplication, matrix identity, and matrix transposition. Additionally, values in both arithmetic and character arrays can be sorted in ascending or descending order. Arrays used in arithmetic operations must have the same number of dimensions. Let's look at some of the operations available.

Array Addition and Subtraction

Example 1:

```
10    DIM X(5), Y(5), Z(5)
20    MAT X = Y+Z
```

In this example, each member of the array X is to be assigned the sum of the corresponding members of the arrays Y and Z. The values of Y(1) and Z(1) are added and the sum is assigned to X(1), the values of Y(2) and Z(2) are added and assigned to X(2), and so on.

Example 2:

```
30    DIM X(5), Y(5), Z(5)
40    MAT X = Y-Z
```

This example is like the first example, except that the array X is assigned the difference between the corresponding members of the arrays Y and Z.

Scalar Multiplication

Scalar multiplication is the process whereby each member of an array is multiplied by the same number.

Example:

```
35    DIM A(10,5), B(14)
40    MAT A(14) = (4)*B
```

In statement 40, A is redimensioned to correspond to the dimensions of the array B. Then, the value in each member of B is multiplied by 4 and the product is assigned to the corresponding member of A; B(1)*4 is assigned to A(1), B(2)*4 to A(2), etc.

Sorting

Sorting operations can be performed on character as well as on arithmetic arrays. Character arrays are sorted alphabetically, arithmetic arrays numerically. Arrays can be sorted in ascending or descending sequence, depending on the keywords ASORT and DSORT.

Example:

```
20    DIM A$(3,3), B$(3,3)
50    MAT B$ = ASORT(A$)
```

The values in the members of the array A$ are compared and arranged in ascending alphabetic order, and then are entered, in sorted order, into the array B$. The keyword ASORT indicates ascending sorting order. If the keyword DSORT were used, the array would be sorted in descending order.

Identity Function

An identity function may be performed only on a square array, that is, a two-dimensional array having the same number of rows and columns. The identity function assigns the value 1 to all array members whose subscripts are equal and the value 0 to all other array members.

Example:

```
75    DIM C(3,3)
80    MAT C=IDN
```

The value 1 is assigned to members C(1,1), C(2,2), and C(3,3). All other members, C(1,2), C(1,3), etc., are assigned the value 0. The array would look like this:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Inverse Function

An inverse function causes the matrix inverse of one square array to be assigned to another square array. Not every matrix has an inverse; before using the inverse function, you should test the matrix with the determinant function DET. The inverse of a matrix exists if DET returns a value other than 0.

Example:

```
 85    DIM J(2,2), K(2,2)
 90    IF DET(K)=0 THEN 110
 95    MAT J = INV(K)
100    GO TO 120
110    PRINT 'MAT K SINGULAR'
120    ...
```

If the array K had the value:

$$\begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}$$

array J would be assigned the matrix inverse:

$$\begin{bmatrix} 2 & -1 \\ -1 & 1 \end{bmatrix}$$

Transpose Function

The transpose function causes the values of one two-dimensional array to be transposed into another two-dimensional array. The values contained in column 1 of one array are transferred into row 1 of the other, the values in column 2 are transferred into row 2, etc. The number of rows in each array must equal the number of columns in the other array.

Example:

```
 45    DIM A(3,4), B(4,3)
 50    MAT B = TRN(A)
```

If A contained the values:

$$\begin{bmatrix} 1 & 10 & 100 & 1000 \\ 2 & 20 & 200 & 2000 \\ 3 & 30 & 300 & 3000 \end{bmatrix}$$

B would contain

$$\begin{bmatrix} 1 & 2 & 3 \\ 10 & 20 & 30 \\ 100 & 200 & 300 \\ 1000 & 2000 & 3000 \end{bmatrix}$$

Matrix Multiplication

Matrix multiplication is the process whereby the matrix product of two arithmetic arrays is assigned to a third array.
All three arrays involved in matrix multiplication must be two-dimensional.

Example 1:

```
65    DIM X(2,2), Y(2,2), Z(2,2)
70    MAT Z = X*Y
```

If X contained $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ and Y contained $\begin{bmatrix} e & f \\ g & h \end{bmatrix}$

the values of Z $\begin{bmatrix} j & k \\ l & m \end{bmatrix}$, would be constructed as follows:

j = a*e + b*g
(members in first row of X times members in first column of Y)

k = a*f + b*h
(members in first row of X times members in second column of Y)

l = c*e + d*g
(members in second row of X times members in first column of Y)

m = c*f + d*h
(members in second row of X times members in second column of Y)

All the arrays shown in example 1 are two-dimensional, square, and with the same number of members. Arrays used in matrix multiplication need not be square or have the same number of members, but must be two-dimensional and *conformable*. Look at this example:

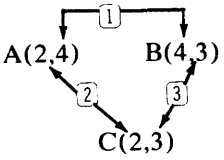
```
75   DIM  A(2,4), B(4,3), C(2,3)
80   MAT  C = A*B
```

Remember that the first subscript in a two-dimensional array indicates the number of rows, and the second subscript indicates the number of columns. (In the example above, A has two rows and four columns.) To be conformable for matrix multiplication, arrays must meet these requirements:

1. The number of columns in the first array to be multiplied must equal the number of rows in the second. In the example above, A(x,4) = B(4,x).
2. The number of rows in the receiving array must equal the number of rows in the first array. In the example, C(2,x) = A(2,x).

3. The number of columns in the receiving array must equal the number of columns in the second array. In the example, C(x,3) = B(x,3).

These requirements are graphically represented below:



The arrays in statements 75 and 80 are conformable, and thus are valid for matrix multiplication operations.

If A contained $\begin{bmatrix} a & b & c & d \\ e & f & g & h \end{bmatrix}$ and B contained $\begin{bmatrix} i & j & k \\ l & m & n \\ o & p & q \\ r & s & t \end{bmatrix}$

the values of C $\begin{bmatrix} u & v & w \\ x & y & z \end{bmatrix}$, would be constructed as follows:

u = a*i + b*l + c*o + d*r
(members in first row of A times the members in first column of B)

v = a*j + b*m + c*p + d*s
(members in first row of A times the members in second column of B)

w = a*k + b*n + c*q + d*t
(members in first row of A times the members in third column of B)

x = e*i + f*l + g*o + h*r
(members in second row of A times the members in first column of B)

y = e*j + f*m + g*p + h*s
(members in second row of A times the members in second column of B)

z = e*k + f*n + g*q + h*t
(members in second row of A times the members in third column of B)

Functions and Subroutines

In addition to the intrinsic functions supplied as part of the BASIC language (see Part II of this publication for a list of these functions), you can define any other function or write a program segment, called a subroutine, which you expect to use several times in your program.

Functions

User-written functions can be arithmetic or character. An arithmetic function is named by the letters FN followed by a single letter or digit. A character function is named by the letters FN followed by a single letter and the currency symbol, $.

The following can be names of arithmetic functions:

```
FNA
FNB
FN3
FN#
```

The following can be names of character functions:

```
FNA$
FN#$
```

A user-written function is named and defined by the DEF statement. For example:

```
10    DEF FNE(X) = EXP (X**2)
```

defines the natural exponential of X squared, using the intrinsic function EXP. The arithmetic variable X, enclosed in parentheses after the function name FNE, is called a *dummy variable.* You can have more than one dummy variable, and the list of variables can contain both arithmetic and character variables. Your function performs its defined calculation on the actual values supplied for these dummy variables. (The expression value substituted for each dummy variable is called an *argument.*) After defining a function, the function name and its accompanying argument(s) can be used anywhere in your program. For example:

```
10    DEF FNE(X) = EXP(X**2)
      .
      .
      .
50    LET Y = FNE(.5)
60    LET Z = FNE (C + 2)
70    PRINT FNE(3.75) + Y / Z
```

User-defined functions can be defined in one statement or over a group of statements. A function defined in one statement, such as the function illustrated above, is called a *single-line* function. A function defined over many statements is called a *multiline* function. A multiline function begins with the word DEF, the function name, and any arguments, the same as single-line functions. However, the DEF statement does not contain the equal sign or an expression. Rather, the expression is developed by the statements following the DEF, and is defined in a RETURN statement, which computes the value and "returns" the value to the program. The end of a multiline function is defined by the FNEND statement. Here is the way the statements in a multiline function must be sequenced:

```
DEF function name [ (args, if any) ]
.
.
.
RETURN expression
.
.
.
FNEND
```

Here is an example of a multiline function:

```
30    DEF FNA(X,Y)
40    IF X > 0 THEN RETURN X+Y ELSE RETURN X-Y
50    FNEND
```

This function uses two arithmetic variables as dummy arguments, labeled X and Y. The function tests the value of the first argument. If that value is greater than zero, it is added to the value of the second argument and the sum is returned to the program. If the value is less than or equal to zero, the value of Y is subtracted from it and the difference is returned.

If this function were used in the following program, C would have a value of 7 and D would have a value of -2.

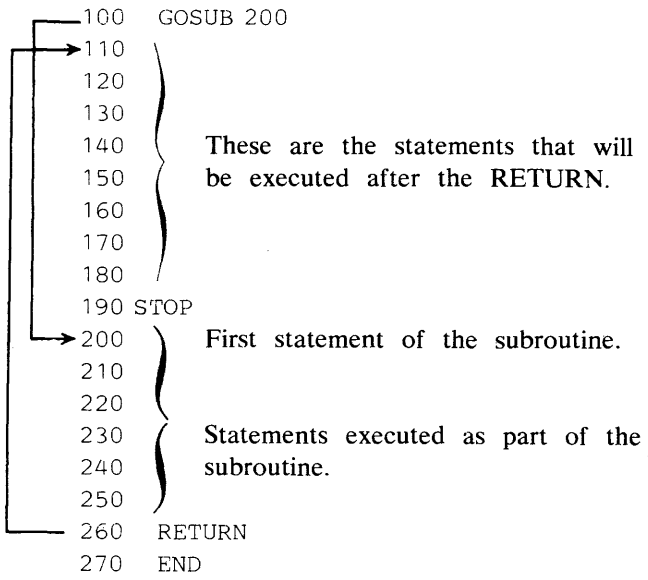
```
 30   DEF FNA(X,Y)
 40   IF X .GT. 0 THEN RETURN X+Y ELSE RETURN X-Y
 50   FNEND
      .
      .
      .
100   LET A = 5
120   LET B = 2
130   LET C = FNA(A,B)
150   LET D = FNA(0,2)
```

Subroutines

Another way of writing a group of statements to be executed at different times in your program is to group them into a subroutine. Execution of a subroutine begins with the GOSUB statement, where the number specified in the statement is the statement number of the first statement in the subroutine. For example:

```
100   GOSUB   200
```

causes the computer to skip, or "branch," to statement 200, the first statement in the subroutine. Program execution continues from that point. To cause the computer to branch back to statement 100 (actually, to the next sequential statement following statement 100), the last statement of the subroutine must be a RETURN statement. (This RETURN statement, unlike a RETURN used with a function, contains no expression.) A program containing a subroutine could be sequenced like this:



Statement 100 branches to statement 200. Statement 260 returns control to statement 110. Statement 190 tells the computer the end of the program has been reached. The STOP statement is similar to an END statement except that higher-numbered statements may follow it. Its use is to denote the end of program execution when the logical conclusion of the program occurs somewhere in the middle of the program, as shown here. The STOP statement here is equivalent to writing GO TO 270.

A program illustrating the use of a subroutine is shown below. This program determines the greatest common divisor of three integers. The first two numbers are selected in program statements 30 and 40, and their greatest common divisor (CD) is determined in the subroutine, statements 200-310. The CD just found is assigned to X in statement 60. The third number read in from the INPUT statement is assigned to Y in statement 70. The subroutine is entered a second time from statement 80 to find the greatest common divisor (CD) of these two numbers. The result is, of course, the greatest common divisor of the three given numbers. It is printed out with them in statement 90.

```
 10  PRINT 'ENTER THREE INTEGERS'
 20  INPUT A, B, C
 30  LET X = A
 40  LET Y = B
 50  GOSUB 200
 60  LET X = G
 70  LET Y = C
 80  GOSUB 200
 85  PRINT 'A', 'B', 'C', 'CD'
 90  PRINT A, B, C, G
100  GO TO 320
200  LET Q = INT(X/Y)
210  LET R = X-Q*Y
220  IF R = 0 THEN 300
230  LET X = Y
240  LET Y = R
250  GO TO 200
300  LET G = Y
310  RETURN
320  END
```

Let's assume these numbers are entered when the INPUT statement is executed:

```
ENTER THREE INTEGERS
?
60,90,120
```

The output will be:

```
1       19      37      55          (Print Positions)
A       B       C       CD
 60      90      120     30
```

Another example of input and resulting output is:

```
ENTER THREE INTEGERS
?
32,384,72

A       B       C       CD
 32      384     72      8
```

Computed GOSUB Statement

The computed GOSUB statement is similar to the computed GOTO statement previously discussed in "Testing and Controlling Program Data." They both cause a branch to one of a number of statements based on the computed value of an expression. The difference between the two statements is that the GOSUB branches to a subroutine; the RETURN statement in the subroutine returns program execution to the statement following the computed GOSUB statement.

Consider this example:

```
30   GOSUB 120, 175, 195 ON X-Y
```

A branch is made to one of three subroutines, either the one beginning with statement 120, the one beginning with statement 175, or the one beginning with statement 195, depending on whether the integer value contained in the expression X-Y is 1, 2, or 3, respectively. If the expression X-Y results in a value other than 1, 2, or 3, program execution continues with the statement following the GOSUB.

PRINT USING Image and FORM

The PRINT USING statement is quite similar to the PRINT statement but is much more useful for controlling the format of the answer to be printed. PRINT USING is used in conjunction with an Image or FORM statement to print values according to the format specified in these statements. The PRINT USING statement includes the values to be printed and the statement number of a corresponding Image or FORM statement which specifies the format of the print line. For example:

```
30  PRINT USING 40, N,A
```

This statement refers to statement number 40, an Image or FORM statement, which will instruct the computer how to format the arithmetic variables N and A on the print line.

The Image Statement

Statement 40 could look like this:

```
40  :IN ## YRS AMT = $####.##
```

The colon beginning statement 40 identifies it as an Image statement. The alphabetic characters are printed exactly as they appear in the statement, and the pound sign (#) is the symbol used to indicate that a value will be supplied from the output list in the PRINT USING statement. The value of N replaces the first set of #'s, and the value of A replaces the final set. The decimal point appearing in the final set indicates that the value of A is to be aligned on the decimal point in the image specification.

If N contained the value 10 and A the value 1628.88, the output line produced by statements 30 and 40 would look like:

```
IN 10 YRS AMT = $1628.88
```

In the Image statement, the pound sign, #, is used as a place holder. In statement 40, the first set of #'s indicates that a value is to be displayed using two positions; the second set displays a value over six positions aligned on a decimal point between the fourth and fifth positions. If the value to be printed were smaller than six digits, say the value 300.40, the first, or *high-order* position, would be printed with a blank.

The Image statement can also contain a place holder for an exponential value, using the symbols | | | |. If you wanted to print a value containing an exponent, the Image statement could contain the following sequence of symbols:

```
##.##||||
```

This sequence states that a value is to be printed with four digits aligned on a decimal point between the second and third digits, followed by an exponential value. An exponential value is always printed using four positions for the format: E±dd. The letter E is followed by a plus or minus sign indicating a positive or negative exponent, followed by two digits. Therefore, a set of four | 's must always be specified as placeholders for exponents. If an Image specification contained this sequence:

```
##.##||||
```

the table below shows how different values would be printed by that sequence.

If the number is	The printed format is
123	12.30E+01
12.3	12.30E+00
.123	12.30E-02

The specification calls for four digits to be printed aligned on the decimal point. Therefore, the number 123 is represented as 12.30 with an exponent of +1. The exponent tells us two things: the direction in which the decimal point is to be moved, (+, to the right, and -, to the left), and the number of digits over which it is to be moved. In the first number, the exponent +1 tells us to move the decimal point one position to the right; the number 12.30E+01 is the same as 123. In the second example, the number 12.3 can be aligned on the decimal point with no action required by the exponent, hence an exponent of E+00; the number 12.30E+00 is the same as 12.3. The third example tells us to move the decimal point two positions to the left; 12.30E-02 is the same as .123.

The FORM Statement

The FORM statement, similar to the Image statement, offers greater formatting capabilities. It provides for the same specification characters, # and |, and also for many more through the use of the PIC keyword, to specify numeric data. It provides a special code to specify character data. It contains format control specifications to tell the computer how to position output on a print line; one of these specifications, SKIP, must be coded on the FORM statement to cause a line to be printed.

Numeric Specification—PIC

The PIC specification in the FORM statement shows a "picture" of the way a number should be formatted. This picture is enclosed in parentheses. The symbols #, ., and |, previously illustrated in the Image statement, could be used in the FORM statement in this format:

```
PIC(##.##||||)
```

You recall that the # symbol is used as a place holder for a digit and the symbol | is used as a place holder for an exponent. The PIC specification has these additional place holders, or *digit specifiers:*

Symbol	Meaning
Z	Replace a leading zero with a blank.
*	Replace a leading zero with an asterisk.
$	Floating dollar sign. A dollar sign is to be printed immediately before the first significant digit.
+	Floating sign. A plus sign for a positive number, or a minus sign for a negative number, is to be printed immediately before the first significant digit.
-	Floating minus sign. A minus sign for a negative number, or a blank for a positive number, is to be printed immediately before the first significant digit.

Here are examples of digit specifiers. Assume a data item containing the value 112233 is to be printed.

If the PIC specification were:	Printed output would be:
PIC(#########)	000112233
PIC(ZZZZZZZZZ)	112233
PIC(ZZZZZZ###)	112233
PIC(******###)	***112233
PIC($$$$$$###)	$112233
PIC(++++++###)	+112233
PIC(---######)	112233

If a floating character (dollar sign, plus sign, or minus sign) is specified only once in a PIC specification, it does not float through the field but instead is printed in the indicated position. For example:

If the PIC specification were:	Printed output would be:
PIC($ZZZZZ###)	$ƀƀ112233
PIC(+ZZZZ###)	+ƀ112233

(The character ƀ represents a blank.)

The PIC specification can also contain *insertion characters,* to edit a printed item. The difference between digit specifiers and insertion characters is that digit specifiers indicate how the number itself is to be treated, while insertion characters simply insert additional characters into a field, generally to improve readability. The following insertion characters can be specified:

Symbol	Meaning
B	Print a blank unconditionally.
,	Print a comma conditionally (that is, only if a digit precedes the comma.)
/	Print a slash conditionally (only if a digit precedes the slash.)
.	Print a decimal point conditionally (if the value to be printed is non-zero and zero suppression is not in effect).
+	Trailing sign. When the + appears in the rightmost position of a PIC specification, it is treated as a trailing sign. A plus sign is printed for a positive number, a minus sign for a negative number.
-	Trailing minus sign. When the - appears in the rightmost position of a PIC specification, it is treated as a trailing sign. A minus sign is printed for a negative number, a blank for a positive number.
CR	Trailing credit sign. When the CR appears in the rightmost positions of a PIC specification, it is treated as a trailing credit sign. The letters, CR, are printed for a negative value and two blanks are printed for a positive value or zero.
DB	Trailing debit sign. When the DB appears in the rightmost positions of a PIC specification, it is treated as a trailing debit sign. The letters, DB, are printed for a negative value and two blanks are printed for a positive value or zero.

Here are examples of insertion characters added to the examples previously shown:

If the PIC specification were:	Printed output would be:
PIC(###B##B####)	000 11 2233
PIC(ZZZBZZBZ###)	11 2233
PIC(ZZZ,ZZZ,###)	112,233
PIC(ZZZZZ/Z#/##)	11/22/33
PIC(******#.##)	*112233.00
PIC($$$$$$###+)	$112233+
PIC($$$,$$$,$$$.##)	$112,233.00

In the first example, a blank is entered after the third and fifth digits. Since # is denoted as the digit specifier, leading zeros are not suppressed.

The second example illustrates the blank used with the Z digit specifier which does suppress leading zeros.

The third example illustrates the use of commas. The first comma is not printed because no digit precedes it (zero suppression having been specified); the second comma is printed.

The fourth example inserts slashes.

The fifth example illustrates the effect of a decimal point; since the number 112233 is an integer number, it is aligned on the decimal point and zeros print out in the decimal portion of the field.

The sixth example adds a trailing sign to a field that also contains floating dollar signs.

The seventh example adds commas and a decimal point to format a dollar amount. Note that the first comma is not printed, nor is its absence marked by a blank—it simply disappears, as it did in example three. The dollar sign floats over the comma.

The following are examples of PIC specifications with trailing signs along with the printed results for negative and positive values.

PIC specification	Printed Output for a Negative 123	Printed Output for a Positive 123
PIC (###+)	123-	123+
PIC (### -)	123-	123
PIC (###CR)	123CR	123
PIC (###DB)	123DB	123

Here are some more examples of PIC:

```
PIC (ZZZ###+)
PIC ($$$,$$$,###.##)
```

The first example states that a number containing up to six significant digits may be printed. Leading zeros appearing in the first three positions are suppressed. Either a plus sign or a minus sign is to be printed in the last position. If the number 1234 were to be printed using this specification, it would look like:

```
␣␣1234+
```

The second example states that the dollar sign is to print before the first printed digit. The comma insertion character indicates that a comma is to be printed *only* if a significant digit precedes it. If the amount $12628.88 were to be printed using this specification, it would look like:

```
␣␣␣␣$12,628.88
```

If the number 3.33 were to be printed, it would look like:

```
␣␣␣␣␣␣␣$003.33
```

Character Specification—PIC

The PIC specification in the FORM statement can also accept character data. The length of the character data string to be printed is determined by the total number of positions in the PIC specification.

Character data is printed starting at the first position of the PIC specification. Any unused portion of the PIC specification is replaced by blanks. If the length of the character data string exceeds the PIC specification length, then the character data string is truncated on the right to the length of the PIC specification.

For example, executing the following:

```
10    A = 12345.67
20    PRINT USING 40, 'HEADING1'
30    PRINT USING 40, A
40    FORM PIC(ZZZ,ZZZ,##), SKIP
```

results in the output:

```
HEADING1
12,345.67
```

Character Specification—C and Literals

The character specification code C is used in the FORM statement to indicate a place where character data is to appear. The actual character data is written in the corresponding PRINT USING statement. Character data can also be specified in the FORM statement by using literals that must be enclosed in single or double quotation marks.

To print character data using the character specification code C, the corresponding PRINT USING statement could be written as follows:

```
30    PRINT USING 50, 'COST OF ',A1,' CHAIRS IS' , B1
```

the corresponding FORM statement could look like this:

```
50    FORM C, PIC(Z#), C, PIC ($$$,$$#,##)
```

Using the literals, the corresponding PRINT USING statement could be written like this:

```
30    PRINT USING 50, A1, B1
```

the corresponding FORM statement could look like this:

```
50 FORM 'COST OF ' PIC(Z#), ' CHAIRS IS ', PIC($$$,$$#,##)
```

The first PIC specification describes the arithmetic variable A1; if the value is zero, a blank is printed in the leftmost position, followed by a zero. The second PIC specification describes the variable B1.

The character specification code C marks a place for character data regardless of the number of characters to be printed. You could specify the exact number of characters to be printed by indicating the number after the C code. For example:

```
C6
```

This specification indicates that six characters are to be printed. Care should be used when specifying a number, because only that number of characters is printed. For example, if you specified C6 to print the character string COST OF, only the characters COST O would be printed.

Format Control Specifications—X, POS, SKIP

Format control specifications provide great flexibility in formatting an output line. These specifications allow you to space over a number of print positions on a line, to specify the print position where a data item is to begin printing, and to skip print lines.

The Xn specification spaces over *n* print positions. For example, X10 would cause the terminal or printer to space the next 10 positions before printing a data item.

The POSn specification prints a data item beginning in position *n*. For example, POS50 would cause the next data item to print beginning in position 50.

The SKIPn specification skips *n* print lines. To skip five lines, specify SKIP5. To skip to the next line, specify SKIP1 or SKIP with no number. For example, to cause statement 50, shown earlier, to print a line, SKIP must be added to it:

```
50  FORM C, PIC(Z#), C, PIC($$$,$$#.##), SKIP
```

Statement 50 is now complete, and if combined with PRINT USING statement number 30,

```
30  PRINT USING 50, 'COST OF  ', A1, '  CHAIRS IS',  B1
```

would result in this output:

```
COST OF  14  CHAIRS IS $1,510.00
```

Here are additional statements using format control specifications:

Example 1:

```
140  PRINT USING  145, A1,B1
145  FORM  POS15, PIC(Z#), POS32, PIC($$$,$$#.##), SKIP1
```

Statement 145 uses the POS and SKIP control specifications. POS15 positions the terminal or printer at position 15 before printing the value contained in A1 described by the PIC specification. POS32 begins printing the value of B1 at position 32. After all printing is completed, SKIP1 causes the carriage to skip to the next print line.

Example 2:

```
110  PRINT USING  115, 'COST OF', A1, 'CHAIRS IS', B1
115  FORM  X5, C, POS15, PIC(Z#), POS20, C, POS32,
     PIC($$$,$$#.##), SKIP1
```

In statement 115, X5 states that the first five positions of the print line are to be skipped, and the character data controlled by the C code, the string COST OF, is to be printed. POS15 prints the value of A1 beginning in position 15.

POS20 prints the character string CHAIRS IS beginning in position 20. POS32 prints the value of B1 beginning in position 32. SKIP1 causes the line to be printed.

Note that although statement 115 is shown coded over two lines in this example, when you enter it at the terminal, it cannot be continued onto a second line.

Following is a program that uses these statements.

```
100   A1 = 15
105   B1 = A1 * 115.25
110   PRINT USING 115, 'COST OF', A1, 'CHAIRS IS', B1
115   FORM X5, C, POS15, PIC(Z#), POS20, C, POS32,
      PIC($$$,$$#.##), SKIP1
120   FOR A1 = 14 TO 1 STEP -1
130   B1 = A1 * 115.25
140   PRINT USING  145, A1, B1
145   FORM POS15, PIC(Z#), POS32, PIC($$$,$$#.##), SKIP1
150   NEXT A1
160   END
```

This program finds the cost of 15 chairs to 1 chair at $115.25 each. Statements 110 and 115 print out the first line, statements 140 and 145 print out all succeeding lines based on the loop defined between statements 120 and 150.

Output from this program would look like this:

Print Position	6	15	20	32
	COST OF	15	CHAIRS IS	$1,728.75
		14		$1,613.50
		13		$1,498.25
		12		$1,383.00
		11		$1,267.75
		10		$1,152.50
		9		$1,037.25
		8		$922.00
		7		$806.75
		6		$691.50
		5		$576.25
		4		$461.00
		3		$345.75
		2		$230.50
		1		$115.25

Example 3:

```
50    R$ = 'WINS. '
60    IF A < B THEN R$ = 'LOSES.'
70    IF A = B THEN R$ = 'TIES. '
80    PRINT USING 90, 'HOME TEAM XXXXXX   FINAL SCORE',
      A, '-', B,R$
90    FORM POS10, C, PIC(Z#), C, PIC(Z#), POS20, C6
```

Statement 90 uses the POS20 and C6 control specifications to overlay position 20 of the print line with the value of R$. If A is 3 and B is 24, the printed line would look like this:

Print Position	10	20	28
	HOME TEAM	LOSES.	FINAL SCORE 3-24

Program Chaining

With the program chaining technique, BASIC programs can be shared with other BASIC programs and with other users. For example, suppose that in writing a program you discover that an operation you want to perform is available as a separate program. It could be time saving to you to be able to use that program in conjunction with the one you are currently writing. The CHAIN and USE statements can help you access and execute that program.

The CHAIN statement is used in one BASIC program to tell the computer to stop executing the current program and start executing another BASIC program. To tell the computer which program to start executing, you name it in the CHAIN statement. Here's an example:

```
500  CHAIN 'PROGB'
```

This statement instructs the computer to begin executing the program named PROGB. Note that when the CHAIN statement is executed, the current program (the program containing the CHAIN statement) is terminated.

You can "pass" a character-string value to the chained program by specifying the value in the CHAIN statement right after the program name. For example:

```
500  CHAIN 'PROGB', J$
```

The value contained in the character variable J$ is passed to PROGB, that is, it becomes accessible for use in that program.

The length of the character-string value you can pass to the chained program has a maximum length of 255 characters.

In the program being chained, the USE statement designates the character variable that will receive the value passed from the CHAIN statement. For example, the value passed by J$ to PROGB can be received by PROGB in the statement:

```
200  USE K$
```

Note that the USE statement is written in the new program, the program being chained, and the CHAIN statement is written in the chaining program, the one requesting execution of another program.

The USE statement may appear anywhere in the chained program. It results in the USE variable being initialized with the value passed by the chaining program.

The CHAIN and USE statements derive their value in being able to help you string two or more programs together instead of having to code similar program sections for individual programs.

An example of CHAIN and USE is:

Chaining Program

```
 10  REM  THIS IS PROGA
          .
          .
          .
300  CHAIN  'PROGB',  A$
310  END
```

Chained Program

```
 10  REM  THIS IS PROGB
          .
          .
          .
 50  USE Z$
          .
          .
          .
120  END
```

When statement 300 in PROGA is reached, execution transfers to PROGB. At the start of execution of PROGB, the variable Z$ is initialized to the value contained in variable A$ of the CHAIN statement.

```
CHAIN 'PROGB',A
```

is illegal because A is numeric.

Stream-Oriented Files

A file is a group of related data items which are stored together. A stream-oriented file is one in which all the items are arranged as a stream of data, that is, all the items are stored in sequential order.

In the section "More About Loops—Using the FOR and NEXT Statements," we illustrated the use of the compound interest program shown below:

```
 90  PRINT 'ENTER PRINCIPAL'
100  INPUT P
105  PRINT 'TIME', 'RATE', 'AMOUNT'
110  FOR T = 1 TO 10 REM VARY THE TIME
120  FOR R = 1 TO 20 REM VARY THE RATE
130  LET A = P*(1 + R/100)**T REM COMPUTE THE AMOUNT
140  PRINT T, R, A
150  NEXT R REM USE THE NEXT RATE
160  NEXT T REM USE THE NEXT TIME
170  END
```

You may recall that the PRINT statement in this program was executed 200 times to produce an output listing containing 200 values. These 200 values could be grouped as a stream-oriented file. In fact, instead of printing them, we could store them into the file and use them at a later time. By substituting the PUT statement for the PRINT statement, we can create a stream-oriented file; for example:

```
140  PUT 'FILE1', T, R, A
```

This PUT statement instructs the computer to put the values contained in the variables T, R, and A into the file named FILE1. As far as the computer is concerned, both PUT and PRINT mean output; the only difference is whether the output goes into a file or to the terminal.

Naming a File

A filename is always a character expression. It can be a character constant, a character variable, a character-valued function reference, a character array member, or any combination of these. The following examples illustrate different ways to name a file.

Example 1: A filename as a character constant:

```
80   PUT  'XYZ', M
```

The character constant XYZ is the name of the file. Statement 80 writes data into the file XYZ.

Example 2: A filename as a character variable:

```
120  H$  =  'PAYFILE'
130  PUT  H$,  J
```

The filename is the value in the character variable H$. Statement 130 writes data into the file named PAYFILE.

Example 3: A filename as a character-valued function reference:

```
100  F$  =  'FILE1FILE2FILE3FILE4'
110  PUT  STR(F$,16,5),  A,B,C
```

In this example, the filename is a value returned by the function reference STR(F$,16,5). Statement 110 writes data into the file named FILE4, which is the value extracted from the string F$ by the function reference.

Example 4: A filename as a character array member:

```
200   PUT  A$(3)X,Y,Z
```

In this example, the filename is the value contained in the third member of the array A$. If A$ had four members, having the values FILEA, FILEB, FILEC, and FILED, statement 200 would write to the file named FILEC.

Example 5: A filename as a concatenated character value:

```
120   A$ = 'SYS'
130   B$ = '001'
140   C$ = '002'
150   PUT A$ || B$, X, Y, Z
160   PUT A$ || C$, X1, X2, X3
```

In statement 150, the variables A$ and B$ are concatenated to form the filename SYS001; in statement 160, A$ and C$ are concatenated to form the name SYS002.

Information regarding filename syntax appears in "Appendix A: Implementation Considerations."

Retrieving Data From a File

To access data in a stream-oriented file, you use the GET statement, which is the input counterpart to the PUT statement.

To access the first set of values from the file created with the PUT statement above, we can use the GET statement written as follows:

```
20    GET 'FILE1', T, R, A
```

This statement assigns the first three values contained in FILE1 to the variables T, R, and A. It is not necessary that we use the same variable names used when the file was created; for example, we could assign these values to variables X, Y, and Z. The important requirement is that the values in the file and the variables to which they are assigned must be of the same type—arithmetic variables for arithmetic values, character variables for character values.

After the first GET is executed, the file is positioned at the next value. Thus, a second GET for FILE1 would access the next three values in the file. If we wanted to access all the values stored previously, we could issue the GET statement 200 times, or enclose one GET statement in a loop, as follows:

```
10    FOR X = 1 TO 200
20    GET 'FILE1', T, R, A
30    PRINT T, R, A
40    NEXT X
50    END
```

This program would print the 200 values for each T, R, and A. Note that at the end of the program, FILE1 is still available for use.

Activating and Deactivating Files

Files must be activated or "opened" before they can be used. A file can be opened implicitly by the system at the first appearance of the file name in a GET or PUT statement, or it can be opened explicitly by including an OPEN statement in the program. We could have included the following statement as the first statement in our program shown above:

```
5   OPEN 'FILE1' IN
```

The word IN indicates that the file is to be used for retrieving data items from the file for use in the program. If a file were to be used with PUT statements, it could be opened explicitly as an output file with this statement:

```
100  OPEN 'TF' OUT
```

Normally, a file is deactivated or "closed" by the system after execution of your program. However, if you want to switch an input file to output (or vice versa) and continue to use it in the same program, you must explicitly deactivate it by using the CLOSE statement before reopening it. (If you did not use the CLOSE statement and attempted to use an output file for input or vice versa, execution of your program would be terminated.) CLOSE deactivates the file; a subsequent OPEN, GET, or PUT statement opens (reactivates) the file for its new use and repositions it at its beginning.

Remember that under ordinary circumstances, both OPEN and CLOSE are optional; the first GET or PUT will open a file implicitly, and the system will close a file at the end of program execution. The one time that CLOSE is required is if you use the same file for *both* input and output operations in the same program. For example:

```
40   PUT 'AF', A, B, C, D, E
     .
     .
     .
80   CLOSE 'AF'
90   GET 'AF', V, W, X, Y, Z
```

Statement 40 creates an output file named AF and places five values in it. At statement 80, AF is deactivated. In statement 90, AF is reactivated as an input file and the same five values are read and made available for use later in the program.

Notice what happens when an input file is closed and reactivated as an output file.

```
 30  OPEN 'AF' IN
 40  GET 'AF', A, B, C, D, E
 50  LET B = A
 60  LET A = 36
 70  LET C = C+B
100  CLOSE 'AF'
110  OPEN 'AF' OUT
120  PUT 'AF', A, B, C
```

A previously created file, named AF, is activated for input. In statement 40, five values are made available to the program. In statements 50 through 70, new values are acquired for A, B, and C. Statement 100 deactivates AF, and statement 110 re-opens the file for output. Statement 120 places the new values for A through C into the file. All of the old values in the file are lost.

VS BASIC allows you to open or close more than one file with a single OPEN or CLOSE statement. This capability is referred to as a multiple OPEN or CLOSE.

The maximum number of concurrently active input/output files is 15.

Repositioning Files

You may have an occasion to use an input file or an output file more than one time in the same program. The RESET statement allows you to reposition the file without deactivating it (deactivation is necessary only when the function of a file is changed from input to output or vice versa). For example:

```
50   GET "ABC", X, Y, Z, Q, R, S
     .
     .
     .
100  RESET "ABC"
110  GET "ABC", X, Y, Z, Q, R, S
     .
     .
     .
150  RESET "ABC"
160  GET "ABC", X, Y, Z, Q, R, S
```

Between statements 50 and 100, the variables X, Y, Z, Q, R, and S could be used in one set of calculations and their values changed. By repositioning the file, the original values in the file could again be made available and put into variables X, Y, Z, Q, R, and S again for different calculations or uses between statements 110 and 150, and again between 160 and the end of the program. Actually, the RESET statement used in this way functions for files in the same way that the RESTORE statement does for the data table created by the DATA statement.

To add data to the end of the file, you can reset it to its end by using the RESET statement with the END keyword:

```
200  RESET "ABC" END
```

This statement positions ABC to the end of the last data item in the file. PUT statements appearing after statement 200 will place additional values into the file. In effect, RESET END changes an input file to an output file.

Input/Output Error Handling

Certain error conditions can occur while you are processing files. As an example, when reading through a file, you need to take action after the last item is read, otherwise the computer will terminate the program. The EOF clause (for *E*nd *O*f *F*ile) can be written in the GET statement to branch to another program statement when the end of the file is reached.

A GET statement with an EOF clause could look like this:

```
40   GET "MYFILE", X, Y, Z, EOF 100
```

This statement directs the computer to statement 100 when the end of the file is reached. At statement 100, you could end the program, or close the file and continue processing, or perform any number of actions. The important thing is that specifying the EOF clause allows you to retain control of program execution.

The EOF clause can be specified on the PUT statement as well, to prevent writing out beyond the available file space. Note that if an EOF condition occurs, all of the PUT statements may not have been processed.

These are other error handling clauses:

Clause	Meaning
IOERR n	Branch to the statement numbered *n* if a hardware malfunction prevents reading or writing of a record. IOERR can be specified on the GET, PUT, OPEN, CLOSE, and RESET statements.
CONV n	Branch to the statement numbered *n* if a conversion error occurs while a data item is being assigned, for example, if an attempt is made to read character data into a numeric variable. CONV can be specified on the GET statement but not on the PUT statement.

In place of writing these error handling clauses on many GET, PUT, OPEN, CLOSE, and RESET statements throughout your program, you can write them on one or more EXIT statements. An EXIT statement is used in conjunction with many input/output statements to group error handling clauses in one place. The statement could look like this:

```
80    EXIT EOF 100, IOERR 150, CONV 200
```

This statement tells the computer to branch to statement 100 when the end of the file is reached, to branch to statement 150 if a hardware error is encountered, and to branch to statement 200 if a data conversion error is encountered.

If you use the EXIT statement, you should write an EXIT clause on each appropriate input/output statement to tell the computer which statement to branch to. For example:

```
40    GET "MYFILE", X, Y, Z, EXIT 80
      .
      .
      .
80    EXIT  EOF 100, IOERR 150, CONV 200
```

If an error occurs while statement 40 is executing, the program uses statement 80 to determine the appropriate branch to a statement listed for the error condition.

To summarize, a file can be created through the use of PUT statements. Once a file has been created, it can be used as input to the same program by deactivating and reactivating the file, or it can be used as input to some other program (not the one in which it was created) just by using the GET statement. Files can be explicitly opened with the OPEN statement and explicitly closed with the CLOSE statement. The CLOSE statement must be specified to close a file whose function is being changed between input and output. Additionally, values in the file can be retrieved more than once by using the RESET statement, or new values can be added to a file by using the RESET statement with the END keyword.

Error conditions can be controlled by specifying error handling clauses in either the GET, PUT, OPEN, CLOSE, or RESET statements, or in the EXIT statement.

Record-Oriented Files

PUT and GET statements work with stream-oriented files. VS BASIC provides another kind of file capability that permits you to group related data items into records. For example, if you were maintaining a file that described the fifty states, each record might contain the state's name, its population, its area, the capital city, the largest city, etc. All data items for each state could be grouped together into a single record in a *record-oriented* file.

You can define three types of record-oriented files: entry-sequenced, key-sequenced, and relative-record

An *entry-sequenced* file is one in which the records are stored in the order in which they are entered. To use our example of the fifty states, if you enter the records in alphabetic order, the first record would be Alabama, followed by Alaska, Arizona, Arkansas, and so on. If you enter them in geographic order, say with the New England states first, they could be stored in the order Maine, New Hampshire, Vermont, and so on. In an entry-sequenced file, all records are retrieved in sequential order.

A *key-sequenced* file is one in which each record is stored according to a unique identification called a *key*. If the fifty states were stored by key, the name "Montana" could be the key to one record, the name "Oregon", the key to another. When you retrieve a record, you identify which key to look for. The file is searched until the record with the particular key you specified is found. In a key-sequenced file, each record thus can be retrieved directly. Records can also be retrieved in sequential order. If you do not specify keys for the records, they will be retrieved in the order in which they are stored.

A *relative-record* file is one in which each record is stored in a fixed-length 'slot' according to the relative-record number specified at the time the record is written. This number indicates the position of a particular record slot in the file relative to the start of the file. Relative-record numbers start at 1 and continue up to the maximum number of records that can be stored in the file. Relative record 5, then, is in the fifth record slot in the file.

When you retrieve a relative record, you can specify the relative-record number of the record to be found. As with a key-sequenced file, the record is retrieved directly.

Whether you have an entry-sequenced file, a key-sequenced file or a relative-record file depends on how the file is set up. Depending on your environment, you may or may not define record-oriented files yourself. Whatever the case, however, you should design the records for your record-oriented files. For information on record-oriented files as they relate to VSPC, CMS, TSO, OS/VS, and DOS/VS, see Appendix A. "Implementation Considerations."

Designing a Record

Assume you are teaching a class and you decide to set up a record for each person. Each record will contain the person's name, his home address, test marks for five tests you plan to give, his average mark, and a code to indicate whether he is an honor student. The entries in the record might look like this:

NAME	25	characters
ADDRESS	65	characters
GRADES	15	characters
AVERAGE	5	characters
HONORS	1	characters

Altogether, these entries take up 111 characters. You decide to include additional space in each record for possible entries to be added later, such as awards, special achievements, remarks, etc. Altogether, you decide to have a record with 150 characters, and have your installation set up an entry-sequenced file named CLASS with a maximum of 150 positions for the record. Each record could be smaller or equal to the maximum size.

An Entry-Sequenced File

Entering Records

The WRITE FILE statement is the record-oriented counterpart to the PUT statement. It writes records into a file.

At the beginning of the school session, the only information available to you for each student is his name and address. You could write one WRITE FILE statement for each student to enter his name and address, like this:

```
50    WRITE FILE 'CLASS', 'BUTLER, J.S.  ',
      '323 W. 76 STREET, NEW YORK, 10023'
```

(To illustrate the example, the WRITE FILE statement is shown continued onto a second line; note that in actual practice, BASIC statements cannot be continued.)

You could also write one generalized WRITE FILE statement using two character variables for the name and address, like this:

```
50    WRITE FILE 'CLASS', N$, A$
```

This statement would enter the values of the two variables N$ and A$.

This DIM statement could be included in the program, to assign a length of 25 to N$ and 65 to A$:

```
10    DIM N$25, A$65
```

Each record written by the WRITE FILE statement would be arranged in the file this way:

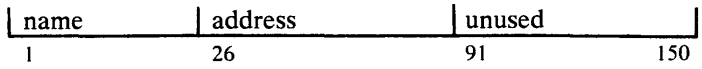
name	address	
1	26	90

Note that this WRITE FILE statement writes a record 90 positions long, and not the 150 that you set up as the maximum size. This record would contain space for the person's name and address but no space for the additional entries to come later. To make room for additional information, you should use the FORM statement in conjunction with WRITE FILE. The FORM statement was described earlier in the discussion of PRINT. The WRITE FILE statement contains a USING clause with the statement number of the

FORM statement, and the FORM statement describes how the entries are to be formatted into the record. The combination of WRITE FILE and FORM statements could look like this:

```
50    WRITE FILE USING 55  'CLASS', N$, A$
55    FORM POS1, C, POS26, C, POS91, X60
```

This FORM statement says that, beginning at position 1 in the record, a character variable is to be written, (N$); beginning at position 26 of the record, a second character variable is to be written, (A$); and, after that variable, an additional 60 spaces are to be skipped. This combination effectively increases the size of the record from 90 to 150:



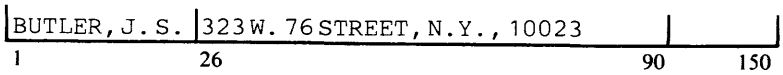
The following program shows how you could enter the names and addresses of the students into the file named CLASS.

```
10    DIM N$25, A$65
20    PRINT 'ENTER NAME AND ADDRESS'
30    INPUT  N$, A$
40    IF N$ = 'LAST' GO TO 70
50    WRITE FILE USING 55 'CLASS', N$, A$
55    FORM  POS1, C, POS26, C, POS91, X60
60    GOTO 30
70    END
```

The program is constructed to recognize the word LAST as the end of input; therefore, the last input item should be coded "LAST", "X". Your input could look like this:

```
"Butler,J.S."   ,  "323W.76Street,N.Y.,10023"
"Cook,A.B."     ,  "3062Street,WestNewYork,N.J.,07094"
      .
      .
      .
"Smith,C.A."    ,  "228E.55Street,N.Y.,10022"
"Young,W."      ,  "3230165Street,Flushing,N.Y.,11358"
"LAST"          ,  "X"
```

After the records are entered, the first record in CLASS would look like this:



After the file has been created, if you decide to add more records, say for a new student who registers late, the WRITE FILE statement can be used to enter additional records. No RESET statement is necessary as with stream-oriented files; the WRITE FILE statement automatically positions an entry-sequenced file at its end, unless the file has been opened with the REUSE clause specified. In this case, records are written at the beginning of the file, writing over any existing records.

Retrieving Records

To retrieve a record, use the READ FILE statement, which is the record-oriented counterpart to the GET statement. The READ FILE statement could be written like this:

```
80    READ FILE USING 85 'CLASS', Y$, Z$
```

This statement reads in two values from a record in the file named CLASS, and assigns them to the variables Y$ and Z$. The corresponding FORM statement describes from which positions in CLASS the values are to be obtained. The FORM statement could be written like this to obtain the name and address information:

```
85    FORM POS1, C, POS26, C
```

It is not necessary to read the items in the same order in which they appear in the record. For example, if it is more convenient to read the address first, the statements could be written:

```
80    READ FILE USING 85 'CLASS', Z$, Y$
85    FORM POS26, C, POS1, C
```

Nor is it necessary to read *all* the items in a record. If you were interested only in obtaining name information, you could use this READ FILE and FORM combination:

```
40    READ FILE USING 45 'CLASS', J$
45    FORM  POS1, C
```

This combination might be helpful when you wish to insert test marks for each student. You could read through the file sequentially, obtain each student's record, print his name at the terminal for verification, and enter the corresponding mark.

The READ FILE statement, like the GET statement, can contain an EOF clause to transfer control when the end of the file is reached. In the program shown below, the READ FILE statement causes program control to branch to statement 100 at the end of the file, which is used to print a message.

This program shows how you can read each student's record to insert a test mark. The program also introduces the REWRITE FILE statement, which is used to update an existing record, and shows how OPEN and CLOSE statements can be used with record-oriented files.

```
 20   DIM J$25
 30   OPEN FILE 'CLASS' ALL HOLD
 40   READ FILE  USING 45 'CLASS', J$,  EOF 100
 45   FORM POS1, C
 50   PRINT J$
 60   INPUT G REM OBTAIN NEW MARK
 70   REWRITE FILE USING 75 'CLASS', G
 75   FORM POS 101, PIC(ZZ#)
 80   GOTO 40 REM TO OBTAIN NEXT RECORD
100   PRINT 'END OF FILE--LAST RECORD READ'
110   CLOSE FILE 'CLASS'
120   END
```

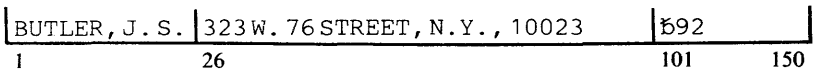
Statement 30 opens the file. ALL is a special keyword used with record-oriented files to indicate that *both* input and output operations can take place on the file. ALL is required if any rewriting operations are to take place. HOLD means that no other user can get to a record while you are updating it. There may be occasions when more than one user may have access to the same file. By specifying HOLD, you prevent other users from getting the particular record you are updating until you are finished with it.

Statements 40 and 45 obtain the name information from the file. Statement 50 prints the name at the terminal, allowing you to verify it and enter the corresponding test mark in statement 60. Statement 70 is the REWRITE FILE statement, which enters one data item into the record just read, the numeric variable G. Statement 75 says that the variable is to be entered beginning at position 101 of the record, in the format PIC(ZZ#), three digits with leading zeros suppressed. The remaining statements cause the program to cycle through all the records and close the file after the last record is handled.

Updating Records

The REWRITE FILE statement writes over all or part of a record. It can rewrite an existing data item, and can write new information into an unused portion of a record. In order to perform rewriting operations on a file, you must explicitly open the file with the OPEN FILE statement specifying ALL. To update records in an entry-sequenced file, you must read each record before rewriting it.

If the mark for BUTLER were 92, his record would look like this after the REWRITE FILE statement is executed:



Note that it is not necessary to enter data immediately after the last data item previously written in the record; that is, you can begin writing the marks in position 101 rather than in position 91. A rewritten record must be the same size as the original record in an entry-sequenced file.

The REUSE keyword on the OPEN FILE statement can be used to write over a file. By specifying OUT REUSE or ALL[HOLD] REUSE on the OPEN FILE statement subsequent WRITE FILE statements will place data at the beginning of a record-oriented file, writing over any existing records. Thus the REUSE clause enables you to use the same file space several times. For example:

```
10    OPEN FILE 'DAILY' OUT REUSE
20    OPEN FILE A$ ALL REUSE, EXIT 50
```

Statement 10 opens the file named DAILY as an output file, and the specification of REUSE indicates that file space can be used again. The file will be positioned at its beginning and any records that were on the file prior to the open will be written over.

Statement 20 opens the file A$. The ALL keyword means both input and output operations can take place on the file. REUSE means the file space will be used more than once.

Rereading Records

In the program shown above, when statement 40 reads a record, that record remains available until another record is read or written (or the end of the program is reached). Thus, the record is available for rereading operations as well as for rewriting operations.

Suppose the student C. A. Smith says that he is not receiving class mail that the other students are getting. He suspects his address may have been entered incorrectly. Since the address is not normally read in, you can use the

REREAD FILE statement to obtain this information. The REREAD FILE and FORM statements could be written like this:

```
100   REREAD FILE USING 105 'CLASS', K$
105   FORM  POS26, C
```

These statements retrieve the information starting at position 26, the address location.

The program below shows how these statements could be used to verify the address, and further demonstrates the use of the REWRITE FILE statement as well, to write over existing data.

```
 20   DIM J$25, K$65, L$65, C$1
 30   OPEN FILE 'CLASS' ALL HOLD
 40   READ FILE  USING 45 'CLASS', J$, EOF 200
 45   FORM  POS1, C
 50   PRINT J$
 60   INPUT G
 70   IF J$ = 'SMITH,C.A.' GO TO 100
 80   REWRITE FILE USING 85 'CLASS', G
 85   FORM  POS104, PIC(ZZ#)
 90   GO TO 40
100   REREAD FILE USING  105 'CLASS',K$
105   FORM  POS26, C
110   PRINT 'SMITH ADDRESS IS'; K$; 'TYPE R IF CORRECT'
120   INPUT C$
130   IF C$ = 'R' GO TO 80
135   PRINT 'TYPE IN CORRECT ADDRESS'
140   INPUT L$
150   REWRITE FILE USING 155 'CLASS', L$, G
155   FORM  POS26, C, POS 104, PIC(ZZ#)
160   GO TO 40
200   PRINT 'END OF FILE--LAST RECORD READ'
210   CLOSE FILE 'CLASS'
220   END
```

This program reads each student's name, prints it at the terminal, and enters the next test mark into position 104 of each record. The program also checks for Smith's name, and, upon encountering it, branches to statements 100 through 160. Statement 100 rereads the record, obtaining the address information. Statement 110 prints the address for verification. Statement 120 requests a code, the character R if the address is right, any other character if the address if wrong. Statement 130 tests that code and, if the address is right, continues processing other records with no further action. Otherwise, statement 140 requests the correct address, which statement 150 writes into the record, replacing the incorrect address.

As with rewriting operations, you must read a record before you can reread it.

Opening, Closing, and Repositioning Files

Record-oriented files, like stream-oriented files, can be opened and closed implicitly or explicitly. A file is opened implicitly for input by the first occurrence of a READ FILE statement and for output by the first occurrence of a WRITE FILE statement if it has not been previously opened explicitly. It is opened explicitly through the OPEN FILE statement. As you may recall, for stream-oriented files, OPEN is specified with the keywords IN for input or OUT for output. Record-oriented files can be opened in the same way, and, in addition, can be opened for *both* input and output if ALL is specified. Statement 30 in the preceding program shows this use of the OPEN statement.

CLOSE FILE is used in exactly the same way for record-oriented files as CLOSE is used for stream-oriented files. It closes the file, or if the statement is not present, the system closes the file at the end of program execution.

Inclusion of the REUSE keyword on the OPEN FILE statement positions a record-oriented file at its beginning so that the file can be reused by subsequent WRITE FILE statements.

The RESET FILE statement may also be used to reposition a file to its beginning.

The maximum number of concurrently active input/output files is 15.

Using the EXIT Statement

The section "Stream-Oriented Files" introduced the EXIT statement, to specify the transfer of control in the event of certain error conditions. The EXIT statement can be used for record-oriented files as well. An EXIT statement for an entry-sequenced file could look exactly the same as for a stream-oriented file:

```
80   EXIT EOF 100, IOERR 150, CONV 200
```

If you use the EXIT statement, remember to include an EXIT clause, specifying the statement number of the EXIT statement, on each appropriate input/output statement.

A Key-Sequenced File

The same VS BASIC statements are used for key-sequenced and entry-sequenced files, with one additional statement available for key-sequenced files. The DELETE FILE statement is used to erase a record. Records in an entry-sequenced file cannot be deleted. An entry-sequenced record that is no longer needed can be written over with blanks, zeros, or other characters, utilizing the REWRITE FILE statement. Also, by using the REUSE keyword on the OPEN FILE statement, a file can be positioned to its beginning, so that the file can be used again.

To compare how BASIC statements are used, let us go back and create a key-sequenced file for the class records. One of the differences between a key-sequenced file and an entry-sequenced one is that before the file is created you must tell your installation personnel the position and length of a key within a record. The key can be constructed of any number of characters up to 255, and must be a character value. It can contain, or can consist completely of, digits, as in a charge account number or an employee serial number. In the key-sequenced file for class records, a logical choice for a key would be the name of the person. Assume, therefore, that you have the installation create a key-sequenced file having a maximum size of 150 positions, with the key in the first 25 positions. To avoid confusing this file with CLASS, let's call it GRADE.

Entering Records

Another difference with a key-sequenced file is that no matter how you enter records, they will be stored according to key; you do not have to sort them beforehand. In fact, because the person's name is being used as the key in GRADE, these particular records will be sorted into alphabetic order by the computer.

To enter records into GRADE, use the WRITE FILE statement in the same manner as for entry-sequenced files:

```
50   WRITE FILE  USING 55 'GRADE', N$, A$
```

One of the data items in the output list must be used to write the key. In this example, the key is the name field, N$.

The FORM statement could look like this:

```
55   FORM  POS1, C, POS26, C
```

The statement tells the computer to enter the first variable, (N$), beginning in position 1 of the record, and then enter the second variable, (A$), beginning in position 26.

A key-sequenced record, unlike an entry-sequenced one, can be made larger later (up to the maximum record size). Thus, it is unnecessary in this FORM statement to specify X60 to reserve space for later use.

When formatting the key field, you should exercise care in putting the key into the proper position in the file. For purposes of simplicity, these examples use the first 25 record positions for the key. The occasion may arise, however, when you might have a file with the key in some other positions. By careful use of POS, you can ensure that the key will be properly inserted. Also, you can use the intrinsic function KPS(filename), to find the position, relative to 1, of the start of an embedded key in the file named *filename,* and you can use the intrinsic function KLN(filename) to find the length of the key.

After the records are entered into GRADE, additional records can be added and will be stored in key-sequenced order. The records in GRADE can be retrieved directly or sequentially with the READ FILE statement.

Retrieving Individual Records

To retrieve a particular record, specify the KEY clause in the READ FILE statement. If you wanted Smith's record, you would specify his name in the clause:

```
70 READ FILE USING 75 'GRADE', KEY='SMITH,C.A.', F$, G$
```

The computer will search for the record whose key matches, then will read the values from the record into the variables F$ and G$.

If, however, the computer cannot find the key, it automatically terminates program execution unless you instruct it to take alternative action. For example, if you enter the key incorrectly (say you spell the name SMIHT), the match would never be found. To protect program execution, include the NOKEY clause on the READ FILE statement:

```
70   READ FILE USING 75 'GRADE', KEY='SMITH,C.A.',
     F$, G$, NOKEY 200
```

The NOKEY clause tells the computer that if the matching key cannot be found, the program should branch to statement number 200. The NOKEY clause for key-sequenced files is similar to the EOF clause for entry-sequenced files; it permits you to retain control of program execution if a particular condition arises. It is a good idea to code the NOKEY clause on any statement that matches keys.

The READ FILE statement shown above will of course search only for the key "SMITH, C. A.". To provide flexibility, you can specify a character variable in the KEY clause, and assign the character string to the variable

through other means, such as an INPUT statement. This READ FILE statement could be used to search for any character string assigned to the variable K$:

```
70   READ FILE USING 75 'GRADE',KEY=K$,F$,G$,NOKEY 200
```

Retrieving Records Sequentially

To read records sequentially, write the READ FILE statement without the KEY clause. Such a statement is the same as a READ FILE statement for an entry-sequenced file. For example,

```
90   READ FILE USING 95 'GRADE', Y$, Z$, EOF 210
```

In effect, you are telling the computer not to look for any key, just retrieve the next record in the file.

Using this statement, you can read the entire file from beginning to end as a sequential file, or can begin reading it sequentially from some point in the file after having searched particular keys with a READ FILE statement having the KEY clause.

Updating Records With and Without Keys

The REWRITE FILE statement can be used to update records in a sequential manner in the same way as for entry-sequenced files, that is, by rewriting each record after it is read. No KEY clause is required. If a KEY clause is specified, the REWRITE FILE statement can update a particular record.

If you needed to update Smith's address, you could do it quite simply in a key-sequenced file by using these statements:

```
40   DIM N$25, A$65
50   INPUT N$, A$
60   REWRITE FILE  USING 65  'GRADE', KEY=N$, A$,
     NOKEY 80
65   FORM  POS26, C
70   STOP
80   PRINT 'NO MATCH FOUND'
90   END
```

In statement 50 you enter the name and the correct address. Statements 60 and 65 replace whatever address was in the record with the new address. Statement 80 informs you if no match was found.

Note that if the KEY clause is used in the REWRITE FILE statement, no READ FILE statement is required to retrieve the record first. If the KEY clause is specified in the REWRITE FILE statement, the record matching that key is brought in from the file; thus, the REWRITE FILE statement with a KEY clause retrieves, as well as rewrites.

The REWRITE FILE statement can write over existing data or unused portions of a record, but must not change the contents of the field containing the key information. Unlike records in entry-sequenced files, a rewritten record in a key-sequenced file can be made larger, but not smaller, than the original record. Fields not written over remain unchanged.

As another example of REWRITE FILE, assume that during the school term you give the students an extra credit project; their final grade will be raised by five to ten points depending on the quality of their work. Before the end of the term, you add in the extra credit for those students who handed in the

project. The short program below illustrates how the REWRITE FILE statement can be useful in updating the records.

```
 90  DIM N$25
100  PRINT 'ENTER STUDENT'S NAME AND EXTRA CREDIT MARK'
110  INPUT N$, E
120  IF N$='LAST' GO TO 170
130  REWRITE FILE USING 135 'GRADE', KEY=N$, E,
     NOKEY 150
135  FORM  POS140, PIC(Z#)
140  GOTO 110
150  PRINT 'NO MATCH FOUND FOR', N$
160  GOTO 110
170  END
```

Statement 100 prompts you for input information. Statement 110 accepts the student name in N$ and the mark in E. Statement 120 tests whether the end of input has been reached; assume the last input data item should have the word LAST as the student's name. Statement 130 enters the mark recorded in E into the file after the key has been matched with the name in N$. Statement 135 formats the mark into positions 140 and 141.

Using Generic Keys

The character string specified in a KEY clause does not have to be the same length as the matching key in a record. The computer searches a record key for the characters you specify in the KEY clause, starting at key position 1. Therefore, you can specify a key containing fewer characters than a full key. This is called a *generic key.*

To illustrate, assume that a file named CITIES has these keys:

```
AUSTINbbbbbb
BIRMINGHAMb
BOISEbbbbbb
BOSTONbbbbb
MINNEAPOLIS
NEWbORLEANS
SACRAMENTOb
```

The following statement uses the string 'BO' as a generic key to search the file named CITIES.

```
110  READ FILE USING 130 'CITIES', KEY='BO', P$, NOKEY 200
```

The computer will search for the first record whose key has 'BO' as its first two characters. In the file above, both BOISE and BOSTON start with these characters condition. Since the search is made according to the EBCDIC collating sequence, the computer will pick BOISE. (Refer to Appendix B. "Collating Sequence of the BASIC Character Set" for a list of the EBCDIC collating sequence.)

Here's another example:

```
100  READ FILE USING 130 'CITIES', KEY='L', P$, NOKEY 200
```

In this example the generic key is 'L'. Since there is no key in the file which starts with 'L', no match is found, and the branch specified in the NOKEY clause will be taken.

In addition to the equal sign (=) or .EQ. in a KEY clause, you can also specify the relational operator 'greater than or equal to' (>= or ≥ or .GE.). For example:

```
110 READ FILE USING 130 'CITIES', KEY .GE. 'L',
    P$, NOKEY 200
```

The generic key is once again 'L', and again there is no match for it. But the relational operator tells the computer that if there is no equal, it is to choose the next key in the EBCDIC collating sequence that is greater than the one specified. In the file above, it is MINNEAPOLIS.

Rereading Records

The REREAD FILE statement does not specify the KEY clause, since the record to be reread is already available. The REREAD FILE statement would be written exactly the same for a key-sequenced file as for an entry-sequenced one.

Deleting Records

Records in a key-sequenced file can be erased by using the DELETE FILE statement, and specifying the key of the record to be deleted. For example:

```
90  DELETE FILE 'GRADE', KEY=N$, NOKEY 130
```

This statement would delete the record whose key matched the character value in N$, or would branch to statement 130 if the key could not be matched.

Repositioning Files

The RESET FILE statement can reposition a file to its beginning. If RESET FILE contains a KEY clause, the file will be repositioned to the particular record associated with that key.

Inclusion of the OUT REUSE or ALL REUSE clause on the OPEN FILE statement positions a record-oriented file at its beginning so that the file can be reused by subsequent WRITE FILE statements.

Key Clauses on the EXIT Statement

For key-sequenced files, the EXIT statement can specify these clauses in addition to the other clauses available:

- NOKEY, to transfer control if no key satisfying a KEY clause can be found,
- DUPKEY, to transfer control if a key specified for a new record already exists in a file.

An EXIT statement specifying error handling and KEY clauses could look like this:

```
180 EXIT EOF 300,IOERR320,CONV 350,NOKEY 130,DUPKEY 200
```

When using the EXIT statement, remember to include an EXIT clause on each appropriate input/output statement. For example, to refer to the EXIT statement above, the DELETE FILE statement previously illustrated could be written:

```
90  DELETE FILE  'GRADE', KEY=N$, EXIT 180
```

A Relative Record File

Records in a relative record file are fixed length and each record is identified by its position relative to the start of the file. That is, relative-record number 5 refers to the fifth record in the file. The number is used to directly access records in the file. Records in a relative-record file may or may not contain data. When a record does not contain data, it is called a null record.

'Fixed length' means that, when the file is created, the same amount of space is allowed for each record to be written into the file. A record does not necessarily have to use all of the space allowed.

However, once you've written a short record into the space (that is, one that does not use all of the space) you cannot extend that record—that is, if you update the record by rewriting it, the rewritten record cannot be longer than the previous record written.

Entering Records

Records in a relative record file are stored in sequence according to the relative record number you specify when you write them to the file.

To enter a record, you use the WRITE FILE statement containing a REC clause.

```
20  WRITE FILE 'GRADE', REC=5, A$,N$, DUPREC 100
```

This statement causes data from the variables A$ and N$ to be entered into relative record number 5 of the file GRADE. The DUPREC clause transfers program control to statement 100 if a duplicate relative record is encountered. That is, relative record 5 already exists..

Retrieving Records

To retrieve a particular record, you specify the REC clause in the READ FILE statement. For example:

```
50  READ FILE 'ACCT' , REC=5, X$, Y$
```

This statement retrieves relative record 5, then reads the values into the variables X$ and Y$.

If the relative-record number cannot be found, program execution terminates unless you instruct it to take alternative action. For example, if you specified a relative record that did not exist in the file, the REC condition could not be satisfied. By including the NOREC clause on the READ FILE statement, you retain control of program execution. For example:

```
50  READ FILE 'ACCT', REC=5, X$, Y$, NOREC 180
```

The NOREC clause causes the program to branch to statement number 180 if the relative record specified cannot be found.

To read records sequentially, you use the READ FILE statement without the REC clause. By so doing, you will retrieve the next non-null record in the file in relative-record number sequence.

Updating Records

To update records, you use the REWRITE FILE statement. The REWRITE FILE statement with the REC clause updates a particular relative record. A REWRITE FILE without a REC clause must be preceded by a READ FILE statement with a REC clause.

The length of the new record must not be greater than the length of the original record.

Rereading Records

The REREAD FILE statement without the REC clause is used to reread a record. The REC clause is not specified because, as with entry- and key-sequenced files, the record to be read is already available. The REREAD statement is written in the same format as for entry- and key-sequenced files.

Deleting Records

Records in a relative-record file are removed by using the DELETE FILE statement with a REC clause specifying the relative-record number of the record to be deleted.

```
80  DELETE FILE 'GRADE', REC=5, NOREC 120
```

This statement deletes the record whose relative record number is 5. If no match is found, the program branches to statement 120.

Repositioning Files

The RESET FILE statement without a REC clause repositions a file to its beginning. If the RESET FILE statement contains a REC clause, the file is positioned to the particular record specified in the REC clause.

Error Clauses in the EXIT Statement

For relative record files, the EXIT statement can also specify the following additional clauses.

- NOREC to transfer control if a matching relative record is not found when reading a file.
- DUPREC to transfer control if a record with the same relative record number as a new record already exists on a file when writing to a file

An EXIT statement specifying error handling clauses could look like this:

```
180 EXIT EOF 300, IOERR 320, CONV 350, NOREC 130, DUPKEY 200
```

Remember to include an EXIT clause, specifying the statement number of the EXIT statement, on each appropriate input/output statement.

&REC Internal Variable

The internal variable &REC is available for use with relative-record files. This variable contains the number of the last record referred to successfully (that is, without causing a DUPREC, NOREC or any other input/output error.)

The FORM Statement—Differences Between PRINT and Record I/O

The version of the FORM statement used with record-oriented files is similar to that used with PRINT USING in the following ways:

1. Both contain the C character specification.
2. Both contain the PIC numeric specification, with the same digit specifiers and insertion characters.
3. Both contain the format control specifications X and POS.
4. Both can contain the SKIP control.
5. Both can contain literals.

The two versions are different in the following manner:

Under record-oriented I/O, numeric data can be formatted using other specification codes besides PIC. Additional specification codes are:

```
NC
PD
B
S
L
```

NC, PD and B are used to store and retrieve numeric data in special internal formats, and are not generally used in most BASIC applications. Except for one use of NC, they are not further discussed here; additional material on these codes can be found in Part II of this publication under "The FORM Statement."

The NC Specification

The one use of NC applicable to this discussion is in its relationship to PIC. PIC can be used only in output operations; thus, it can appear in FORM statements related to WRITE FILE and REWRITE FILE statements, but not in those related to READ FILE or REREAD FILE statements. To read data that was written using PIC, NC is used, specifying the number of positions in the record to be read. For example,

```
NC4
```

would read four positions of a number.

If a number were written using this PIC specification:

```
PIC(###) or PIC(ZZ#)
```

the NC specification to retrieve it would be:

```
NC3
```

To retrieve only the first two of these digits, you would specify NC2.

Earlier, this FORM statement was used to enter the two-digit numeric variable E into the file called GRADE:

```
135      FORM  POS140, PIC(Z#)
```

To retrieve that value, you could use this FORM statement:

```
55  FORM POS140, NC2
```

NC can also specify the number of decimal digits in a number, in the following manner:

```
NC5.2
```

This specification says that five positions are to be read, and a decimal point is to appear before the two rightmost digits. That is, the five positions could look like this:

```
 12.34   would be read as      12.34
 1.234   would be read as      12.34
 11234   would be read as     112.34
```

If an item were written using this PIC specification,

```
PIC(####.##)
```

The NC specification to retrieve it would be:

```
.NC7.2
```

The first number specified in NC is the field width, that is, the total number of characters to be read, including digits, decimal points, commas, dollar signs, etc. The second number is the number of decimal digits. The following are examples of how PIC and NC can be used in combination:

If PIC were specified:	**NC would be specified:**
`PIC(###.##)`	`NC6.2`
`PIC(ZZZ.##)`	`NC6.2`
`PIC($$,$$$.##)`	`NC9.2`
`PIC(ZZBZZBZZ)`	`NC8  or NC8.0`

The S and L Specifications

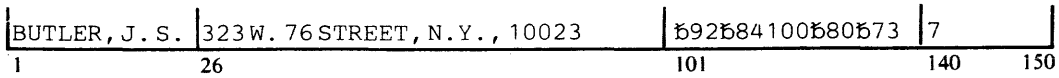
The specification S indicates that an item in a record is in short-form precision. A number in short-form precision takes up four positions in a record. If S is specified for an input operation, the value in the record is moved to the variable specified in the READ FILE or REREAD FILE statement; if the program is in long-form precision, such a value is extended to long-form. If S is specified for an output operation, a short-form value is written from the variable specified in the WRITE FILE or REWRITE FILE statement into the record.

The specification L indicates long-form precision and is the long-form counterpart to the S specification. A number in long-form precision takes up eight positions in a record.

For an input operation, the value in the record is moved to the variable specified in the READ FILE or REREAD FILE statement; if the program is in short-form precision, long-form items are truncated to short-form before being used. For an output operation, a long-form value is written into the record from the variable specified in the WRITE FILE or REWRITE FILE statement.

A Last Example

Continuing with the example for class records after all the marks for five tests and the extra credit for the project have been entered into the file GRADE, the first record in the file could look like this:



If you wanted to print the final mark and the honors status, you could use this program:

```
 10  DIM G(5), M$1, N$25
 20  PRINT USING 25,'NAME', 'FINAL MARK', 'HONORS'
 25  FORM  POS6, C, POS35, C, POS50, C, SKIP2
 30  OPEN FILE 'GRADE' ALL HOLD
 50  READ FILE USING 55 'GRADE', N$, MAT G, E, EOF 110
 55  FORM  POS1, C, POS101, 5*NC3, POS140, NC2
 60  A = SUM(G)/5 + E
 65  IF A> 100 THEN A = 100
 70  IF A > = 90 THEN M$ = '+' ELSE M$ = ' '
 80  PRINT USING 85,N$, A, M$
 85  FORM  POS6, C, POS35, PIC(ZZ#.#), POS50, C, SKIP1
 90  REWRITE FILE USING 95 'GRADE', A, M$
 95  FORM  POS 130, PIC(ZZZ.Z), POS135, C
100  GO TO 50
110  END
```

Statement 10 defines an arithmetic array, G, with five members, a character variable, M$, with one character, and a character variable, N$, with 25 characters. The array G is to hold the five marks for each student, M$ is to hold the honors character, either a + or a blank, and N$ is the name field.

Statements 20 and 25 format a printed heading. Statement 30 opens the file for input, output, and updating operations.

Statement 50 reads the file according to the format shown in statement 55. Remember that although GRADE is a key-sequenced file, its records can be read in sequential order if the KEY clause is not specified. From statement 55 we can determine that the items being retrieved are the name, placed into N$, five sequences of three digits (the five marks beginning in position 101), placed into the array G, and a two-digit number for extra credit, placed into E.

Statement 60 sums the five marks, divides the sum by 5 to find the average, then adds in the extra credit recorded in E, and puts the resulting value into A.

Statement 65 reduces any mark that exceeds 100.

Statement 70 analyzes the value of A. If the value equals or exceeds 90, a plus sign, indicating honor student, is placed into M$. If the value of A is less than 90, M$ is assigned a blank.

Statements 80 and 85 print the student's name (N$), the final mark (A), and the honors code (M$).

Statements 90 and 95 enter the final mark and the honors code into the record, beginning in positions 130 and 135, respectively.

Statement 100 branches back to statement 50 and the next record is read. After all records are read, the program ends.

Output from this program could be the following:

```
Print
Position  6              35              50
          NAME           FINAL MARK      HONORS
          BUTLER, J. S.  92.5            +
          COOK, A. B.    82.0
                .
                .
                .
          SMITH, C. A.   84.0
          YOUNG, W.      97.0            +
```

Summarizing Record-Oriented Statements

All three types of record-oriented files, entry-sequenced, key-sequenced, and relative-record, can be accessed sequentially. Key-sequenced and relative-record files can also be accessed directly.

The OPEN FILE statement explicitly opens a record-oriented file. If IN is specified, the file is opened for input; if OUT is specified, it is opened for output; if ALL is specified, it is opened for both operations.

The WRITE FILE statement writes a record. In an entry-sequenced file, each record is stored in the order in which it is entered. In a key-sequenced file, each record is entered at a point determined by the key, which is one of the fields within the record. In a relative-record file, each record is stored according to its relative position, specified by the REC clause.

The READ FILE statement reads a record. In an entry-sequenced file, each record is read in sequential order. In a key-sequenced file, each record is read in sequential order if no KEY clause is specified in the statement. If KEY is specified, the record having a matching key is read. In a relative-record file, each non-null record is read in sequential order if the REC clause is not specified in the statement. If the REC clause is specified, the record in the relative position specified by the relative-record number is read.

The REREAD FILE statement makes the last record previously read available again, regardless of whether the record was read in sequential order, by key, or by relative-record number.

The REWRITE FILE statement alters an existing record, provided that the file was opened with the OPEN FILE statement specifying ALL. In an entry-sequenced file, the last record read is altered. In a key-sequenced file, the last record read is altered if no KEY clause is specified in the statement. If KEY is specified, the record having a matching key is read and then rewritten. In a relative-record file, the last record read is altered if the REC clause is not specified in the statement. If REC is specified the record with the matching relative-record number is read and rewritten.

The RESET FILE statement repositions a file to its beginning. In a key-sequenced file, if a KEY clause is specified, the file will be repositioned to the particular record associated with that key. In a relative-record file, if a REC clause is specified, the file is repositioned to the particular record indicated by the REC clause.

The DELETE FILE statement deletes a record from a key-sequenced or relative-record file. For key-sequenced files, the KEY clause is required in order to identify the record being deleted. For relative-record files, the REC clause is required to identify the record being deleted.

The EXIT statement specifies the statement number to which control should be given if a particular input/output error occurs. The error keywords that can be written in the statement are EOF, IOERR, CONV, NOKEY, DUPKEY, NOREC, and DUPREC.

The CLOSE FILE statement explicitly closes a record-oriented file.

The FORM statement specifies the format of fields in record-oriented files.

Program Error Handling

Many errors can occur during the execution of your program. VS BASIC provides several ways for you to receive program control when these error conditions occur.

- Use of the ON statement with its associated clauses
- Use of the error clauses on I/O statements (discussed earlier)

With the ON statement, you can specify: (1) where program control should be transferred if certain error conditions occur, using the GOTO or THEN clause, or the statement number; (2) that the printing of certain error messages be suppressed, using the IGNORE clause; and (3) that the system default be taken, using the SYSTEM clause.

The arithmetic and attention interrupts (to suspend program execution) and input/output errors that can be specified in the ON statement are:

OFLOW (overflow)
: This condition is raised whenever an arithmetic operation would result in a value greater than the maximum machine value.

UFLOW (underflow)
: This condition is raised whenever an arithmetic operation would result in a value less than the minimum machine value.

ZDIV (zero divide)
: This condition is raised whenever division by zero is attempted.

ATTN (attention)
: This condition is raised whenever the attention key or equivalent is pressed, and allows the program to specify a statement where program control can optionally be transferred.

INERR (input errors)
: This condition allows control to be transferred to a designated statement if input errors occur.

ERR (error)
: Any error not handled by an ON statement with a specific error clause or by an error clause on an I/O statement.

Once the ON statement is executed and becomes active, it applies to the execution of all subsequent statements in your BASIC program except those within a user-defined function (a user-defined function, however, can have its

own ON statement). The ON statement remains active until program execution terminates or another ON statement that refers to the same error condition is encountered.

The internal variables &CODE, &ERR, &FILE and &LINE can also help you identify errors and where they occur. They can be used in conjunction with the ON statement, and with error clauses on input/output statements and other BASIC statements.

At the beginning of program execution, the internal variables &CODE, &ERR, and &LINE contain the value of zero; &FILE contains blanks. When an error occurs in your program, the internal variable contains the value associated with the error. This is the value that gives you information about the error that occurred. This value is retained until another error occurs.

The following example use the ON statement for arithmetic overflow or division by zero:

```
 10        DIM B(10), C(10)
 20        MAT READ B
 30        DATA 2,0,0,4,5,6,0,8,0,2
 40        ON ZDIV GOTO 120
 50        FOR J=1 TO 2
 60        FOR I=1 TO 10
 70        C(I) = 20/B(I)
 80        NEXT I
 90        ON ZDIV SYSTEM
100        NEXT J
110        STOP
120        PRINT 'THE 'I' VALUE OF B IS ZERO'
130        GO TO 80
140        END
```

The program produces the following results:

```
THE        2 VALUE OF B IS ZERO
THE        3 VALUE OF B IS ZERO
THE        7 VALUE OF B IS ZERO
THE        9 VALUE OF B IS ZERO
ICD413     LINE70: DIVISION BY ZERO
ICD413     LINE70: DIVISION BY ZERO
ICD413     LINE70: DIVISION BY ZERO
ICD413     LINE70: DIVISION BY ZERO
```

In this example, statement 40 sets up a procedure for the error condition division by zero. When statement 70 is executed, as directed by the ON statement on line 40, control goes to line 120 each time a division by zero occurs.

The ON statement at line 90 illustrates one ON statement superseding another ON statement. In this case, the subsequent ON statement directs that the action on division by zero be changed to that of normal system action (printing of an error message each time division by zero takes place and setting the result to zero).

PART II. THE VS BASIC LANGUAGE

- Syntax Definition
- Statements
- The BASIC Character Set
- Data Representations
- BASIC Statements

Syntax Definition

To assist you in using Part II, the 'Syntax Definitions' that follow describe certain syntactical elements of the language that apply to a number of language statements. For example, an 'input-list' can appear in a GET, INPUT, READ, READ FILE, or REREAD FILE statement. To avoid duplication, the items that can make up the input-list are described here instead of being scattered throughout.

input-list: *v* $_1$[,*v* $_2$]...

where *v* is any of the following:

- A scalar variable
- Any array member reference
- A pseudo-variable
- An array reference (that is, MAT *array-name*)
- An array reference with redimensioning (that is, MAT *array-name* ([*exp* $_1$[*exp* $_2$]]))

output-list: *exp* $_1$[,*exp* $_2$]...

where *exp* is either of the following:

- A scalar expression
- An array reference (that is, MAT *array-name*)

redimension specification: *exp* $_1$[*exp* $_2$]

where *exp* $_1$ is an arithmetic expression representing the number of members in a one-dimensional array or the number of rows in a two-dimensional array, and *exp* $_2$ is an arithmetic expression representing the number of columns in a two-dimensional array.

pseudo variable: STR (*x,y,z*)

where *x* is a character variable or array member reference, and *y* and *z* are arithmetic expressions, whose meanings are the same as for the STR intrinsic function.

Syntax Notation

The following conventions are used for syntax notation in this book:

Stacking of items	a b c	or separation of items by a vertical

bar a | b | c indicates that one and only one of the items in the group may be specified.

Brackets [] enclose an item or group of items that is optional; either one item may be specified or none may be specified.

Braces {} enclose a group of items that is not optional; one of the items in the group must be specified.

The ellipsis ... indicates that the item or group of items immediately preceding it can be repeated any number of times (subject to physical limitations, such as line length).

Uppercase letters and punctuation marks other than those described above represent information that must appear exactly as shown.

Lowercase letters represent information that must be supplied by the user.

Statements

A BASIC program consists of a group of numbered statements. There are two types of statements: executable and nonexecutable.

Executable statements are those that specify a program action such as assigning a value to a variable (the LET statement), printing a value (the PRINT statement), or directing the order of program flow (the GOTO statement).

Nonexecutable statements are those that specify information necessary for program execution. The DATA statement, which provides values to be used, and the DIM statement, which specifies the size of data arrays, are typical nonexecutable statements.

Executable and nonexecutable statements may be intermixed when a BASIC program is entered. The maximum number of statements permitted in a single BASIC program is 1000.

Statement Numbers

Each statement in a BASIC program must be preceded by a statement number. A statement number cannot contain more than five digits.

The BASIC Character Set

The characters that have syntactic meaning in BASIC fall into three categories: alphabetic, numeric, and special characters. All elements that make up a BASIC program are constructed from characters in these three categories, with the exception of comments and character constants, either of which can contain any character permitted by the machine configuration on which the BASIC program is processed.

Alphabetic Characters

The alphabetic characters in BASIC are the upper- and lower-case letters of the standard English alphabet (A-Z and a-z) and the following three characters, called the *alphabet extenders*:

@	(the commercial "at" sign)
#	(the number or pound sign)
$	(the currency symbol)

Corresponding upper- and lower-case letters of the standard alphabet are evaluated identically and may be used interchangeably; however, the characters on the same keys as the alphabet extenders are treated differently from the extenders and may not be used in their places. Thus, the symbols A2 and a2 are equivalent, while the symbols $$ and 4$ (where the digit 4 is the lower-case character corresponding to the upper-case $) are not equivalent.

When BASIC is used with languages other than English, the three alphabet extenders can be used to cause printing of letters that are not in the standard alphabet. In such instances, the internal—or EBCDIC—representation of the added character is the same as that of the alphabet extender that it replaces in printing.

Numeric Characters

The numeric characters in BASIC are the digits 0 through 9.

Special Characters

There are twenty-four special characters in BASIC:

Character	Name
	Blank
=	Equal sign or assignment symbol
+	Plus sign
-	Minus sign
*	Asterisk or multiplication symbol
/	Slash or division symbol
↑	Up-arrow or exponentiation symbol
(	Left parenthesis
)	Right parenthesis
,	Comma
.	Point or period
'	Single quotation mark
"	Double quotation mark
;	Semicolon
:	Colon
|	OR sign or vertical bar
&	AND sign or ampersand
!	Exclamation symbol
?	Question mark
>	Greater than symbol
<	Less than symbol
≠	Not equal symbol
≤	Less than or equal to symbol
≥	Greater than or equal to symbol

Certain special characters may be combined to produce other syntactic forms in BASIC, of which the following combinations are examples:

Symbol	Meaning
>=	Greater than or equal to
<=	Less than or equal to
<>	Not equal
**	Exponentiation
||	Concatenation

Use of Blanks

Blanks can be used freely throughout a BASIC program to improve readability. They have no syntactic meaning except within character constants and in the DATA and Image specification statement. Thus, all of the following statements are effectively the same: that is, the integer value 25 is assigned to an arithmetic variable named A2:

```
LET A2 = 25
LETA2=25
LET A 2 = 2 5
L ET A2 = 25
```

In a character constant, the blank is considered to be one of the characters. The constants shown below are not the same:

```
LET A$ = 'MAN Y'
LET A$ = 'MANY'
```

Use of Tab Characters

If your terminal has a tabulation capability, you can use the tab key to enter a tab character. The primary uses of the tab character are to improve readability of, and to speed up, the listing of a source program.

During program entry, the tab can be used (instead of spaces) to move the carrier to the right, one or more positions. Then your program statements are written using this positioning to indent groups of statements, such as those within a FOR - NEXT loop.

When listing a program, the tabulation is used. Since it is faster to move the carrier ten positions with a single tab than it is to space the ten positions, listing is accelerated if the program contains tabs.

A user may therefore, obtain program listings faster, and in a more readable form by using this feature.

Data Representation

The BASIC character set can be used to represent either arithmetic or character data.

Arithmetic Data

Arithmetic data items are those having a numeric value. All numbers in BASIC are expressed to the base ten; that is, they are treated as decimal numbers.

Magnitude

The magnitude of a number is its absolute value. The range of absolute numbers permitted in VS BASIC programs is 0 or in the range of approximately 10^{-78} through 10^{75}.

Arithmetic Precision

In BASIC, the precision of an integer or fixed-point number is the maximum number of digits it may contain. The precision of a floating-point number is the number of digits in the number to the left of the E (see format descriptions below).

VS BASIC supports two levels of arithmetic precision, designated *short form* and *long form.* The level of precision for the two forms is seven significant decimal digits and fifteen significant decimal digits, respectively.

The BASIC user specifies the level of precision (long or short form) under which his program is to be run by a command language statement entered when the program is compiled, or by specifying the precision in the OPTION statement. When the OPTION statement is used, it must be the first statement in your source program.

Arithmetic Data Formats

Arithmetic data may be entered or printed in any of three formats: integer, fixed-point, or floating-point. The appropriate format for a given number depends on its magnitude and the level of arithmetic precision required by the user.

Numbers in any format can be either positive or negative. Negative numbers must be preceded by a minus sign. A plus sign before positive numbers is optional; when no sign is specified, the number is treated as a positive number. The three formats are defined as follows.

Integer Format

Numbers expressed in integer format are written as an optional sign followed by one or more digits.

Examples of numbers in integer format are:

```
0      +2      -23      2683
```

Fixed-Point Format

Numbers expressed in fixed-point format are written as an optional sign, followed by one or more digits and a single decimal point. The decimal point may appear before, after, or among the digits.

Examples of numbers in fixed-point format are:

```
33.      33.00        -.3        +3.56
```

Floating-Point Format

Numbers expressed in floating-point format are written as an integer or fixed-point number followed by the letter E and an optionally signed one- or two-digit exponent.

The value of a floating-point number is equal to the number to the left of the E, multiplied by ten to the power represented by the number to the right of the E. This notation corresponds to standard scientific notation in which values are expressed as numbers multiplied by powers of ten. For example, the value 10^1 can be expressed as 1E7 in BASIC floating-point format. The integer or fixed-point number—in this example, the integer 1—cannot be omitted.

Examples of numbers in floating-point format are:

Floating-Point Number	Equivalent Decimal Value
.25E-4	.000025
+1.0E+5	100000
5E-7	.0000005
-15.33E6	-15330000

Arithmetic Constants

An arithmetic constant in a BASIC program is either an integer, fixed-point, or floating-point number whose value is never altered during execution of the program.

Thus, the integer 1 is a constant in the following statement:

```
75    LET X = X + 1
```

Internal Constants

An internal constant is an arithmetic constant whose value is pre-defined by the BASIC language processor. Unlike other arithmetic constants, internal constants are referred to by names, though like normal constants, their values are never altered during program execution. The internal constants are listed in Figure 1.

Arithmetic Variables

A variable is a named data item whose value is subject to change during execution of the program.

Arithmetic variables are named by a single letter of the extended alphabet, or by a letter followed by a single digit. Examples of such names are: A, A2, #, and #3. As stated in the section "Alphabetic Characters," the variables A and A2 can also be referred to by the symbols a and a2.

When a BASIC program is executed, the initial value of all arithmetic variables is set to zero.

Constant	Name	Short-Form Value	Long-Form Value
π	&PI	3.141593	3.14159265358979
Base of Natural Logs	&E	2.718282	2.71828182845905
Square Root of Two	&SQR2	1.414214	1.41421356237309
Centimeters per inch	&INCM	2.540000	2.54000000000000
Kilograms per pound	&LBKG	.4535924	.453592370000000
Liters per gallon	&GALI	3.785412	3.78541178400000

Figure 1. Internal Constants

Character Data

Character data in BASIC is any data not having an arithmetic value. Like arithmetic data, character data may be handled in the form of constants or variables.

Character Constants

A character constant is a string of characters enclosed in a pair of single or double quotation marks. Any EBCDIC character may appear in a character constant, including digits and characters that are not part of the BASIC character set. Thus, the following are all valid character constants:

```
"ABCDEF"
'1234567'
"a%%345"
'ABC'
```

The length of a character constant is defined as the total number of characters it contains, including blanks, but excluding the delimiting quotation marks. The maximum number of characters permitted in a single character constant is 255.

If single quotation marks are used to enclose a character constant, a single quotation mark within the constant is represented by two consecutive single quotation marks. Unless this procedure is followed, the contained quotation mark will be recognized as the end of the constant. The same procedure is required for character constants bounded by double quotation marks and containing double quotation marks. In neither case is the extra quotation mark considered part of the length of the character string. (Note that quotation marks not enclosing any characters cause 18 blanks to be generated.) The following are some examples of how quotation marks are handled in BASIC character constants:

Form Entered	Actual Constant Value	Length of Constant
'its'	its	3
"its"	its	3
"it's"	it's	4
'it''s'	it's	4
'"its"'	"its"	5
"""its"""	"its"	5
'"it''s"'	"it's"	6

Character Variables

A character variable is a named item of character data whose value is subject to change during execution of the program.

Character variables are named by a single letter of the extended alphabet followed by the currency symbol ($). Examples of such names are: A$ and $$. As stated under the heading "Alphabetic Characters," the variable A$ can also be referred to by the symbol a$.

When a BASIC program is executed, the initial value of each character variable is set to all blank characters.

The length of a character variable is specified in the DIM statement. In the absence of such a specification, the variable is assumed to have a length of 18 characters.

When a character value is assigned to a character variable, the value is adjusted to conform to the length of the variable. Longer values are truncated on the right, and shorter ones are left-justified and padded to the right with blanks.

Internal Variables

An internal variable is a numeric or character variable which can be accessed by the user but whose value is set and changed by VS BASIC. (The value of the variable &BUFF is reset to zero if &BUFF is specified in the RESET statement.)

When a VS BASIC program is executed, the initial value of all the numeric variables is set to zero, and the character variable is set to blanks.

Name	Type	Meaning
&BUFF	Numeric	Contains the number of unprocessed groups of input data values.
&CODE	Numeric	Contains the system return code resulting from a VSAM error.
&ERR	Numeric	Contains the VS BASIC error message number for identifying the particular error that was encountered.
&FILE	Character	Contains the name of the data file associated with the error condition.
&LINE	Numeric	Contains the line number of the statement where the error occurred.
&REC	Numeric	Contains the number of the last record of a relative record file, successfully referred to in an I/O statement.

Figure 2. Internal Variables

Arrays

Data items of the same type (either arithmetic or character) may be grouped together to form an array. An array is a collection of such data items that is referred to by a single name.

Arithmetic arrays are named by a single letter of the extended alphabet. Thus, the letter A may stand for a single arithmetic variable and/or an arithmetic array, while the symbol A2 may stand for only a single arithmetic variable. A single letter stands for an array only when it has been declared as an array implicitly or explicitly. Declaring arrays is discussed in the next section.

All members of an arithmetic array are initially set to zero when the program is executed.

Character arrays, like simple character variables, are named by a single letter of the extended alphabet followed by the currency symbol ($). Thus, the name D$ may refer to either a simple character variable or a character array.

All elements of a character array must be of the same length and are initially set to all blank characters when the program is executed.

BASIC arrays may be either one- or two-dimensional. A one-dimensional array can be thought of as a row of successive data items. A two-dimensional array can be thought of as a rectangular matrix of rows and columns. The following illustration shows a schematic representation of a one-dimensional array containing four elements and a two-dimensional array with four rows and three columns.

ONE-DIMENSIONAL ARRAY NAMED A			
A(1)	A(2)	A(3)	A(4)

TWO-DIMENSIONAL ARRAY NAMED B		
B(1,1)	B(1,2)	B(1,3)
B(2,1)	B(2,2)	B(2,3)
B(3,1)	B(3,2)	B(3,3)
B(4,1)	B(4,2)	B(4,3)

Each member in an array is referred to by the name of the array followed by a subscript in parentheses which indicates the position of the member within the array. The general form for referring to an array member is:

name (e_1[,e_2])

where *name* is the name of the array and *e* is any arithmetic expression whose integer value is greater than zero.

For a one-dimensional array, the expression in the subscript gives the position of the member in the array. Thus, the third member of a one-dimensional array named A$ can be referred to by the symbol A$(3), as in this example:

```
10        LET A$( 3 ) = 'MARCH'
```

For a two-dimensional array, the first expression in the subscript gives the number of the row containing the member referred to. Rows are numbered from top to bottom. The second expression in the subscript gives the number of the column containing the member referred to. Columns are numbered from left to right. Thus, the second member in the fourth row of a

two-dimensional array named B can be referred to by the symbol B(4,2), as in this example:

```
20        LET B(4,2) = 1.53E6
```

Declaring Arrays

The number of dimensions in an array, and the number of data items in each dimension, is established when the array is declared. In BASIC, arrays may be declared either explicitly, by use of the DIM statement, or implicitly. An implicit declaration is either:

1. A reference to a member of an array, without a preceding DIM statement.
2. The use of a name in a context where only an array name is permitted—in an array assignment statement, as a MAT name in an input or output list, or as an argument in the DET, SUM, PRD, or DOT intrinsic functions—without having been declared in a preceding DIM statement.

When an array is declared explicitly, the number of dimensions and the number of members in each dimension are specified in the DIM statement. For character arrays, the DIM statement may also specify the length of the members.

When an array is declared implicitly by a reference to one of its members, it will have the number of dimensions specified in the reference, and each dimension will be assumed to be ten. For example, when no prior DIM statement exists for an array named A, the statement:

```
100       LET A(3) = 50
```

will establish a one-dimensional array containing ten members, the third of which will have the integer value 50. Likewise, when no prior DIM statement exists for an array named B, the statement:

```
110       LET B(5,6) = 6.913
```

will establish a two-dimensional array containing ten rows and ten columns (100 members), with the sixth member of the fifth row containing the value 6.913.

When an array is declared implicitly by context, it is defined as a two-dimensional array, with ten members in each dimension. For example, when no prior DIM statement exists for the array named C, the statement:

```
150       MAT C = (1)
```

will establish a two-dimensional array containing ten rows and ten columns (100 members), with all members containing the value 1.

Whenever a character array is implicitly declared, either by reference to a member or by context, the length of each of its members is assumed to be eighteen characters.

One-dimensional arrays containing more than 10 members and two-dimensional arrays containing more than 100 members must be explicitly declared. Thus, appropriate prior DIM statements must exist for the following statements:

```
250       LET A(15) = 22.4
300       LET B(6,20) = 66.6
```

After an array has been declared, either explicitly or implicitly, it may not appear in a DIM statement elsewhere in the program. Arrays may be

redimensioned, however, by other BASIC statements according to the rules described in the next section, "Redimensioning Arrays."

Redimensioning Arrays

The following rules apply when an array is to be redimensioned:

1. An array can be redimensioned in an array assignment statement, a READ statement, an INPUT statement, a GET statement, a READ FILE statement, or a REREAD FILE statement.
2. An array cannot be redimensioned in a DIM statement.
3. An array can be redimensioned both in number of dimensions and in number of members per dimension as long as the original total number of members is not exceeded.

Naming Conventions for Variables and Arrays

Figure 3 provides a concise review of the names used to refer to variables and arrays in the BASIC language. The symbol *ext* denotes a letter of the extended alphabet.

Data Type	Name	Example
Arithmetic Variable	*ext*[digit]	A, a2, $3
Arithmetic Array	*ext*	A, b, #
Character Variable	*ext*$	A$, C$, @$
Character Array		

Figure 3. Naming Conventions for Variables and Arrays

Functions

The BASIC language supplies functions that perform a number of common operations. These are called the *intrinsic functions* and are summarized in Figure 4. In addition, BASIC allows the user to name and define his own frequently used functions through use of the DEF statement. Multiline functions (that is, functions defined by more than one BASIC statement) can be defined through use of the DEF, FNEND, and RETURN statements.

Function Name	Description	Type of Argument(s)	Type of Value Returned
ABS(*x*)	Absolute value of *x*	Arithmetic	Arithmetic
ACS(*x*)	Arcosine (in radians) of *x*	Arithmetic	Arithmetic
ASN(*x*)	Arcsine (in radians) of *x*	Arithmetic	Arithmetic
ATN(*x*)	Arctangent (in radians) of *x*	Arithmetic	Arithmetic
CEN(*x*)	Centigrade equivalent of *x* Fahrenheit degrees	Arithmetic	Arithmetic
CHR(*x*)	Converts the scalar arithmetic expression *x* to its equivalent character string	Arithmetic	Character
CLK	Time of day in 24-hour clock notation (in the form hh:mm:ss)	—	Character
CNT	Number of data items successfully processed by last I/O statement	—	Arithmetic
COS(*x*)	Cosine of *x* radians	Arithmetic	Arithmetic
COT(*x*)	Cotangent of *x* radians	Arithmetic	Arithmetic
CPU	Seconds taken by program execution	—	Arithmetic
CSC(*x*)	Cosecant of *x* radians	Arithmetic	Arithmetic
DAT[(*x*)]	Current Gregorian date, or Gregorian equivalent of Julian date *x***	Arithmetic	Character
DEG(*x*)	Number of degrees in *x* radians	Arithmetic	Arithmetic
DET(*x*)	Determinant of an arithmetic array	Arithmetic	Arithmetic
DOT(*x,y*)	Dot product* of arrays *x* and *y*	Arithmetic	Arithmetic
EXP(*x*)	Natural exponential of *x*	Arithmetic	Arithmetic
FAH(*x*)	Fahrenheit equivalent of *x* Centigrade degrees	Arithmetic	Arithmetic
HCS(*x*)	Hyperbolic cosine of *x* radians	Arithmetic	Arithmetic
HSN(*x*)	Hyperbolic sine of *x* radians	Arithmetic	Arithmetic
HTN(*x*)	Hyperbolic tangent of *x* radians	Arithmetic	Arithmetic
IDX(*x,y*)	Position of first character of string *y* within string *x*—value 0 returned if string not found	Character	Arithmetic
INT(*x*)	Integral part of *x*	Arithmetic	Arithmetic
JDY[(*x*)]	Current Julian date, or Julian equivalent of Gregorian date *x***	Character	Arithmetic
KLN(*x*)	Length in bytes of embedded key for file *x*	Character	Arithmetic
KPS(*x*)	Byte position at which embedded key for file *x* starts	Character	Arithmetic

* DOT is always calculated in a higher precision if available.
** Gregorian date form is *yyyy/mm/dd,* range is 0000/03/01 to 9999/12/31. Julian date range is 1721120 to 5373484.

Figure 4. Intrinsic Functions (Part 1 of 2)

Function Name	Description	Type of Argument(s)	Type of Value Returned
LEN(*x*)	Length of character string *x*, minus trailing blanks — value 0 returned if string all blanks	Character	Arithmetic
LGT(*x*)	Logarithm of *x* to the base 10	Arithmetic	Arithmetic
LOG(*x*)	Logarithm of *x* to the base e	Arithmetic	Arithmetic
LTW(*x*)	Logarithm of *x* to the base 2	Arithmetic	Arithmetic
MAX(*x,y*[,*z*...])	Maximum value of *x,y,z,...*	Arithmetic	Arithmetic
MIN(*x,y*[,*z*...])	Minimum value of *x,y,z...*	Arithmetic	Arithmetic
NUM(*x*)	Arithmetic value of character string *x*	Character	Arithmetic
PRD(*x*)	Product* of elements in array *x*	Arithmetic	Arithmetic
RAD(*x*)	Number of radians in *x* degrees	Arithmetic	Arithmetic
RLN(*x*)	Length of last record referred to in file *x*	Character	Arithmetic
RND[(*x*)]	Random number; a new stream of random numbers is begun starting with an unpredetermined number if *x* is 0, or starting with *x* if *x* is not 0; if *x* is not specified, the next number from a stream of random numbers is assigned; if no stream was previously defined (that is, no prior RND (*x*)), the stream of random numbers will begin with an unpredetermined number; all random numbers have a decimal value greater than 0, and less than 1, in the precision specified for the program (for example, .4157627 (short), .012345678901234 (long)).	Arithmetic	Arithmetic
SEC(*x*)	Secant of *x* radians	Arithmetic	Arithmetic
SGN(*x*)	Sign of *x* (—1, 0, or +1)	Arithmetic	Arithmetic
SIN(*x*)	Sine of *x* radians	Arithmetic	Arithmetic
SQR(*x*)	Square root of *x*	Arithmetic	Arithmetic
STR(*x,y*[,*z*])	Portion ** of string *x* from *y*th character to end of string or *z* characters from string *x*, starting with *y*th character	Character(*x*) Arithmetic(*y*) Arithmetic(*z*)	Character
SUM(*x*)	Sum* of elements in array *x*	Arithmetic	Arithmetic
TAN(*x*)	Tangent of *x* radians	Arithmetic	Arithmetic
TIM	Time of day in seconds since midnight	Arithmetic	Arithmetic

* PRD and SUM are always calculated in a higher precision if available.
** STR is also a pseudo variable. See "Pseudo Variables" in Part I of this publication for a discussion of its use.

Figure 4. Intrinsic Functions (Part 2 of 2)

Expressions

An expression in BASIC is any representation of an arithmetic or character value. Constants, variables (both scalar and array), array member references, and function references are all considered expressions. Expressions may also be formed by combining any of these value representations with symbols called *operators.*

An operator specifies either the relationship between data items, an arithmetic operation to be performed on them, or whether they are positive or negative. For example, the symbols >, *, and + are operators specifying greater than, multiplication, and positivity (or addition), respectively.

A special class of expressions, called *logical expressions,* is used with the IF statement to test the truth of specified relationships.

Expressions referring to entire arrays, rather than individual array members, are called *array expressions.* Any expression that evaluates to a single value, rather than to a set of values, is called a *scalar expression.*

Arithmetic Expressions and Operators

An arithmetic expression may be an arithmetic constant, an arithmetic variable, a subscripted arithmetic array member reference, or an arithmetic-valued function reference; or it may be a sequence of the above separated by binary arithmetic operators and parentheses and preceded by unary arithmetic operators. Some examples of arithmetic expressions are:

```
A1
X3/(-6)
X+Y+Z
SIN(R)
-6.4
-(X-X**2/2+X)
```

The value of an arithmetic expression is obtained by performing the implied operations on the specified data items according to the rules below.

The five binary arithmetic operators are:

Symbol	Meaning
** or ↑	Exponentiation (either form of the operator is acceptable)
*	Multiplication
/	Division
+	Addition
-	Subtraction

The two unary operators are:

+	Positive
-	Negative

Special cases for the arithmetic operators and the resulting actions are as follows:

Exponentiation: The expression A↑B or A**B is defined as the variable A raised to the B power.

1. If A=B=0, an error will occur.
2. If A=0 and B<0, an error will occur.
3. If A<0 and B is not an integer, an error will occur.
4. If A≠0 and B=0, A↑B or A**B is evaluated as 1.

5. If A=0 and B>0, A↑B or A**B is evaluated as 0.

Multiplication and Addition: A*B and A+B, multiplication and addition respectively, are both commutative, that is, A*B = B*A and A+B = B+A. They are not always associative due to low-order rounding errors, that is, A*(B*C) does not necessarily give the same results as (A*B)*C.

Division: A/B is defined as A divided by B. If B=0, an error "division by zero" will occur.

Subtraction: A-B is defined as A minus B. No special conditions exist.

Unary Operators: The + and - signs may also be used as unary operators to indicate:

- whether a constant is positive (+) or negative(-)
- whether the sign of a value assigned to a variable is to be retained (+) or reversed (-).

Two operators may not be used sequentially. Parentheses may be used to separate operators that would otherwise have to be specified sequentially.

For example:

```
-A+(-(B)) or B**(-2) is valid.
A + -B    or B**-2   is invalid.
```

Priority of Operators

Arithmetic expressions are evaluated according to the priorities of the operators involved. Operations with higher priorities are performed first; those at the same priority level are performed from left to right. The levels of priority of the operators are:

Operator	Priority Level
** or ↑	Highest
unary + and −	↓
* and /	
binary + and −	Lowest

An expression is evaluated by being reduced to its component subexpressions. A subexpression is defined as a group that can be read *operand-operator-operand,* where an *operand* is one of these:

1. A constant
2. A variable
3. A subscripted array member reference
4. A function reference
5. A subexpression

Starting with the first operator to be executed according to the priority scheme above, the operands of its subexpression are reduced to simple references to data in a left-to-right order. This process is repeated as many times as required in a left-to-right and/or descending order of priority of the remaining operators, until the entire expression is evaluated.

The normal priority sequence can be modified by enclosing subexpressions within parentheses. Subexpressions so modified will be evaluated beginning with the innermost set of parentheses.

The following examples illustrate the successive steps in the evaluation of four arithmetic expressions according to the rules described above. In each expression, the variables A, B, and C have been assigned the integer values 4, 6, and 2, respectively.

Expression	Evaluation and Result

`-A**2+B/C*2.5`	`-4**2+6/2*2.5`
	`-16 +6/2*2.5`
	`-16 + 3 *2.5`
	`-16 + 7.5`
	`-8.5`

`(-A**2)+B/C*2.5`	`(-4**2)+6/2*2.5`
	`-16 +6/2*2.5`
	`-16 + 3 *2.5`
	`-16 + 7.5`
	`-8.5`

`-A**(2+B/C)*2.5`	`-4**(2+6/2)*2.5`
	`-4**(2+ 3 )*2.5`
	`-4** 5 *2.5`
	`- 1024 *2.5`
	`-1024 *2.5`
	`-2560`

`-A**((2+B)/C)*2.5`	`-4**((2+6)/2)*2.5`
	`-4**( 8 /2)*2.5`
	`-4** 4 *2.5`
	`- 256 *2.5`
	`-256 *2.5`
	`-640`

Character Expressions and Operators

A character expression is a character constant, a character variable, a subscripted character array member reference, or a character-valued function reference; or it may be a sequence of the above separated by binary character operators and parentheses. The only binary character operator is:

Symbol	Meaning
`||` or .CAT.	Concatenation

The following are examples of valid character expressions:

```
A$
"ABC"
'abc' || 'defg'
D$(4)||D$(5)
D$(4) .CAT. A$ .CAT. 'ABC'
```

The length of the character string resulting from the concatenation of two or more character strings is the sum of the lengths of the individual strings.

Array Expressions

An array expression may appear only on the right side of the equal sign in an array assignment statement. It may take one of the following forms:

Form	Meaning
a	An array
a + *b*	Sum of two arrays
a - *b*	Difference between two arrays
a * *b*	Matrix product of two arrays
(*e*)* *a*	Product of a scalar value and an array
IDN	Identity matrix
INV(*a*)	Inverse of a matrix
TRN(*a*)	Transpose of a matrix
ASORT(*a*)	Ascending sort of an array
DSORT(*a*)	Descending sort of an array

where *a* and *b* are array names, and *e* is a scalar arithmetic expression.

For the array expressions *a,* ASORT(*a*), and DSORT(*a*), *a* can be either an arithmetic or character array. All other array expressions are arithmetic and require arithmetic operands.

The definition of these array expressions can be found in the discussion of the array assignment statement, later in this book.

Logical Expressions

A logical expression is either a logical subexpression or two logical subexpressions joined by a logical operator. It can appear in a BASIC program only as part of an IF statement. It has the general format:

s_1 [*logical-operator* s_2]

where *s* is a logical subexpression and *logical-operator* is either of these:

Operator	Meaning
& or .AND.	And
| or .OR.	Or

The logical expression is evaluated as being satisfied (true) or unsatisfied (false). If the And (&) operator is used, both logical subexpressions must be satisfied for the expression to be satisfied. If the Or (|) operator is used, the expression is satisfied if at least one of the subexpressions is satisfied.

Logical Subexpressions

A logical subexpression compares the values of two arithmetic expressions or two character expressions. It has the general format:

e_1 *relational-operator* e_2

where *e* is a scalar expression, and *relational-operator* is any of those operators described below. Both e_1 and e_2 must be of the same data type (character or arithmetic), and only two expressions may be compared in a single logical subexpression.

The relational operators and their definitions are:

Operator	Meaning
= or .EQ.	Equal
<> or ≠ or .NE.	Not equal
>= or ≥ or .GE.	Greater than or equal to
<= or ≤ or .LE.	Less than or equal
> or .GT.	Greater than
< or .LT.	Less than

When the alphabetic equivalents are used, they must be enclosed in periods.

The two expressions are first evaluated and then compared according to the definition of the relational operator specified. According to the result, the logical subexpression is either satisfied (true) or not satisfied (false).

When character data appears in a logical subexpression, it is evaluated according to the EBCDIC collating sequence, character by character, left to right. Thus, the following subexpressions would all be satisfied:

```
"ABC" = 'ABC'
'able' < 'BALL'
"123" > "ball"
'$123' .LT. "able"
```

When the alphabetic equivalents are used, they must be enclosed in periods.

When character operands of different lengths are compared, the shorter is considered to be extended on the right with blanks to the length of the longer operand. Thus, in the third example above:

```
"123" > "ball"
```

the values compared are 123b and ball, where b is a blank character.

BASIC Statements

Detailed specifications for all of the statements composing the BASIC language are contained in this chapter. For the most part, the statements are arranged in alphabetic order. In a few cases, related statements are discussed together; the combined discussion is cross-referenced, however, in the appropriate alphabetic position.

Each discussion contains the following topics:

Function: A brief description of the purpose of the statement.

General Format: A description of the syntax of each statement.

Action: A discussion of the action that results when the statement is encountered in the BASIC program.

Rules: A discussion of pertinent factors that affect coding of the statement.

Example: An illustration of how the statement would appear in the program.

The Array Assignment Statement

Function:

The array assignment statement causes values to be assigned to the members of an array. The value to be assigned can be derived from a parenthesized scalar expression or from one of the following array expressions:

Array Expression	Meaning
a	Simple array
a+b	Addition
a-b	Subtraction
a*b	Matrix Multiplication
(e)*a	Scalar multiplication
IDN	Identity function
INV(a)	Inverse function
TRN(a)	Transpose function
ASORT(a)	Ascending sort function
DSORT(a)	Descending sort function

For a discussion of how this statement is used, see "Using Arrays" in Part I of this publication.

The Array Assignment Statement (Scalar Value)

Function:

This statement assigns a specified scalar value to each member of an array.

General Format:

MAT *name* [(*r*)] = (*exp*)[REM *comment*]

where *name* is the name of an array, *r* is a redimension specification, *exp* is a scalar expression, and *comment* is one or more EBCDIC characters.

Action:

The parenthesized scalar expression to the right of the equal sign is evaluated, and each member of the array to the left of the equal sign is set to that value.

If a redimension specification follows the array name, the truncated integer portion of each expression value in *r* is used to redimension the array before the scalar value is evaluated and assigned to each of its members.

For character data, if the character value being assigned is shorter than the members of the array, the value is padded on the right with blanks to the length of the array members before being assigned. If the character value being assigned is longer than the members of the array, the value is truncated on the right to the length of the array members before being assigned. Character constants containing no characters (null) are assigned as all blank characters.

The scalar expression to the right of the equal sign must be of the same type (arithmetic or character) as the array to which it is assigned.

Examples:

The first example shows simple assignment of a scalar value to an array; contrast this with the second example in the next section (Simple Array).

```
10   DIM A(2,2)
20   B = 5
30   MAT A = (B)
```

The resulting values are represented below:

A is $\begin{bmatrix} 5 & 5 \\ 5 & 5 \end{bmatrix}$

The second example shows assignment of a scalar value to an array with redimensioning specified.

```
40 DIM Y(3,3)
50 MAT Y(2,2) = (0)
```

The resulting values are represented below:

Y is $\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$

The Array Assignment Statement (Simple Array)

Function:

This statement assigns the elements of one array to another array.

General Format:

MAT *name*$_1$ [(*r*)] = *name*$_2$[REM *comment*]

where *name* is the name of an array, *r* is a redimension specification, and *comment* is one or more EBCDIC characters.

Action:

Each element of the array specified to the right of the equal sign is assigned to the corresponding member of the array to the left of the equal sign.

If a redimension specification follows the name to the left of the equal sign, the truncated integer portion of each expression value in *r* is used to redimension the array before values are assigned to it.

For character arrays, if the members of the array to the right of the equal sign are shorter than the members of the array to the left of the equal sign, each value is padded on the right with blanks to the length of the receiving array before being assigned. If the members of the array to the right of the equal sign are longer than the members of the array to the left of the equal sign, each value is truncated on the right to the length of the receiving array before being assigned.

Rules:

1. Both arrays specified must be of the same type (arithmetic or character).
2. Both arrays specified in the array assignment statement must have identical dimensions (after redimensioning, if any).
3. The same array name should not be specified on both sides of the equal sign.

Examples:

The first example shows assignment of the elements of an array to another array.

```
20      DIM A (2,2), B(2,2)
            .
            .
            .
100     MAT A = B
```

The resulting values are represented below:

If B = $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ A is $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$

The second example can be contrasted with the first example in the section above (Scalar Value).

```
10 DIM A(2,2), B(2,2)
20 B = 5
30 MAT A = B
```

Because B in the MAT assignment statement is recognized as an array, not a scalar, the computer assigns the values of array B to array A:

A is $\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$

The Array Assignment Statement (Addition and Subtraction)

Function:

These statements assign the sum or the difference of the elements of two arrays to the members of a third array.

General Format:

MAT *name* $_1$ [(*r*)] = *name* $_2$ {+ | -) *name* $_3$[REM *comment*]

where *name* is the name of an arithmetic array, *r* is a redimension specification, and *comment* is one or more EBCDIC characters.

Action:

The elements of the arrays specified to the right of the equal sign are added or subtracted, as indicated, and the result of the operation is assigned to the corresponding member in the array to the left of the equal sign.

If a redimension specification follows the name to the left of the equal sign, the truncated integer portion of each expression value in *r* is used to redimension the array before values are assigned to it.

Rules:

1. All three arrays must be arithmetic.
2. All three arrays specified in the statement must have identical dimensions (after redimensioning, if any).
3. If the array to the left of the equal sign is being redimensioned, it cannot appear to the right of the equal sign.

Example:

```
10    DIM X(3,3), Y(2,2), Z(2,2)
           .
           .
           .
100   MAT X(2,2) = Y + Z
```

The resulting values are represented below:

$$\text{If } Y = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \text{ and } Z = \begin{bmatrix} e & f \\ g & h \end{bmatrix} \quad X \text{ is } \begin{bmatrix} a+e & b+f \\ c+g & d+h \end{bmatrix}$$

The Array Assignment Statement (Matrix Multiplication)

Function:

This statement performs the mathematical matrix multiplication of two arithmetic arrays and assigns the product to a third.

General Format:

MAT *name*$_1$ [(*r*)]= *name* $_2$ * *name* $_3$ [REM *comment*]

where *name* is the name of an arithmetic array, *r* is a redimension specification, and *comment* is one or more EBCDIC characters.

Action:

In matrix multiplication, an array A of dimensions (p,m) and an array B of dimensions (m,n) yield a product array C of dimensions (p,n) such that for i = 1,2,...,p and for j = 1,2,...,n:

$$C(i,j) = \sum_{k=1}^{m} A(i,k) * B(k,j)$$

If a redimension specification follows the name to the left of the equal sign, the truncated integer portion of each expression value in *r* is used to redimension the array before values are assigned to it.

Rules:

1. All three arrays specified must be arithmetic.
2. The array specified to the left of the equal sign may not be the same array as either array to the right of the equal sign.
3. All of the following relationships must be true (after redimensioning, if any):

a. All three arrays must be two-dimensional.

b. The number of columns in the array specified by *name*$_2$ must be equal to the number of rows in the array specified by *name*$_3$.

c. The number of rows in the array specified by *name*$_1$ must equal the number of rows in the array specified by *name*$_2$.

d. The number of columns in the array specified by *name*$_1$ must equal the number of columns in the array specified by *name*$_3$.

Example:

```
10    DIM X(2,2), Y(2,2), Z(2,2)
            .
            .
            .
100   MAT Z = X * Y
```

The resulting values are represented below:

$$\text{If X is} \begin{bmatrix} a & b \\ c & d \end{bmatrix} \text{ and Y} = \begin{bmatrix} e & f \\ g & h \end{bmatrix} \text{ Z is } \begin{bmatrix} a*e + b*g & a*f + b*h \\ c*e + d*g & c*f + d*h \end{bmatrix}$$

The Array Assignment Statement (Scalar Multiplication)

Function:

This statement causes the elements of an arithmetic array to be multiplied by the value of an arithmetic expression, and the resulting products to be assigned to the members of another arithmetic array.

General Format:

MAT *name*$_1$ [(*r*)] = (*x*)**name*$_2$ [REM *comment*]

where *name* is the name of an arithmetic array, *r* is a redimension specification, the parenthesized *x* is a scalar arithmetic expression, and *comment* is one or more EBCDIC characters.

Action:

The scalar expression is evaluated, and that value is multiplied by the value of each member in the array to the right of the equal sign. The resulting products are then assigned to the corresponding members of the array to the left of the equal sign.

If a redimension specification follows the name to the left of the equal sign, the truncated integer portion of each expression value in *r* is used to redimension the array before any other action takes place.

Rules:

1. Both arrays specified must be arithmetic.
2. Both arrays specified must have identical dimensions (after redimensioning, if any).
3. If the array to the left of the equal sign is being redimensioned, it cannot appear to the right of the equal sign.

Example:

```
20      DIM X(2,2), Y(2,2)
          .
          .
          .
100     MAT Y = (4) * X
```

The resulting values are represented below:

If X = $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ Y is $\begin{bmatrix} 4*a & 4*b \\ 4*c & 4*d \end{bmatrix}$

The Array Assignment Statement (Identity Function)

Function:

This statement causes an arithmetic array to assume the form of an identity matrix.

General Format:

MAT *name*[(*r*)] = IDN [REM *comment*]

where *name* is the name of an arithmetic array, *r* is a redimension specification, and *comment* is one or more EBCDIC characters.

Action:

Each member of the specified array for which the values of both subscripts are equal, for example, A(2,2) or A(3,3), is assigned the integer value 1. All other members—for example, A(2,3) or A(3,1)—are assigned the value 0.

If a redimension specification follows the array name, the truncated integer portion of each expression value in *r* is used to redimension the array before the assignment of 1 or 0 to each of its members.

Rules:

1. The array specified must be arithmetic.
2. The specified arithmetic array must be a square matrix; that is, the number of rows must equal the number of columns (after redimensioning, if any).

Example:

```
50    DIM X(16)
        .
        .
        .
60    MAT X(4,4) = IDN
```

The resulting values are represented below:

X is $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

The Array Assignment Statement (Inverse Function)

Function:

This statement causes one array to be assigned the mathematical matrix inverse of another array.

General Format:

MAT *name* $_1$ [(*r*)] = INV (*name* $_2$) [REM *comment*]

where *name* is the name of an arithmetic array, *r* is a redimension specification, and *comment* is one or more EBCDIC characters.

Action:

The matrix inverse of the array specified to the right of the equal sign is assigned to the array specified to the left of the equal sign. For the square array A of dimensions (m,m), the inverse array B, if it exists, is an array of identical dimensions such that:

A*B = B*A = I

where I is an identity matrix.

Not every matrix has an inverse. The intrinsic function DET (see "Functions") may be used to determine if a given array has an inverse. The inverse of array A exists if DET(A) $\neq$ 0.

If a redimension specification follows the name to the left of the equal sign, the truncated integer portion of each expression value in *r* is used to redimension the array before values are assigned to it.

Rules:

1. Both arrays specified must be arithmetic.
2. Both arrays specified must be square arrays, and both must have identical dimensions (after redimensioning, if any).
3. The same array name may not be used on both sides of the equal sign.

Example:

```
20      DIM X(2,2), Y(2,2)
             .
             .
             .
80      IF DET(Y) = 0 THEN 300
90      MAT X = INV(Y)
             .
             .
             .
295     GO TO 110
300     PRINT 'SINGULAR MATRIX'
310     END
```

The resulting values are represented below:

If Y is $\begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}$ Then X is $\begin{bmatrix} 2 & -1 \\ -1 & 1 \end{bmatrix}$

The Array Assignment Statement (Transpose Function)

Function:

This statement causes the elements of one array to be replaced by the matrix transpose of another array.

General Format:

MAT *name*$_1$ [(*r*)]= TRN (*name* $_2$)[REM *comment*]

where *name* is the name of an arithmetic array, *r* is a redimension specification, and *comment* is one or more EBCDIC characters.

Action:

The transpose matrix of the array specified to the right of the equal sign is assigned to the array specified to the left of the equal sign. The values in column y of one array become the values in row y of the other array.

If a redimension specification follows the name to the left of the equal sign, the truncated integer portion of each expression value in *r* is used to redimension the array before values are assigned to it.

Rules:

1. Both arrays specified must be arithmetic.
2. Both arrays specified must be two-dimensional, and the number of rows in each array must be equal to the number of columns in the other (after redimensioning, if any).
3. The same array name may not be used on both sides of the equal sign.

Example:

```
40 DIM A(3,2), B(2,3)
      .
      .
      .
60 MAT B = TRN(A)
```

The resulting values are represented below:

If A is $\begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix}$ Then B is $\begin{bmatrix} a & c & e \\ b & d & f \end{bmatrix}$

The Array Assignment Statement (Ascending Sort Function)

Function:

This statement causes the elements of one array to be sorted in ascending order and assigned to the members of another array.

General Format:

MAT *name*$_1$ [(*r*)] = ASORT (*name* $_2$)[REM *comment*]

where *name* is the name of an array, *r* is a redimension specification, and *comment* is one or more EBCDIC characters.

Action:

The elements of the members of the array specified to the right of the equal sign are sorted in ascending order and assigned to the members of the array to the left of the equal sign. For a one-dimensional array A, the smallest value is placed in A(1), the next smallest value in A(2), and so on. For two-dimensional arrays, values are assigned row by row. Thus, for a two-dimensional array B, the smallest value is placed in B(1,1), the next smallest value in B(1,2), and so on.

For a character array, the data is sorted according to the EBCDIC collating sequence, character by character, left to right. If the members of the array to the right of the equal sign are shorter than the members of the array to the left of the equal sign, each value is padded on the right with blanks to the length of the receiving array before being assigned. If the members of the array to the right of the equal sign are longer than the members of the array to the left of the equal sign, each value is truncated on the right to the length of the receiving array before being assigned.

If a redimension specification follows the name to the left of the equal sign, the truncated integer portion of each expression value in *r* is used to redimension the array before values are assigned to it.

The sort operation has no effect on the array specified to the right of the equal sign, unless it is the same array that is specified to the left of the equal sign.

Rules:

1. Both arrays specified must be of the same type (arithmetic or character).
2. Both arrays specified must have identical dimensions (after redimensioning, if any).
3. If the array to the left of the equal sign is being redimensioned, it cannot appear to the right of the equal sign.

Example:

```
20      DIM A(3,3), B(3,3)
          .
          .
          .
50      MAT B = ASORT(A)
```

If A = $\begin{bmatrix} 4 & 8 & 6 \\ 3 & 7 & 2 \\ 5 & 1 & 9 \end{bmatrix}$ Then B is $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$

The Array Assignment Statement (Descending Sort Function)

Function:

This statement causes the elements of one array to be sorted in descending order and assigned to the members of another array.

General Format:

MAT *name*$_1$ [(*r*)] = DSORT (*name*$_2$)[REM *comment*]

where *name* is the name of an array, *r* is a redimension specification, and *comment* is one or more EBCDIC characters.

Action:

The elements of the array specified to the right of the equal sign are sorted in descending order and assigned to the members of the array to the left of the equal sign. For a one-dimensional array A, the largest value is placed in A(1), the next largest value in A(2), and so on. For two-dimensional arrays, values are assigned row by row. Thus, for a two-dimensional array B, the largest value is placed in B(1,1), the next largest value in B(1,2), and so on.

For a character array, the data is sorted according to the EBCDIC collating sequence, character by character, left to right. If the members of the array to the right of the equal sign are shorter than the members of the array to the left of the equal sign, each value is padded on the right with blanks to the length of the receiving array before being assigned. If the members of the array to the right of the equal sign are longer than the members of the array to the left of the equal sign, each value is truncated on the right to the length of the receiving array before being assigned.

If a redimension specification follows the name to the left of the equal sign, the truncated integer portion of each expression value in *r* is used to redimension the array before values are assigned to it.

The sort operation has no effect on the array to the right of the equal sign, unless it is the same array that is specified to the left of the equal sign.

Rules:

1. Both arrays specified must be of the same type (arithmetic or character).
2. Both arrays specified must have identical dimensions (after redimensioning, if any).
3. If the array to the left of the equal sign is being redimensioned, it cannot appear to the right of the equal sign.

Examples:

```
20      DIM A$2(3,3), B$2(3,3)
          .
          .
          .
50      MAT B$ = DSORT(A$)
```

The resulting values are represented below:

$$\text{If A\$} = \begin{bmatrix} CB & BB & AC \\ AB & CA & CC \\ AA & BC & BA \end{bmatrix} \quad \text{Then B\$ is} \quad \begin{bmatrix} CC & CB & CA \\ BC & BB & BA \\ AC & AB & AA \end{bmatrix}$$

```
110     DIM A$3(2,2), B$2(2,2)
120     MAT B$ = DSORT(A$)
```

The resulting values are represented below:

$$\text{If A\$} = \begin{bmatrix} ABC & XYZ \\ RST & UVW \end{bmatrix} \quad \text{Then B\$ is} \quad \begin{bmatrix} XY & UV \\ RS & AB \end{bmatrix}$$

The CHAIN and USE Statements

Function:

The CHAIN statement terminates execution of the program in which it appears and starts another BASIC program, causing it to be executed. The USE statement is a nonexecutable statement in the chained program that specifies a variable to receive a character value passed into the program from the chaining program. For a discussion of how these statements are used, see "Program Chaining" in Part I of this publication.

General Format:

CHAIN *p* [,*a*][REM *comment*]
USE *b* [REM *comment*]

where *p* is the name of the chained program, *a* is a character expression representing a value to be passed to the chained program, *b* is a character variable that is to receive the value *a*, and *comment* is one or more EBCDIC characters.

Action:

When a CHAIN statement is executed, the current program is terminated and the program (*p*) specified in the CHAIN statement is executed. If *a* is specified in the CHAIN statement, the character expression is evaluated and its value passed to the chained program. The maximum length of the character-string is 255 bytes. If a chained program contains a USE statement, the character variable in the USE statement is initialized at the start of program execution by the value passed from the chaining program.

If the character string passed from a chaining program is shorter than the length of the variable in the USE statement, the string is padded on the right with blanks to the length of the USE statement variable. If the character string is longer than the USE statement variable, it is truncated on the right.

Rules:

1. The USE statement can appear anywhere in a program except within a multiline user function definition (that is, between a DEF statement and its associated FNEND statement).
2. There must be no more than one USE statement in the chained program.

Example:

Chaining Program:

```
250 CHAIN 'SCAN',N$
```

Chained Program:

```
30  USE  P$
```

Statement 250 terminates execution of the current program. All active files are closed. The value of N$ is used to initialize the variable P$ in the program SCAN, which is then executed.

The CLOSE Statement

Function:

The CLOSE statement causes input and output files to be deactivated. For a discussion of how this statement is used, see "Activating and Deactivating Files" for stream-oriented files, and "Opening and Closing Files" for record-oriented files in Part I of this publication.

General Format:

CLOSE[FILE] *filename*$_1$ [,*filename*$_2$] ... [,EXIT *es* | ,IOERR *s*] [REM *comment*]

where *filename* is a character expression, *es* is the statement number of the EXIT statement, *s* is an executable statement number, and *comment* is one or more EBCDIC characters.

Action:

The file or files specified in the CLOSE statement are deactivated. If the keyword FILE is specified, the file or files to be deactivated are record-oriented files.

If the EXIT clause is included, and an error condition arises, the specified EXIT statement is examined for the appropriate error condition and control passes to that statement.

If the IOERR clause is included, a device malfunction prevents the closing of the file and control is tranferrred to the specified statement.

An implicit CLOSE statement is automatically executed for each active file at the completion of program execution.

Rules:

1. All of the files specified in one CLOSE statement must be either stream-oriented or record-oriented; the two types cannot be combined in a single statement.
2. A CLOSE statement for record-oriented files must contain the keyword FILE.
3. If a stream-oriented file is to be used for both output and input during execution of a single program, it must be closed between output and input references.
4. If a file specified in a CLOSE statement is not active at the time the CLOSE statement is executed, its appearance in the CLOSE statement is ignored.
5. If EXIT is specified for a particular file, the IOERR error clause cannot be specified.
6. In a multiple CLOSE statement, an error causes transfer of program control to the statement designated, and any subsequent files specified are not processed.
7. Naming conventions for files depend upon the environment in which they are used. See Appendix A. "Implementation Considerations."

Examples:

```
25  CLOSE 'CDF', "RST"
30  CLOSE FILE 'RECFIL', A$
40  CLOSE 'STOCK', A$, IOERR 100
```

The DATA Statement

Function:

The DATA statement is a nonexecutable statement that creates an internal data table from which values are supplied to variables, arrays, and pseudo-variables specified in corresponding READ statements. For a discussion of how this statement is used, see "Getting Data Into the Computer" in Part I of this publication.

General Format:

DATA [n_1*] *constant*$_1$ [,[n_2*] *constant*$_2$] ...

where *constant* is either an arithmetic or character constant, and *n* is a nonzero, unsigned, integer constant specifying the number of times the constant is to appear in succession in the data table.

Action:

Before execution time, a single table is constructed containing all the values from all the DATA statements in the program in their order of appearance by statement number. At the same time, a pointer is set to the first item in the table. The pointer is advanced through the table, item by item, as the data is supplied to items in the input list of a READ statement.

Rules:

1. Each item of data in a DATA statement must be of the same type as that specified by the variable to which it is to be assigned in the corresponding READ statement. Thus, if the third constant in the DATA statement is a character constant, then the READ statement variable to which it is assigned must be a character variable.
2. DATA statements may be placed either before or after the READ statements to which they supply data.
3. A DATA statement cannot appear within a multiline function definition (between a DEF statement and its associated FNEND statement).
4. A character constant should be bounded by a pair of single or double quotation marks, if the constant:
   - contains commas or slashes
   - has leading or trailing blanks or tab characters
   - has leading single or double quotation marks.
   - starts with an integer immediately followed by an asterisk.

   Otherwise, a character constant does not have to be enclosed in quotation marks.
5. A number that is not enclosed in quotation marks can be assigned to either a character or numeric variable; a quoted constant may only be assigned to a character variable.
6. Comments are not allowed on the DATA statement.

Examples:

```
10  DATA 'JONES', 15.00, SMITH, 20.50
20  DATA 3*1.0, 2.0, 2*3.0
30  DATA 3*ABC,'3*ABC'
```

Note in the last example, that the first item will initialize 3 character variables, each with a value of ABC, whereas the second item will initialize 1 character variable with the value of 3* ABC.

The DEF, RETURN, and FNEND Statements

Function:

The DEF statement is a nonexecutable statement that defines a user-written function. It can be used alone to define a single-line function. In conjunction with the RETURN and FNEND statements, it defines a multiline function. For a discussion of how these statements are used, see "Functions" in Part I of this publication.

General Format:

Single-line function definition (arithmetic):

[DEF] FN *a* [(*dv*$_1$[,*dv*$_2$]...)] = *scalar-arithmetic-expression* [REM *comment*]

Single-line function definition (character):

[DEF] FN *b* [(*dv*$_1$[,*dv*$_2$]...)] = *scalar-character-expression* [REM *comment*]

Multiline function definition (arithmetic):

```
[DEF] FN a [(dv1[, dv2]...)] [REM comment]
   .
   .
   .
RETURN scalar-arithmetic-expression [REM comment]
   .
[RETURN scalar-arithmetic-expression] [REM comment]
   .
   .
FNEND [comment] [REM comment]
```

Multiline function definition (character):

```
[DEF] FN b$ [(dv1[,dv2]...)] [REM comment]
   .
   .
   .
RETURN scalar-character-expression [REM comment]
   .
[RETURN scalar-character-expression] [REM comment]
   .
   .
FNEND [[REM] comment]
```

where *a* is any letter of the extended alphabet or a digit 0 through 9, *b* is any letter of the extended alphabet, *dv* is a *dummy variable,* which can be an arithmetic or character variable, or a character variable followed by an unsigned, nonzero, integer constant defining the length of the dummy variable, and *comment* is one or more EBCDIC characters.

A reference to a user-written function has the general format:

Arithmetic Function:

FN*a* [(*x*$_1$[,*x*$_2$]...)] [REM *comment*]

Character Function:

FN*b*$ [(*x*$_1$[,*x*$_2$]...)] [REM *comment*]

where FN*a* or FN*b* $ is the name of the function and *x* is an *argument,* which can be either an arithmetic or character expression.

Action:

When a reference to a user-written function is encountered in an expression at execution time, the current value of each argument *x* is used to initialize the corresponding dummy variable *dv.* When the expression in the DEF or RETURN statement is evaluated, the result is assigned as the value of the function reference.

If a character argument is shorter than the corresponding dummy variable, the value of the argument is padded on the right with blanks to the length of the dummy variable before being assigned to it. If a character argument is longer than its corresponding dummy variable, its value is truncated on the right to the length of the dummy variable before being assigned to it.

The values of any program variables or array member references that appear either in the DEF statement expression or within the multiline function definition are set at the time of invocation.

Rules:

1. A function may be defined anywhere in a BASIC program, either before or after references to it.
2. A function of a given name may be defined only once in a program.
3. A function definition may not contain references to itself nor to other functions which refer to it in their definitions.
4. A function reference to a user-written function may appear anywhere in a BASIC expression that a constant, variable, subscripted array member reference, or intrinsic function reference may appear.
5. A function must not change the value of any variable which appears in the statement containing the function reference.
6. The list of arguments in a reference to a user-written function must agree in number, order, and type (arithmetic or character) with the dummy variables in the DEF statement.
7. The maximum number of arguments in a user-written function is 25.9
8. If there is no length specified for a character dummy variable in the DEF statement, the dummy variable is assumed to have a length of eighteen characters.
9. The dummy variable *dv* has meaning only within the function definition. Consequently, it is possible to have a dummy variable with the same name as a variable used elsewhere in the program. The BASIC language will recognize each as a unique identifier, and no conflict of names or values will result from this duplicate usage—for example, modification of one has no effect on the other.
10. The maximum number of user-written functions permitted in a BASIC program is 39 numeric functions and 29 character functions. The maximum number of active function references is 47, unless there are active GOSUB statements. For every two active GOSUB statements, 1 must be subtracted from this maximum. Thus, with four active GOSUB statements, the maximum number of active function references is 45. (See "The GOSUB and RETURN Statements.")

11. The following statements may not appear within a multiline user function definition (between a DEF statement and its associated FNEND statement): DATA, DEF, DIM, END, FORM, Image, and USE.

12. An FNEND statement may not appear outside the multiline user function definition; a RETURN statement outside of a function definition can be of the form; RETURN[[REM] *comment*].

13. Transfer of control into or out of multiline function definition is not permitted.

    a. An I/O statement within a multiline function definition may not refer to an EXIT statement that is outside of the function definition; similarly, an I/O statement outside of a function definition may not refer to an EXIT statement within the function definition.

    b. The ON statement cannot transfer control into or out of a user-defined function.

14. User-defined functions that are referred to during an I/O operation may not themselves perform any I/O.

15. If control is passed to a DEF statement, either through normal sequential execution or from elsewhere in the program, control goes to the statement following the function definition.

16. The last executable statement preceding the FNEND statement must prevent control from passing to the FNEND statement—such as a RETURN, STOP, CHAIN, or unconditional GOTO.

Examples:

After execution of the following series of BASIC statements, the variable Z$ will have the value "AREA CODE" and W$ will have the value "ZIP CODE".

```
00    DIM Z$9, W$8
10    LET Y$ = 'AREA '
20    DEF FNA$ (X$5) = X$||'CODE'
30    LET Z$ = FNA$(Y$)
40    LET W$ = FNA$('ZIP ')
```

In the next example, the variable R will have the integer value 72 after execution of statement 80. When statement 80 is executed, the current value of Y, which is 2, is substituted for each occurrence of the dummy variable X in the arithmetic expression of statement 100. Since the function FNC, defined in statement 100, uses the function FNB in its definition, the value 2 is substituted for each occurrence of X in the arithmetic expression of statement 90. The resulting value, 47, is then substituted for the function reference FNB(X) in statement 100. The current value of Y, 2, is then added to 47, and the resulting value, 49, is substituted for the function reference FNC(Y) in statement 80. This value is added to 23, and the resulting value, 72, is assigned to the variable R.

```
70    LET Y = 2
80    LET R = FNC(Y) + 23
90    DEF FNB(X) = 5*X**2+27
100   DEF FNC(X) = FNB(X) + X
```

The next example shows a multiline function definition. When these statements are executed, C will have a value of 7 and D will have a value of -2.

```
10    LET A = 5
20    LET B = 2
30    DEF FNA (X,Y)
40    IF X>0 THEN RETURN X+Y ELSE RETURN X-Y
50    FNEND
60    LET C = FNA(A,B)
70    LET A = 0
80    LET D = FNA(A,B)
```

The DELETE FILE Statement

Function:

The DELETE FILE statement causes a specified record to be deleted from a key-sequenced or relative-record record-oriented file. For a discussion of how this statement is used, see "Deleting Records" in Part I of this publication.

General Format—Key Sequenced Files:

DELETE FILE *filename*, **KEY** = *exp* [,**EXIT** *es* / ,**IOERR** *s*] [,**NOKEY** *s*] [**REM** *comment*]

General Format-Relative Record Files:

DELETE FILE *filename*, **REC** = *e* [,**EXIT** *es* / ,**IOERR** *s*] [,**NOREC** *s*] [**REM** *comment*]

where *filename* is a character expression, *exp* is a character expression, *es* is the number of an EXIT statement, *e* is a positive scalar arithmetic expression, *s* is the number of an executable statement, and *comment* is one or more EBCDIC characters.

Action:

If it has not been previously opened, the file specified in the DELETE FILE statement is activated for input and output operations. Execution of the DELETE FILE statement causes the record whose key is specified in the KEY clause or whose relative-record number is specified in the REC clause, to be deleted from the specified file. The file is then positioned after the deleted record.

The character expression in the KEY clause is evaluated and truncated on the right to the length of the record key before being compared with the actual record key. A character string shorter than the record key is compared with the first *n* characters of the key, where *n* is the number of characters in the character string, and the first record satisfying that condition is deleted from the file.

The numeric expression in the REC clause is the integer part of a positive number which gives the relative record number 1, 2, 3,...*m*, with *m* being the maximum number of records in the file. The relative record number is not part of the data record; it is used to calculate the relative position of the record within the file.

If the NOKEY clause is included, control is transferred to the specified statement if a record with the specified key does not exist on the file.

If the NOREC clause is included, and a record with the specified relative-record number does not exist on the file, control is transferred to the specified statement.

If the IOERR clause is included, control is transferred to the specified statement if a device malfunction prevents deletion of the record.

If the EXIT clause is included, the specified EXIT statement is examined for the appropriate error condition if any of the above problems occur.

Rules:

1. The NOKEY or NOREC and IOERR clauses can be specified in any order. The presence of any of these clauses precludes the specification of an EXIT clause.
2. The specified file must be a record-oriented file that is activated for both input and output operations.
3. The internal variable &REC contains the relative record number of the last record successfully referred to.
4. Naming conventions for files depend upon the environment in which they are used. See Appendix A. "Implementation Considerations."

Example:

```
30       DELETE FILE "FAB", KEY = "320"
```

The DIM Statement

Function:

The DIM statement is a nonexecutable statement used to specify the size of arrays and the length of a character variable or array member. For a discussion of how this statement is used, see "Rules for Forming Variables" for character data and "Using Arrays" for arrays in Part I of this publication.

General Format:

DIM *name*$_1$ [*len*$_1$] [(r_1 [,c_1])] [,*name*$_2$ [*len*$_2$] [(r_2 [,c_2])]] ... [REM *comment*]

where *name* is an arithmetic or character array name or a character scalar variable, *len* is a nonzero, unsigned, integer constant specifying the length of a character variable or a character array member, *r* and *c* are nonzero, unsigned, integer constants specifying the first and second dimensions of an array, respectively, and *comment* is one or more EBCDIC characters.

Action:

A character scalar variable in a DIM statement is defined as having the number of characters in its associated length specification *len*.

A one-dimensional array whose name is specified in a DIM statement is defined as having the number of members represented by the integer *r*.

A two-dimensional array whose name is specified in the DIM statement is defined as having *r* number of rows and *c* number of columns.

All members of a character array whose name is specified in a DIM statement are defined as having the length specified by *len*; if *len* is absent for that array name, all members are assumed to have a length of 18 characters.

The initial value of each arithmetic array member is zero. The initial value of each character array member is all blank characters.

Rules:

1. An array name may not appear in a DIM statement if it has been previously defined, either implicitly, or explicitly in a prior DIM statement. (Arrays may be redimensioned after definition according to the rules explained in the section "Redimensioning Arrays.")
2. A character variable may not appear in a DIM statement if it has appeared in any previous statement in the program.
3. Arrays of one or two dimensions may be defined in a DIM statement.
4. A length specification must not accompany an arithmetic array name; it is optional for a character array name; it must be specified for a character scalar variable.
5. The maximum permissible length of a character variable or character array member is 255.
6. The maximum permissible size of one- and two-dimensional arrays is 32767 members.

Example:

```
20    DIM A(4,2), B$6(2,2), D$8, Z$(5)
```

When the program is executed, the initial value of the arrays will be:

$$A=\begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

$$B=\begin{bmatrix} \text{ƀ ƀ ƀ ƀ ƀ ƀ} & \text{ƀ ƀ ƀ ƀ ƀ ƀ} \\ \text{ƀ ƀ ƀ ƀ ƀ ƀ} & \text{ƀ ƀ ƀ ƀ ƀ ƀ} \end{bmatrix}$$

D$= ƀ ƀ ƀ ƀ ƀ ƀ ƀ ƀ

Z$ = Five strings of 18 blank characters each.

The END Statement

Function:

The END statement signifies the end of source program statements, and, at execution time, causes the termination of a BASIC program. For a discussion of how this statement is used, see "A Simple Program" in Part I of this publication.

General Format:

```
END [ [REM] comment                    ]
    [ RC = exp [REM comment]           ]
```

where *comment* is one or more EBCDIC characters and *exp* is a numeric expression which returns the integer part of a non-negative value.

Action:

The END statement indicates the logical end of a program. When it is encountered during program compilation, it causes any statements that follow it numerically to be excluded from the program. When an END statement is encountered at execution time, it causes closing of all open files and termination of processing. By specifying the expression RC=, the user can return a completion code to the host system at the end of execution.

Rule:

1. The END statement is optional. If omitted by the user, END will be assumed by the system to follow the highest-numbered statement in the program.

Example:

```
999 END RC=16 REM RETURN TO HOST SYSTEM
```

The EXIT Statement

Function:

The EXIT statement is a nonexecutable statement used in conjunction with input/output statements for stream and record-oriented files to allow transfer of control when certain errors occur during the I/O operation. For a discussion of how this statement is used, see "Input/Output Error Handling" for stream-oriented files and "Using the EXIT Statement" and "Key Clauses on the EXIT statement" for record-oriented files, in Part I of this publication.

General Format:

EXIT [EOF *s*][,IOERR *s*][,CONV *s*][,DUPKEY *s*][,NOKEY *s*]
[,DUPREC *s*][,NOREC *s*][REM *comment*]

where *s* is the statement number to receive control if an error condition of the type specified arises, and *comment* is one or more EBCDIC characters.

Action:

If an error occurs during execution of an I/O statement that contains an EXIT clause, the specified EXIT statement is examined. If the error condition appears as a parameter in the EXIT statement, control is transferred to the statement number specified for that error condition. If the error condition is not included in the EXIT statement, program execution is terminated.

Rules:

1. The parameters of the EXIT statement may be specified in any order.
2. At least one parameter must appear in the EXIT statement, and no error condition can be specified more than once.
3. The exact meaning of the error condition varies with the I/O statement during which the error occurred. (See the discussion of the appropriate statement for this information.)
4. The maximum number of EXIT statements permitted in a program is 50.

Example:

```
100    EXIT EOF 200, CONV 250
200    EXIT DUPKEY 500, NOKEY 550, IOERR 700
300    EXIT NOREC 800, DUPREC 830, EOF 550
```

The FNEND Statement

See "The DEF, RETURN, and FNEND Statements."

The FOR and NEXT Statements

Function:

Together, a FOR statement and its paired NEXT statement delimit a "FOR loop"—a set of BASIC statements that may be executed a number of times. The FOR statement marks the beginning of the loop and specifies the conditions of its execution and termination. The NEXT statement marks the end of the loop. For a discussion of how these statements are used, see "More About Loops—Using FOR and NEXT Statements" in Part I of this publication.

General Format:

FOR *av* = x_1 TO x_2 [STEP x_3] [REM *comment*]

.
.
.

NEXT *av* [REM *comment*]

where *av* is a simple arithmetic variable called the control variable, x_1 is an arithmetic expression which assigns an initial value to *av*, x_2 is an arithmetic expression representing the test value that will cause execution of the loop to be terminated, x_3 is an arithmetic expression representing the value of the increment to be added to *av* at the end of each execution of the loop and *comment* is one or more EBCDIC characters. The arithmetic variable *av* must be the same in any given pair of FOR and NEXT statements.

Action:

All expressions (x_1, x_2, and x_3) are evaluated. The initial value x_1 is tested against the final value x_2. If the initial value is greater than (less than, for negative increments) the final value, the loop is not executed; instead, the value of the control variable *av* is left unchanged and control goes to the statement following the NEXT statement.

If the loop is executed, the control variable *av* is set equal to the initial value x_1. The statements in the loop are executed; when the NEXT statement is executed, the specified increment x_3 is added to the control variable *av*, which is then compared with the specified final value x_2. If the control variable *av* is still less than (greater than, for negative increments) or equal to the final value x_2, the loop is executed and the cycle continues until an increment is made which renders the control variable greater than (less than, for negative increments) the specified final value x_2. At that time, the control value is set back to its last value and control "falls through" to the first executable statement following the NEXT statement.

Rules:

1. The value of the control variable *av* may be modified by statements within the FOR loop, but its initial value x_1, final value x_2, and increment x_3 are established during the initial execution of the FOR statement and are not affected by any statements within the FOR loop.
2. If the optional STEP clause is omitted in the FOR statement, the increment value is automatically set to +1.
3. If the STEP option is assigned a value that is contradictory to the increment direction implied by the initial and final values (for example, FOR X = 1 TO 5 STEP -1), the FOR loop is not executed, no value is assigned to the control variable *av*, and execution proceeds from the first executable statement following the associated NEXT statement.
4. If the value of the STEP option is zero, the FOR loop can be exited only if the control variable is set outside of the specified range or if control is transferred out of the FOR loop.
5. Transfer of control into or out of a FOR loop is permitted; a NEXT statement acts as a non-executable statement if its associated FOR statement has not been executed.
6. FOR loops may be nested within one another as long as the internal FOR loop falls entirely within the external FOR loop (see example).
7. The maximum number of FOR loops permitted in a program is 80.
8. The maximum number of levels permitted when FOR loops are nested is 15.

Examples:

The first example shows a simple FOR loop that increments the control variable by 2 until the value 25 is exceeded:

```
20     FOR I = 1 TO 25 STEP 2
            .
            .
            .
95     NEXT I
```

The second example shows the correct technique for nesting FOR loops. The inner loop is executed 100 times for each execution of the outer loop.

```
10     FOR J = A TO B STEP C(1)**3
            .
            .
            .
150    FOR K = 1 TO 100
            .
            .
            .
280    NEXT K
            .
            .
            .
620    NEXT J
```

The FORM Statement

This statement can be used to format print lines with the PRINT USING statement, and to format records with other BASIC statements that process record-oriented files.

FORM With PRINT USING

See "PRINT USING FORM" under "The PRINT Statement."

FORM With Record-Oriented Files

Function:

This statement is used in conjunction with a READ FILE, REREAD FILE, WRITE FILE, or REWRITE FILE statement to specify the formatting of records in record-oriented files. For a discussion of how this statement is used, see "Record-Oriented Files" in Part I of this publication.

General Format:

$$\text{FORM} \left\{ \begin{matrix} \text{"}lit_1\text{"} \\ c_1 \\ [n_1 *]d_1 \end{matrix} \right\} \left[\begin{matrix} \text{"}lit_2\text{"} \\ ,c_2 \\ ,[n_2 *]d_2 \end{matrix} \right] \ldots [\text{REM } comment]$$

where *"lit"* is a literal specified by a quoted character constant , *c* is a control specification as defined below, *d* is a data form specification, as defined below, and *n* is an unsigned, nonzero, integer constant, or an arithmetic variable whose value is greater than zero, that indicates the number of consecutive times that the data form specification is to be interpreted before the next data form or control specification is examined, and *comment* is one or more EBCDIC characters.

A control specification can be either of the following:

X[*n*]
POS[*n*]
SKIP[*n*]

where *n* is an unsigned, nonzero, integer constant, or an arithmetic variable whose value is greater than zero. The SKIP clause is ignored when processed in conjunction with record-oriented READ FILE and WRITE FILE statements.

A data form specification can be any of the following:

B[*w*]
C[*w*]
NC *w* [.*d*]
PD *w* [.*d*]
S
L

$$\text{PIC}(\left\{ \begin{matrix} ds_1 \\ ic_1 \end{matrix} \right\} \left[\begin{matrix} ds_2 \\ ic_2 \end{matrix} \right] \ldots [|||||][tr])$$

where *w* is an unsigned, nonzero, integer constant, *d* is an unsigned, nonzero, integer constant, *ds* is one of the digit specifiers #, Z, *, $, +, or -, *ic* is one of these insertion characters:

,	comma
/	slash
B	blank
.	decimal point

and *tr* is one of these trailing characters:

+	trailing plus
-	trailing minus
$	trailing dollar sign
CR	trailing credit sign
DB	trailing debit sign

Action:

When a READ FILE, REREAD FILE, WRITE FILE, or REWRITE FILE statement with a USING clause is executed, the specified FORM statement is used for conversion of the data items specified in the record I/O statement. Each data item is matched against a data form specification in the FORM statement. If there are more items in the I/O statement than there are data form specifications in the FORM statement, the FORM statement is reused from its beginning until all of the items in the I/O statement have been exhausted. Excess data form specifications are ignored. Arrays in the I/O statement are handled row by row.

Control Specifications: Control specifications may be interspersed with data form specifications in the FORM statement to pass over unwanted parts of a record.

If POS *n* is specified, the record pointer is set at position *n* of the record. The value *n* must be between 1 and the length of the current record. The position specified can be either to the right or to the left of the current position in the record. If an arithmetic variable is used to specify *n*, the truncated integer portion of the value is used to determine the record position. If POS alone is specified (without a position number), it is assumed that *n* has a value of 1.

If X*n* is specified, the record pointer spaces forward *n* positions in the record. If an arithmetic variable is used to specify *n*, the truncated integer portion of the value is used to determine the number of forward spaces. If X alone is specified (without a number), it is assumed that *n* has a value of 1.

Data Form Specifications: Data form specifications define the exact format of each item in the record. A single data form specification can be interpreted *n* consecutive times by prefixing it with an *n** replication factor. If an arithmetic variable is used to specify *n*, the truncated integer portion of the value is used to determine the number of consecutive interpretations of the data form specification.

The specification B[*w*] is used to allow access to fixed-point binary integer data in the record. The values of *w* can be 2, 4, or 8 depending on the length of the numeric data to be converted.

- For input, the next *w* positions of the record contain a binary value that is to be converted to a VS BASIC arithmetic value and moved to the corresponding arithmetic variable in the READ FILE or the REREAD FILE statement. If *w* is omitted, 4 positions or a length of 4 is assumed.

- For output, an arithmetic value for an arithmetic expression in the WRITE FILE or REWRITE FILE statement is converted to a truncated fixed-point binary integer format field of length *w* and placed in the record. If *w* is omitted, 4 positions or a length of 4 is assumed.

The specification C[*w*] indicates that the item in the record is a character value. Its exact meaning varies for input and output operations:

- For input, the next *w* characters from the record are moved to the corresponding character variable in the input list of the READ FILE or REREAD FILE statement. If the length of the character variable is greater than *w*, the value extracted from the record is padded on the right with blanks to the length of the variable. If the length of the character variable is less than *w*, a number of characters equal to the length of the variable is moved from the record, and the remaining record positions are passed over until *w* characters have been either moved or spaced over. If *w* is omitted, a number of characters equal to the length of the variable is moved from the record.
- For output, the next *w* characters placed in the record will be from the value of a character expression in the WRITE FILE or REWRITE FILE statement. If the length of the character value is less than *w*, the value will be padded on the right with blanks to a length of *w* before being written in the record. If the length of the character value is greater than *w*, only the first *w* characters in the string will be written in the record. If *w* is omitted, the number of characters placed in the record will equal the length of the character value.

The numeric conversion specification NC *w* [.*d*] is used for a numeric data item in the record as follows:

- For input, the next *w* positions of the record contain a numeric value that will be converted to internal arithmetic representation. In the record, the numeric value is a string consisting of digits in combination with any of the characters $, +, -, *, /, ƀ, comma, decimal point, or exponential notation (E or D ± numeric constant). Zoned decimal fields can also be read. The constant *d* indicates the number of decimal positions in the field, and, if present, will override an explicit decimal point in the input field. If this causes the position of the decimal point to change, the value is also thereby changed.
- For output, an arithmetic value from an expression in the WRITE FILE or REWRITE FILE statement is converted to a signed, zoned decimal field of length *w* and placed in the record. If *d* is specified, *d* decimal positions will be present in the record field; otherwise, all *w* positions will represent the integer portion of the arithmetic value.

The specification PD *w* [.*d*] is used for a numeric data item in packed decimal format in the record as follows:

- For input, the next *w* positions of the record contain a numeric value in packed decimal format (two digits per position, except for the last position which contains one digit and a sign). The item is to be converted to internal arithmetic representation and moved to the corresponding arithmetic variable in the READ FILE or REREAD FILE statement. The constant *d* indicates the number of decimal positions in the field; if it is omitted, the field is assumed to have no decimal positions.
- For output, an arithmetic value from an expression in the WRITE FILE or REWRITE FILE statement is converted to a packed decimal field of

length *w* and placed in the record. If *d* is specified, *d* decimal positions will be present in the record field; otherwise, all *w* positions will represent the integer portion of the arithmetic value.

The specification S indicates that the item in the record occupies four positions (short form) and is a numeric value in internal format:

- For input, the short form numeric value in the record is moved without conversion to the arithmetic variable specified in the input list of the READ FILE or REREAD FILE statement. If long form precision has been specified for the executing program, the value is extended on the right with zeros before being assigned to the variable.
- For output, an arithmetic value from an expression in the output list of the WRITE FILE or REWRITE FILE statement is written in the record in short form precision without conversion.

The specification L indicates that the item in the record occupies eight positions (long form) and is a numeric value in internal format:

- For input, the long form numeric value in the record is moved without conversion to the arithmetic variable specified in the input list of the READ FILE or REREAD FILE statement. If short form precision has been specified for the executing program, the value is truncated on the right before being assigned to the variable.
- For output, an arithmetic value from an expression in the output list of the WRITE FILE or REWRITE FILE statement is written in the record in long form precision without conversion.

The PIC specification is a string of digit specifiers and/or insertion characters, optionally followed by an exponent specifier. It can refer to arithmetic or character data. When used with arithmetic data, the field is edited as indicated by the PIC specification. When used with character data, the total number of positions in the PIC specification indicates the length of the character data. The PIC specification can be used for output operations only.

With respect to the printing of arithmetic data, the following rules apply:

1. Digit specifiers can be conditional or unconditional. They are:

   # This character is placed in any position that must always contain a numeric digit.

   Z This character causes a leading zero in the associated position to be replaced by a blank.

   * This character causes a leading zero in the associated position to be replaced by an asterisk.

   $ This character is placed in each position that can potentially be occupied by a floating dollar sign—that is, a dollar sign to the immediate left of the first significant digit. Nonsignificant zeros are suppressed.

   + This character is placed in each position that can potentially contain a floating high-order sign. The appearance of either a plus sign or a minus sign is guaranteed in the record. Nonsignificant zeros are suppressed.

- This character is placed in each position that can potentially contain a floating high-order minus sign if the value in the record is negative. Nonsignificant zeros are suppressed.

2. Insertion characters are conditional or unconditional. They are:

   B This unconditional character always causes a blank to be inserted in the corresponding position of the record.

   , This character is inserted in the corresponding position of the record unless zero suppression is in effect and no significant digits appear to the left of the comma in the record. In this case, the comma will be replaced by a floating or zero suppression character.

   / This character is inserted in the corresponding position of the record unless zero suppression is in effect and no significant digits appear to the left of the slash in the record. In this case, the slash will be replaced by a floating or zero suppression character.

   This character is inserted in the corresponding position of the record unless zero suppression has been specified for every digit position and the value is zero. In this case, the decimal point will be replaced by a floating or zero suppression character.

3. The exponent specifier | | | | causes the following sequence of characters to be placed in the corresponding positions of the record:

   a. The letter E

   b. The exponent sign (plus or minus)

   c. Two digits representing the value of the exponent.

   These characters do not appear in the record when zero suppression is in effect and the value to be placed in the record equals zero.

4. Trailing characters are conditional. They are:

trailing +
This character causes a plus sign or a minus sign to be inserted in the corresponding position of the record unless zero suppression is in effect and no significant digits appear to the left of the sign in the record. In this case, the sign will be replaced by an asterisk or a blank.

trailing —
This character is inserted in the corresponding position of the record if the value to be displayed is negative, unless zero suppression is in effect and no significant digits appear to the left of the minus sign in the record. In this case, the minus sign will be replaced by an asterisk or a blank.

trailing $
This character is inserted in the corresponding position of the record unless zero suppression is in effect and no significant digits appear to the left of the dollar sign in the record. In this case, the dollar sign will be replaced by an asterisk or a blank.

trailing credit CR
These alphabetic characters are inserted in the corresponding positions of the record if the value to be displayed is negative. The two alphabetic characters are replaced by two blanks if the value is positive.

trailing debit DB
These alphabetic characters are inserted in the corresponding positions of the record if the value to be displayed is negative. If the value is positive, the two alphabetic characters are replaced by two blanks.

Rules:

1. The maximum number of FORM statements permitted in a single BASIC program is dependent on the number of Image statements also present. Together, they may not exceed 50.
2. FORM statements are nonexecutable and may be placed anywhere in a BASIC program, either before or after the I/O statements that refer to them. However, they may not appear within a multiline function definition (that is, between a DEF statement and its associated FNEND statement).
3. A PIC specification in a FORM statement may not contain both the Z and * digit specifiers.
4. A PIC string must be from 1 to 32 characters long.
5. A single $, +, or - as the leftmost character in a PIC string is treated as a static character. Two or -ore $, +, or - signs at the leftmost end of a PIC string are treated as floating characters. The same character cannot appear as both a static character and part of a floating character string in a single PIC string.
6. A string of floating characters must contain at least one more floating character than the maximum number of expected digits in the output field.
7. A PIC string cannot end with a B, slash (/), or comma (,) insertion character.
8. A PIC string cannot begin with a slash (/) or comma (,) insertion character.
9. There cannot be more than one decimal point (.) insertion character in a PIC string.
10. A PIC string must include at least one #, Z, *, or floating string.
11. No # digit specifiers may appear to the left of a zero suppression character or a floating character.
12. A # digit specifier may not appear in a PIC string that contains a decimal point followed by zero suppression or floating characters.
13. The symbols +, -, CR, and DB cannot appear in the same PIC string.
14. A trailing character may not appear in a PIC string in which that trailing character is used as either a static character or as part of a floating character string.
15. Blanks replace any unused portion of the PIC specification resulting from a character data assignment.
16. If the length of the character string exceeds the length of the PIC specification, the character string is truncated on the right to the length of the PIC specification.
17. The SKIP clause is ignored if specified in the FORM statement when processed in conjunction with record-oriented input/output statements.

Examples:

```
25  READ FILE USING 30 'FILEA', KEY=N$, A$, G
30  FORM X25, C, X10, NC7
```

The record whose key satisfies the value in N$ is to be read. Two values from the record are to be put into the input list variables described in the READ FILE statement. One is a character value, to be placed into A$, the other is a numeric value, to be placed into G. The FORM statement says that the first 25 positions of the record are to be skipped and the number of characters equal in length to the variable A$ (eighteen by default in this example), are to be read from the record into A$. Ten more positions are to be skipped, and the next seven positions in the record are to be converted and read into the numeric variable G.

```
110 REWRITE FILE USING 100 'FILEA', A$, M
100 FORM X25, C, POS150, PIC(Z##.##)
```

Two values from the input list in the REWRITE FILE statement are to be entered into a record in the file FILEA. The first 25 positions in the record are to be skipped, a character value equal in length to the variable A$ is to be inserted into the record, and, in position 150 of the record, the numeric value in M is to be inserted according to the PIC specification.

The GET Statement

Function:

The GET statement causes values to be assigned to variables from a specified stream-oriented file. For a discussion of how this statement is used, see "Retrieving a File" in Part I of this publication.

General Format:

[MAT] GET *filename, input-list* [,EXIT *es*] [,EOF *s*][,CONV *s*][,IOERR *s*] [REM *comment*]

where *filename* is a character expression, *es* is the number of an EXIT statement, *s* is the number of any executable statement, and *comment* is one or more EBCDIC characters.

Action:

If it has not been specified in a previously executed OPEN statement, a stream-oriented file is activated for input by the first execution of a GET statement specifying its file name. The file is positioned at its beginning and values are assigned from it to the items specified in the input list of the GET statement. Values are assigned to arrays row by row. Subsequent GET statements for the same file cause values to be assigned beginning at the current file position.

Note: Data items in the file can be separated by an unquoted blank string of one or more blank characters, or by a comma.

If the EOF clause is included in the GET statement, control is transferred to the specified statement if the input file is exhausted before all items in the input list are filled.

If the CONV clause is included, control is transferred to the specified statement if a conversion error occurs (for example, if an attempt is made to read character data with a numeric variable.)

If the IOERR clause is included, control is transferred to the specified statement if a device malfunction prevents reading of an item in the input file.

If the EXIT clause is included, the specified EXIT statement is examined for the appropriate error condition if any of the above problems occur.

Subscripted references to array members in the input list are evaluated as they occur, from left to right. Thus, an assigned variable in a GET statement may be used subsequently as the subscript of another variable in the same GET statement.

Arithmetic values are assigned in the form (long or short) specified for the program in which the GET statement appears. Thus, arithmetic values from a file that was created in long form are assigned in short form arithmetic in a program using short form precision. Likewise, arithmetic values from a file that was created in short form are assigned in long form arithmetic in a program using long form precision.

If a redimension specification follows an array name in a GET statement, the truncated integer portion of each expression value is used to redimension the array immediately before any data values from a file are assigned to that array.

A file is deactivated in response to a CLOSE statement or at the end of program execution.

Rules:

1. A file currently activated as an output file may not be specified in a GET statement. It must first be closed.
2. Each value assigned in a GET statement must be of the same data type (character or arithmetic) as the corresponding item in the input list.
3. If the input file is exhausted before all items in the input list are filled, program execution is terminated unless an EOF clause, or an EXIT clause pointing to an EXIT statement with an EOF clause, is specified.
4. The EOF, CONV, and IOERR clauses can be specified in any order. The presence of any of these clauses in the GET statement precludes the specification of an EXIT clause.
5. If all of the items in the input list are array references (that is, array names preceded by the keyword MAT), the MAT can be dropped from each of the array references and specified once before the keyword GET.
6. Consecutive commas (null items) are not allowed between data items within stream-oriented files.
7. Naming conventions for files depend upon the environment in which they are used. See Appendix A. "Implementation Considerations."

Examples:

```
70  GET 'ABF', X, Y, MAT Z(100), A(4), A(5), D$, E$
80  MAT GET 'BCD', A, B(10), Z(5,20)
```

The GOSUB and RETURN Statements

Function:

GOSUB and RETURN statements are used together in the creation of subroutines. The GOSUB statement transfers control, conditionally or unconditionally, to a specified statement. The RETURN statement transfers control to the first executable statement following the last active GOSUB statement executed. A GOSUB becomes inactive when a RETURN is executed for it. For a discussion of how these statements are used, see "Subroutines" in Part I of this publication.

General Format:

The GOSUB statement may be written in either of two forms, simple or computed.

Simple:

GOSUB *s* [REM *comment*]

Computed:

GOSUB s_1 [, s_2]... ON *e* [REM *comment*]

where *s* is the number of a statement to which control is to be transferred *e* is an arithmetic expression, and *comment* is one or more EBCDIC characters.

The RETURN statement has the following format:

RETURN [[REM] *comment*]

where *comment* is one or more EBCDIC characters. The REM keyword must be specified on a RETURN statement comment that is in a multiline function definition (that is, a RETURN statement that appears between a DEF statement and its corresponding FNEND statement).

Action:

Execution of a simple GOSUB statement causes an unconditional transfer of control to the statement whose number is specified.

Execution of a computed GOSUB statement causes the arithmetic expression *e* to be evaluated and control transferred to the statement whose numerical position in the list of statement numbers (reading from left to right) is equal to the truncated integer value of the expression. Thus an expression with a value of 2.75 would cause control to be transferred to the second statement in the list. If the expression has a value less than 1 or greater than the total number of statements listed, control "falls through" to the first executable statement following the computed GOSUB.

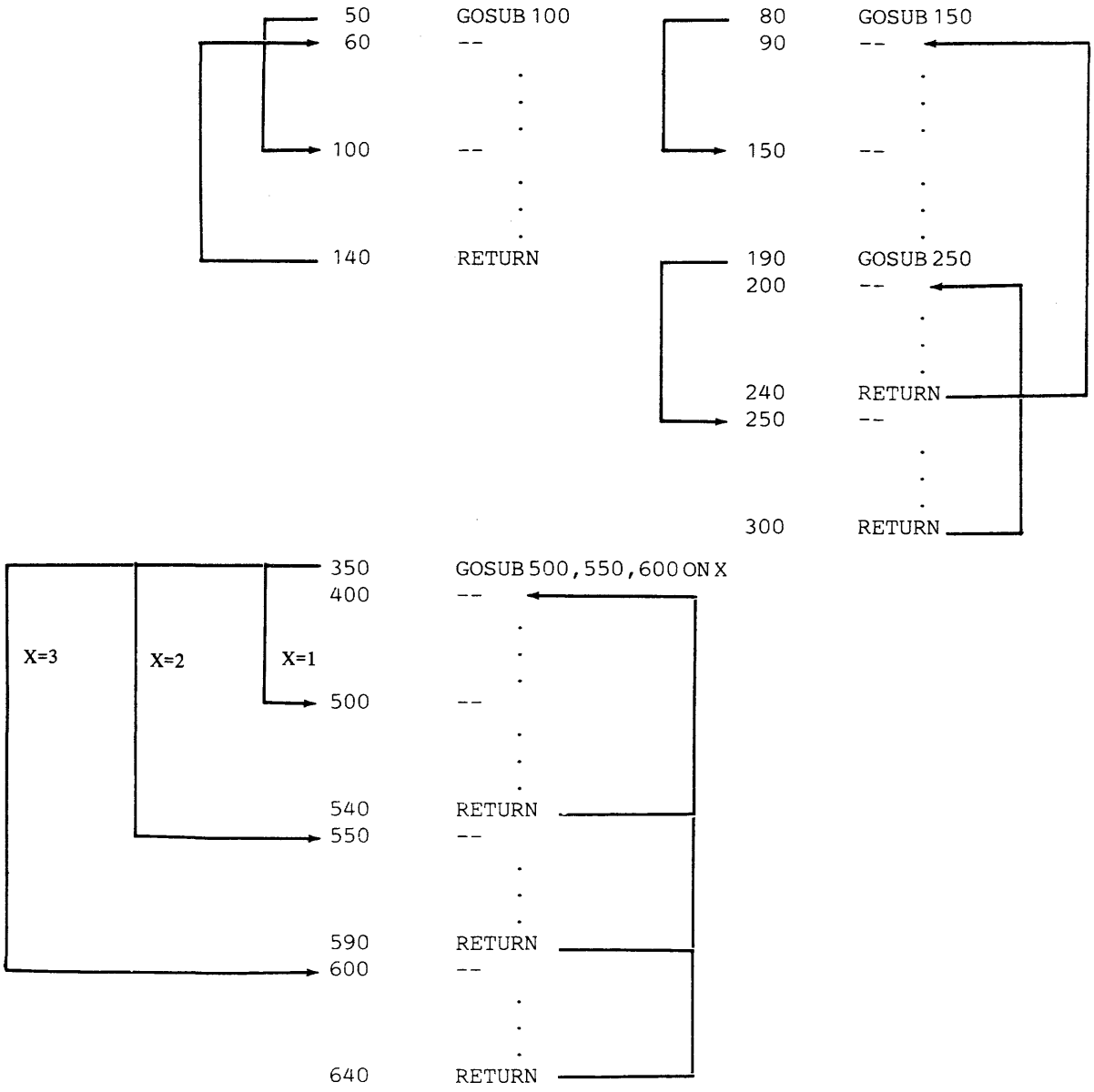
When a simple or computed GOSUB statement causes control to be transferred to a nonexecutable statement, control is passed to the first executable statement following the specified statement.

Execution of the RETURN statement causes an unconditional transfer of control to the first executable statement following the last active GOSUB statement executed.

Rules:

1. More than one GOSUB statement may be executed before a RETURN statement is executed, but whenever a RETURN statement is executed, there must be at least one active GOSUB statement (that is, an already executed GOSUB statement for which a corresponding RETURN statement has not been executed).
2. The maximum number of simultaneously active GOSUB statements is 94. (This number may be reduced if nested function references occur within a GOSUB/RETURN group. See "The DEF, RETURN, and FNEND Statements" for details.)
3. A GOSUB statement may not transfer control into or out of a multiline function definition.

Examples:



The GOTO Statement

Function:

The GOTO statement transfers control, either conditionally or unconditionally, to a specified statement. For a discussion of how this statement is used, see "Loops" and "The Computed GOTO Statement" in Part I of this publication.

General Format:

The GOTO statement may be written in either of two forms, simple or computed.

Simple:

GOTO *s* [REM *comment*]

Computed:

GOTO s_1[s_2]...ON *e* [REM *comment*]

where *s* is the number of a statement to which control is to be transferred, *e* is an arithmetic expression, and *comment* is one or more EBCDIC characters.

Action:

Execution of a simple GOTO statement causes an unconditional transfer of control to the statement whose number is specified.

Execution of a computed GOTO statement causes the arithmetic expression *e* to be evaluated and control transferred to the statement whose numerical position in the list of statement numbers (reading from left to right) is equal to the truncated integer value of the expression. Thus, an expression with a value of 2.75 would cause control to be transferred to the second statement in the list. If the expression has a value less than 1 or greater than the total number of statement numbers listed, control "falls through" to the first executable statement following the computed GOTO statement.

When a simple or computed GOTO statement causes control to be transferred to a nonexecutable statement, control is passed to the first executable statement following the one specified.

Rules:

1. A GOTO may not transfer control into or out of a multiline function definition.
2. The maximum number of nonexistent line numbers that may be referred to is 24.

Examples:

The following statement will pass control to statement number 20:

```
100  GOTO 20
```

When X=4, the following statement will pass control to statement number 60:

```
50   GOTO 40,60,15,100 ON (X+4)/4
```

The IF Statement

Function:

The IF statement causes program action to be determined as the result of the evaluation of a logical expression. For a discussion of how this statement is used, see "Using the IF Statement" in Part I of this publication.

General Format:

$$\text{IF logical-expression} \left\{ \begin{matrix} \text{GOTO} & s_1 \\ \text{THEN} & \{s_1 \mid t_1\} \end{matrix} \right\} [\text{ELSE}\{s_2 \mid t_2\}\ ]\ [\text{REM } \textit{comment}]$$

where s is the number of a statement to which control can be transferred and t is any of the following statements:

any kind of assignment	PRINT
CHAIN	PRINT TO
CLOSE	PUT
DELETE FILE	READ
GET	READ FILE
GOSUB	REREAD FILE
GOTO	RESET
INPUT	RESTORE
INPUT FROM	RETURN
LET	REWRITE FILE
ON	STOP
OPEN	WRITE FILE
PAUSE	

and *comment* is one or more EBCDIC characters.

Action:

When an IF statement is executed, the logical expression is evaluated. If it is true, either of the following occurs:

- Control is transferred to the statement number specified in the GOTO or THEN clause.
- The statement physically included in the THEN clause is executed. Unless this statement itself transfers control to another statement, control is then passed to the first executable statement following the IF statement.

If the relationship is not true, one of the following occurs:

- Control is transferred to the statement specified in the ELSE clause.
- The statement physically included in the ELSE clause is executed. Unless this statement itself transfers control to another statement, control is then passed to the first executable statement following the IF statement.
- When no ELSE clause is present, control is passed to the first executable statement following the IF statement.

In the event that a statement specified to receive control is nonexecutable, control is passed to the first executable statement following the specified statement.

Rules:

1. The expressions being compared within the logical expression must contain data of the same type (character or arithmetic).
2. The THEN and GOTO clauses are interchangeable in the IF statement. Either may be used, but not both.
3. An IF statement may not transfer control into or out of a multiline function definition.

Examples:

```
30   IF A(3) ≠ X+2/Z THEN 85
40   IF R$ > "CAT" GOTO 70
50   IF S2 = 37.222 THEN N=0  ELSE N=1
60   IF X .GT. Y THEN X = Y ELSE 90
70   IF R < S & A$ = 'CAT' THEN 100 ELSE A = B
```

The Image Statement

See "PRINT Using Image" under "The PRINT Statement."

Note: Comments are not allowed on the Image statement.

The INPUT Statement

Function:

The INPUT statement allows the BASIC user to assign values to variables from the terminal at execution time. For a discussion of how this statement is used, see "Getting Data Into the Computer" in Part I of this publication.

General Format:

[MAT] INPUT *input-list* [REM *comment*]

Action:

When an INPUT statement is encountered at execution time, it causes a question mark to be printed out at the terminal and program execution to be temporarily interrupted. The user then enters a set of values which are assigned, in order of appearance, to the items specified in the input list. When the complete set has been entered, program execution resumes.

Subscripted references to array members in the input list are evaluated as they occur; thus, an assigned variable in an INPUT statement may be used subsequently as the subscript of another variable in the same statement.

If a redimension specification follows an array name in the input list, the truncated integer portion of each expression value is used to redimension the array before data values are entered.

A character constant entered at the terminal that is shorter than the variable in the input list is padded on the right with blanks to the length of the variable before being assigned. A character constant longer than the corresponding variable is truncated on the right to the length of the variable before being assigned. A character constant containing no characters (null) is assigned as all blank characters.

When an INPUT statement is executed immediately after a PRINT statement in which the final delimiter is a comma or semicolon, the question mark generated by the INPUT statement is printed following the last data item on the same print line. In all other instances, the question mark appears as the first character on the next print line.

Buffered-Ahead Terminal Input

The buffered-ahead terminal input facility allows groups of data values to be entered on a single line, to satisfy the request of more than one consecutively executed INPUT statement. This facility is invoked by entering a semicolon directly after the last data item in response to an INPUT statement request. Additional data items can then be entered on the same input line to satisfy the requests of subsequent input statements.

When buffered-ahead terminal input is active, subsequent INPUT statements retrieve input data until a carrier return is encountered, indicating there is no more data.

The internal variable &BUFF is available for use with the buffered-ahead facility and contains the number of unprocessed groups of input data items remaining to satisfy subsequent INPUT statement requests. When buffered-ahead terminal input is not active the internal variable, &BUFF, contains the value zero. A test can be made using &BUFF, to determine the amount of input data available by using a statement such as:

```
IF &BUFF r exp
```

where *r* is a relational operator and *exp* is an arithmetic expression.

The internal variable &BUFF is also used with the RESET statement. RESET &BUFF results in the following actions taken:

1. The value of &BUFF is reset to zero.
2. All groups of data values (identified by a preceding semicolon) not yet accessed by an INPUT statement are discarded. For example, if the following input line were entered in response to an initial INPUT statement request,

   ```
   1,2,3;4,5,6;7,8,9
   ```

the execution of RESET &BUFF before another INPUT statement is executed will result in the data value groups of 4,5,6 and 7,8,9 being discarded.

Rules:

1. Each value entered must be of the same data type (character or arithmetic) as the corresponding item in the input list. Data types may be mixed in the same statement.
2. Each value entered must be separated from the next by a comma. Two consecutive commas are treated as a null entry, that is, the value of the corresponding item in the input list is unchanged. When the same input value is to occur several times in succession, the following shorthand notation can be used:

   *j**value

where *j* is an unsigned, nonzero, integer constant specifying the number of consecutive times the value occurs. Multiple null entries can be specified by the notation:

   *j**

   where *j* is the number of consecutive null entries.

3. Values are entered one line at a time. If the current input list item is a scalar, a single question mark is printed, and as many values should be entered as there are consecutive scalar items in the I/O list. If the current item is an array, a single question mark is printed, and enough values to fill one row of the array should be entered. A double question mark is then printed before new values are accepted for additional rows of the array.

   If a line is filled before all expected values (scalars or array row members) are entered, a comma at the end of the current line will allow additional values to be entered on a new input line.

4. A character constant in the input stream must be bounded by a pair of single or double quotation marks, if the constant:

- contains commas, semicolons or slashes
- has leading or trailing blanks or tabs
- has leading single or double quotes
- starts with an integer immediately followed by an asterisk

5. The number of values entered at execution time should be equal to the number of items specified in the input list of the INPUT statement except when using buffered-ahead terminal input. However, a slash not enclosed

in quotation marks can be entered in the input line to cause all remaining items in the input list to be left unchanged.

6. Transmission of input values continues until each item in the input list has been either filled or explicitly left unchanged.
7. If all of the items in the input list are array references (that is, array names preceded by the keyword MAT), the MAT can be dropped from each of the array references and specified once before the keyword INPUT.
8. Buffered-ahead terminal input is limited to the total number of INPUT statements that can be satisfied by input data entered on a single input line. The maximum number of data values that can be entered cannot exceed the width of the terminal input line.
9. The procedure for retry or re-entry of data after an error is described in your *Terminal User's Guide* or *Programmer's Guide.*

Example:

```
10      INPUT A$, R(3),X,Y(X),MAT Q(6),MAT B(2,2)
?
"DOG", 4E-7,8,.013
?
6*10
?
2*3.0
??
2*3.0
```

Buffered-Ahead Terminal Input Example:

```
100       PRINT 'ENTER EMPLOYEE SERIAL NUMBER'
200       INPUT M
300       IF &BUFF .NE. 0 GOTO 500
400       PRINT 'ENTER HOURS AND HOURLY RATE'
500       INPUT H,R
600       IF R .LT. 5 .OR. R .GT. 10 GOTO 2000
700       IF &BUFF .NE. 0 GOTO 900
800       PRINT 'ENTER NAME'
900       INPUT N$
1000      PUT 'FILEZ', M, H, R, N$
1010        .
1020        .
1030        .
2000      PRINT 'ERROR IN RATE - REKEY'
2100      RESET &BUFF
2200      INPUT R
2300      GO TO 600
2400      END
```

Normal execution:

```
ENTER EMPLOYEE SERIAL NUMBER
?
123
ENTER HOURS AND HOURLY RATE
?
37,7.5
ENTER NAME
?
joe
ENTER EMPLOYEE SERIAL NUMBER
?
```

Buffered-Ahead Facility:

```
ENTER EMPLOYEE SERIAL NUMBER
?
123;37,7.5;joe
ENTER EMPLOYEE SERIAL NUMBER
?
```

The INPUT FROM Statement

Function:

The INPUT FROM statement allows data normally requested from the terminal to be retrieved from a entry-sequenced record-oriented file or from the terminal.

General Format:

INPUT FROM *filename* [REM *comment*]

where *filename* is a character expression and *comment* is one or more EBCDIC characters.

Action:

If an entry-sequenced record-oriented data file has not been previously opened by an OPEN FILE statement, the INPUT FROM statement opens and activates that file as an input file and causes subsequent [MAT]INPUT statements to obtain values from that file, starting at the beginning. A file is deactivated by a CLOSE FILE statement or at the end of program execution.

If the value of the character expression specified in the INPUT FROM statement consists of one or more blanks, subsequent INPUT statements accept values from the user's terminal. The INPUT FROM statement does not close any previously activated data file.

Rules:

1. Only entry-sequenced record-oriented data files can be accessed as a result of the execution of the INPUT FROM statement. Each record in the file must consist of data as it would have been entered on a single line at a terminal. Each line contains EBCDIC values delimited by commas or semicolons.
2. The INPUT FROM statement can only be associated with one file at any one time.
3. INPUT, CLOSE FILE, and RESET FILE are the only VS BASIC statements that can refer to a record-oriented file that has been opened for access by the INPUT FROM statement. Conversely, any record-oriented file opened for accessing by a READ FILE/REREAD FILE statement may not be accessed by an INPUT FROM statement.
4. Errors that occur with input data from a file can be handled by the ON INERR or ON ERR statements.
5. In a file where the first item in the records may be a line number, the line number is passed to the program as the first variable in the I/O list.

Example:

```
 10   PRINT 'ENTER FILENAME FOR THE INPUT DATA'
 20   INPUT A$
 30   OPEN FILE A$ IN
        .
        .
        .
 80   INPUT FROM A$
 90   MAT INPUT A, B
100   INPUT FROM ' '
110   PRINT 'INPUT FROM FILE' ;A$; ' COMPLETED'
120   PRINT 'ADDITIONAL DATA WILL BE INPUT FROM TERMINAL'
130   INPUT G1, G2
```

This program requests the value for A$ to be entered from the terminal and causes the values for the arrays A and B to be entered from the data file whose name is taken from A$. The values for G1 and G2 are entered from the terminal.

The LET Statement (Scalar Assignment Statement)

Function:

The LET statement assigns the value of an expression to one or more variables. For a discussion of how this statement is used, see "Getting Data Into the Computer" and "Using Arrays" in Part I of this publication.

General Format:

[LET] $v_1[,v_2]$ ... = *exp* [REM *comment*]

where *v* is a scalar variable, a subscripted array member reference or a pseudo-variable, *exp* is an expression, and *comment* is one or more EBCDIC characters.

Action:

The expression *exp* is evaluated once; then the resulting value is assigned to the specified variable list from left to right.

A character value containing fewer characters than the variable receiving it is padded on the right with blanks to the length of the variable before being assigned. A character value containing more characters than the variable receiving it is truncated on the right to the length of the variable before being assigned. Character constants containing no characters (null) are assigned as all blank characters.

Rules:

1. Data values to the right of the equal sign must be of the same type (arithmetic or character) as the variables to which they are assigned.
2. Subscripted references to array members are permitted in the scalar assignment statement, but unsubscripted array references may appear only in the array assignment statement (see "Array Assignment Statement").

Examples:

```
10    LET Z$ = "CAT"
20    LET X = 9
30    LET Y(X) = 2
40    X, Y(X) = X/Y(X)
```

After execution of statement 10, the character variable Z$ will contain the word CAT followed by 15 blank characters (Z$ has an implicitly defined length of 18 characters).

After execution of statement 20, the arithmetic variable X will have the integer value 9.

After execution of statement 30, the ninth member of the one-dimensional arithmetic array Y will have the integer value 2.

After execution of statement 40, the arithmetic variable X will have the decimal value 4.5, as will the fourth member of the one-dimensional array Y. The action of the assignment statement in statement 40 is to first evaluate the expression on the right according to the current values of the variables X and Y(X), 9 and 2, respectively. The resulting value, 4.5, is then assigned to the variable X. The new value of X, 4.5, is then used in the evaluation of the subscript of the array variable Y(X), for which purpose only the truncated integer portion, 4, is considered. Thus, the fourth member of array Y is set to the expression value 4.5.

If statement 40 had been LET Y(X),X = X/Y(X), the resulting values would have been 4.5 for the ninth member of array Y and for the variable X.

The NEXT Statement

See "The FOR and NEXT Statements."

The ON Statement

Function:

The ON statement allows the user to specify certain conditions which, when they occur, will be handled by the program itself rather than cause program termination. (See 'Program Error Handling' in Part I of this publication for a discussion of how this statement is used.)

General Format:

ON { OFLOW / UFLOW / ZDIV } { IGNORE / [GOTO | THEN] *s* / SYSTEM } [REM *comment*]

ON { ATTN / INERR / ERR } { [GOTO | THEN] *s* / SYSTEM } [REM *comment*]

where *s* is the number of any statement, and *comment* is one or more EBCDIC characters.

OFLOW applies to the condition of arithmetic overflow.

UFLOW applies to the condition of arithmetic underflow.

ZDIV applies to the condition of division by zero.

ATTN applies to the condition associated with a terminal "attention" key or equivalent.

INERR applies to the condition for errors arising from input in response to the INPUT statement.

ERR applies to those errors not covered by any of the above clauses or by errors local to input/output statements (that is, EOF, IOERR, CONV).

Action:

A condition can be specified in an ON statement and be activated by executing that ON statement. When, subsequently, the condition occurs during program execution, the action taken depends upon the clause specified. If *s,* GOTO *s,* or THEN *s* is specified, then control is transferred to statement *s* when the condition arises.

If SYSTEM is specified, then the system defaults are taken. This clause is used to reset a previous ON statement including a GOTO, THEN, or IGNORE clause, that is, it deactivates an ON condition.

The IGNORE clause applies only to those ON statement which handle arithmetic error conditions (OFLOW, UFLOW, and ZDIV) and suppresses the printing of the messages 'OVERFLOW', 'UNDERFLOW', or 'DIVISION BY ZERO' which occur when the system substitutes the maximum machine magnitude value (or zero in the case of UNDERFLOW), and carries on. The clause allows a user to suppress a message which might otherwise occur in the middle of a printed report.

ON ATTN does not suppress the normal attention-interrupt message that is issued when the attention key is pressed, but it will allow the user to specify a statement which can receive control instead of resuming normal execution.

The ATTN (attention) clause is used in conjunction with a terminal "attention" key or equivalent. The ON ATTN statement, if active, provides for alternative courses of action by the user when an "attention" signal is sent from the terminal.

The following choices can be made when the message is printed at the terminal:

Press C/R
(carrier return or equivalent)
resumes program execution at the line number specified by an active ON ATTN statement.

Enter RESume
continues program execution at point of interruption, overriding any action ON ATTN statement.

Any other keyed user input causes program execution to terminate.

The INERR clause will override the normal system action for those errors occurring in response to an INPUT statement. Such input errors include the entering of too much data, too little data, or a data conversion error.

ON ERR will handle all program errors for which no other action has been specified through an ON ATTN, ON OFLOW, ON UFLOW, ON ZDIV, or ON INERR, or through local error clauses on specific input/output statements and associated EXIT statements.

Rules:

1. Any of the error clauses on I/O statements: EXIT, IOERR, CONV, EOF, NOKEY, DUPKEY, NOREC, and DUPREC take precedence for that I/O statement over an active ON ERR statement.
2. Errors occurring on I/O statements that do not have the appropriate error clause will be handled by an active ON ERR statement.
3. Once the ON statement is executed and becomes active, it applies to the execution of all subsequent statements in your program except within user-defined functions; a user-defined function can, however, have its own ON statements. The same rules apply to the ON statement as to the GOTO statement, that is, it is not permitted to transfer control into or out of a user-defined function.
4. Any active ON statement is superseded by the execution of another ON statement containing the same condition clause.
5. The ON statement cannot be specified as a conditionally executable statement within an IF statement. However, an ON statement can be a statement referred to by the GOTO or THEN clause of the IF statement.

Examples:

```
100 ON ATTN GOTO 1000
200 ON ERR THEN 4000
300 ON OFLOW IGNORE
400 ON INERR SYSTEM
```

Internal Variables:

Four internal variables, &ERR, &LINE, &CODE, and &FILE, are available for further identifying error conditions at time of execution. These are:

&ERR
This arithmetic internal variable will contain a specific error number as a result of the occurrence of a particular error during execution. Error numbers assigned are those appearing in the ICDnnn message identification.

&LINE
This arithmetic internal variable will contain the line number of the statement at which an error occurred.

&CODE
This arithmetic internal variable will contain the code returned by the host system when a VSAM error occurs; otherwise, &CODE will contain zero.

&FILE
This internal variable will contain the name of the file associated with the error condition raised.

Action:

At the beginning of program execution, the internal variables &ERR, &LINE, and &CODE contain zero; &FILE contains blanks. After any error, handled either by an ON statement or by the error clauses associated with I/O statements, these variables can be tested to determine the type of error, the possible VSAM error code, the line number of the statement at which the error occurred, and the name of the file associated with the error (where applicable). The four internal variables always contain values associated with the last (current) error occurrence.

Rules:

1. &ERR, &LINE, &CODE, and &FILE cannot be assigned values by the user. They are "read-only" internal variables.

Example:

```
 100     ON ERR GOTO 1000
             .
             .
             .
1000     IF &ERR = nnn GOTO 500
1010     PRINT 'PROGRAM ERROR AT LINE '; &LINE
```

where nnn is the number associated with the ICDnnn message identification identifying a specific error.

The OPEN Statement

Function:

The OPEN statement causes input and output files to be activated. For a discussion of how this statement is used, see "Activating and Deactivating Files" for stream-oriented files, and "Opening and Closing Files" for record-oriented files, in Part I of this publication.

General Format—Stream-Oriented Files:

OPEN *filename*$_1$ { **IN** / **OUT** } [*filename*$_2$ . . .] [**,EXIT** *es* / **,IOERR** *s*] [**REM** *comment*]

General Format - Record-Oriented Files:

OPEN FILE *filename*$_1$ { **IN** / **OUT [REUSE]** / **ALL [HOLD][REUSE]** } [*,filename*$_2$ . . .]

[**,EXIT** *es* / **,IOERR** *s*] [**REM** *comment*]

where *filename* is a character expression, *es* is the EXIT statement number, *s* is a statement number, and *comment* is one or more EBCDIC characters.

Action:

When the OPEN statement is executed, the specified file or files are activated as indicated. If IN is specified, the file is activated for input only. If OUT is specified, the file is activated for output only. If ALL is specified, the file is a record-oriented file activated for input, output, and updating operations. The inclusion of the REUSE keyword specifies a record-oriented file as reusable and positions the file at its beginning for the placement of data by subsequent WRITE FILE statements. Inclusion of the HOLD keyword means that while a record is being processed, no other user can access the record. The HOLD keyword is useful when changes in existing records are to take place.

When the EXIT or IOERR error clause is specified, program control is transferred to the designated statement when the error condition occurs.

The OPEN statement positions the file at its beginning. An OPEN statement that refers to a file that is already opened is ignored.

Rules:

1. If a file is opened for input only or for output only, no replacement (REWRITE FILE) or deletion (DELETE FILE) is allowed.
2. For an existing stream-oriented file, specifying OPEN with the OUT clause deletes the file contents. Therefore, to add records to an existing file, open the file using the RESET statement with the END clause, not OPEN with OUT.
3. To overwrite an existing record-oriented file, use the OPEN FILE with the OUT REUSE clause.
4. In a multiple OPEN statement, an error causes transfer of program control to the statement designated. Subsequent files specified on the OPEN statement are not processed.

5. Only one error clause, which will apply to all files, may be specified in a multiple OPEN FILE.

6. Naming conventions for files depend upon the environment in which they are used. See Appendix A. "Implementation Considerations."

Example:

```
40  OPEN 'XYZ' IN, ''ABC'' OUT
50  OPEN FILE A$ ALL HOLD, B$||C$ IN
```

OUT REUSE clause

```
10  OPEN FILE 'DAILY' OUT REUSE
20  OPEN FILE A$ ALL REUSE, EXIT 50
```

Error clause

```
10  OPEN 'STOCK' IN, EXIT 99
```

The OPTION Statement

Function:

The OPTION statement allows the VS BASIC user to select specific options that can be applied to a VS BASIC program.

General Format:

OPTION opt_1 [,opt_2] [**REM** *comment*]

where opt_1 and opt_2 are either:

INVP or $\left\{\begin{matrix}\textbf{SPREC}\\ \textbf{LPREC}\end{matrix}\right\}$

INVP specifies an inverted print edit option, SPREC specifies a short precision option, LPREC specifies a long precision option, and *comment* is one or more EBCDIC characters.

Action:

When the inverted print (INVP) keyword is encountered at execution time, it causes the inversion of the comma and period (decimal point) special characters when a [MAT] PRINT [USING] statement is executed. That is, the decimal point replaces the comma (and vice versa) in the printing of numeric data.

Specifying LPREC or SPREC results in long or short precision, respectively, being applied to the arithmetic data in your VS BASIC program. Use of the LPREC or SPREC in the OPTION statement overrides any precision specification in the command used to execute the program or the system-supplied default.

Rules:

1. When used, the OPTION statement must be the first statement in a BASIC program.
2. When the option is specified, it applies throughout the execution of the BASIC program.
3. The options may appear in any order within the OPTION statement. If both LPREC and SPREC are specified, the last one stated is effective.

Example:

```
10 OPTION SPREC, INVP
20 A = 12345.67
30 PRINT Using 40, A
40 FORM PIC (ZZ,ZZZ.##), SKIP
```

Statement 10 causes the short precision and inverted print options to apply to the program. The output of the PRINT USING statement is printed as 12.345,67 in place of the representation, 12,345.67.

The PAUSE Statement

Function:

The PAUSE statement causes program execution to be interrupted and a message to be printed. (Program execution is not interrupted in the batch environment.)

General Format:

PAUSE [[REM] *comment*]

where *comment* is one or more EBCDIC characters.

Action:

When a PAUSE statement is encountered during program execution, execution is interrupted and the following line is printed at the terminal:

PAUSE AT LINE s

where *s* is the number of the PAUSE statement.

When a PAUSE statement is executed immediately after a PRINT statement, the message PAUSE AT LINE *s* is printed on the line below the last line of output from the PRINT statement, even if the final delimiter of that statement is a comma or semicolon.

Rule:

The procedure for subsequent resumption of program execution after a PAUSE statement is to issue a carrier return.

Example:

The following statement would cause the message PAUSE AT LINE 80 to be displayed and processing to be suspended until the user presses the carrier return or equivalent.

```
80 PAUSE
```

The PRINT Statement

The PRINT statement causes the values of specified expressions to be printed at the terminal. It has three forms: PRINT, PRINT with a USING clause referring to an Image statement, and PRINT with a USING clause referring to a FORM statement.

PRINT

Function:

For this form of the PRINT statement, the format of printed values is standardized, but the BASIC user can control the spacing between values on the printed line. For a discussion of how this statement is used, see "Getting Data Out Using the PRINT Statement" in Part I of this publication.

General Format:

[MAT] PRINT [*exp* $_1$] [, | ;] [*exp* $_2$]... [REM *comment*]

where *exp* is a scalar expression, an array name preceded by the keyword MAT, or TAB(*n*), where *n* is a scalar arithmetic expression resulting in a positive integer; the comma and semicolon are delimiters that specify the formatting of the output and *comment* is one or more EBCDIC characters.

As the format description indicates, a comma or semicolon delimiter is optional. An expression in a PRINT statement may be followed by a *null delimiter* of one or more blank characters, or of no characters at all. Null delimiters may be used between two expressions when one, and only one, of them is a character constant. Two consecutive character constants, or a character constant immediately adjacent to a character constant that is part of an expression, or two expressions of which neither is a character constant, must be separated by either a comma or semicolon delimiter.

Action:

When a PRINT statement is executed, each specified expression value is converted to the appropriate standard output format, as described below, and printed at the terminal in the order in which it appears in the PRINT statement. The carriage is then positioned as specified by the delimiter immediately following the expression.

Each array specified in the PRINT statement is printed by rows, the first row of each array beginning at the start of a new line and separated from the preceding line by two blank lines. The remaining rows of each array begin at the start of a new line and are separated from the preceding line by a single blank line.

After the printing of each array member, the carriage is repositioned as specified by the delimiting character that follows the array name. If the final delimiter is a null delimiter, it will be treated as a comma. After the final row has been printed, the carriage is positioned to the start of the next line.

If TAB (*n*) is specified in the output list, the value of *n* indicates the position for the next value or item to be printed. The value of *n* should be between 1 and the defined width of the terminal print line. If the value of *n* is greater than the defined width of the terminal print line, then the modulo of the line width is taken as the line position. If *n* is not specified or the value of *n* is less than 1, the TAB control specification defaults to 1.

Standard Output Formats: If the expression value to be printed is a character

constant, the actual characters contained or referred to in it (including trailing blanks) are printed. For all other character expressions, trailing blanks are not printed.

If the expression value is an arithmetic expression, it is converted for printing to one of the following standard output formats. (The letter P in these descriptions denotes the maximum number of digits—15 and 7, respectively—provided for long-form and short-form arithmetic.)

1. Integer-format, consisting of a sign (blank or minus, and up to P significant decimal digits for integers whose absolute value is less than 1E+P.
2. Floating-point format, consisting of a sign (blank or minus), up to 11 significant decimal digits, a decimal point following the first digit, the letter E, and a signed exponent consisting of two digits. Floating-point format is used to print numbers whose absolute value is less than 1E-1 or greater than or equal to 1E+P. Printed values are rounded off, not truncated.
3. Fixed-point format, consisting of a sign (blank or minus), up to P significant digits, and a decimal point in the appropriate position. Fixed-point format is used to print numbers whose absolute values are not included in the integer and floating-point descriptions above. Printed values are rounded off, not truncated.

The following examples show how various arithmetic values would be printed in response to a PRINT statement in a program run with short-form arithmetic, providing a precision of seven significant digits. The symbol ƀ represents a blank character, which always appears in the sign position of a positive number.

Value Given	Value Printed
123	ƀ123
-1234567	-1.234567E+06
123.4	ƀ123.4
12345.678	ƀ12345.68
999999	ƀ999999
1000000	ƀ1.000000E+06

Spacing of Printed Values: The converted value of each expression specified in the PRINT statement is printed at the terminal in its own print zone. Print zones may be either "full" or "packed," as specified by the delimiter following the expression, and a single printed line may be made up of values in either or both zone types.

For numeric data, the full print zone, which is specified by a comma, is always eighteen characters in length, measured from the first character of the expression value to be printed. For character data, the full print zone is the smallest multiple of eighteen large enough to accommodate the data. Since most printed values are shorter than eighteen characters, a line of full print zones usually produces widely spaced output arranged in columnar fashion.

The packed print zone, which is specified by a semicolon or a null delimiter, varies in length according to the length and data type of the expression that it contains. Packed zones usually produce a denser line of output than full zones.

If the expression to be printed is a character constant, the length of the packed print zone containing it is equal to the length of the character string itself, including all blanks, but not the enclosing single or double quotation marks.

For any other character expression, the length of the packed print zone containing it is equal to the length of the character string, minus any trailing blanks.

If the expression is arithmetic, the length of the packed print zone containing it is determined by the length of the converted value, including sign, digits, decimal point, and exponent, as shown in Figure 5. (Note that positive numbers are preceded by a blank character in the sign position, as described in the standard output formats above.)

Length of Converted Data Item	Length of Packed Print Zone	Example (b represents a blank)
2-4 characters	6 characters	b17.3b
5-7 characters	9 characters	b17.357bb
8-10 characters	12 characters	-45.63927bbb
11-13 characters	15 characters	b1.73579E-23bbb
14-17 characters	18 characters	-892270493115663bb

Figure 5. Packed Print Zone Lengths for Arithmetic Expressions

Positioning of the Carrier: The movements of the carrier at the terminal before, during, and after the printing of expression values depend on both the type of expression being printed and the delimiter following it in the PRINT statement. Figure 6 shows the variety of carrier actions that are possible.

Rules:

1. If all of the expressions in the PRINT statement are array references (that is, array names preceded by the keyword MAT), the MAT can be dropped from each of the array references and specified once before the keyword PRINT.
2. An array cannot be redimensioned in a PRINT statement.
3. If the last character in a PRINT statement is a comma or semicolon, and the PRINT statement is followed by a STOP, CHAIN, or END statement with no intervening PRINT statement, the output is printed before the program terminates.
4. If the value of *n* in the TAB control specification is less than the current line position, then the current line is printed and the line position set to the value of *n* on the next line.
5. The TAB control specification overrides the trailing delimiter (either the comma or the semicolon).
6. The TAB control specification cannot be specified when the MAT option is specified with PRINT USING.

Data Type	Delimiter	Carrier Position For Printing	Carrier Position After Printing
Arithmetic Expression	Comma	If the line contains sufficient space to accommodate the value, printing will begin at the current carrier position. If not, printing will start at the beginning of the next line.	The carrier will be moved past any remaining spaces in the full print zone. If the end of the line is encountered, the carrier will be moved to the beginning of the next line.
	Semicolon	If the line contains sufficient space to accommodate the value, printing will begin at the current carrier position. If not, printing will start at the beginning of the next line.	The carrier will be moved past any remaining spaces in the packed print zone. If the end of the line is encountered, the carrier will be moved to the beginning of the next line.
	Null (Not end of statement)	If the line contains sufficient space to accommodate the value, printing will begin at the current carrier position. If not, printing will start at the beginning of the next line.	The carrier will be left at the print position immediately following the data item.
	Null (End of statement)	If the line contains sufficient space to accommodate the value, printing will begin at the current carrier position. If not, printing will start at the beginning of the next line.	The carrier will be moved to the beginning of the next line.
Character Expression	Comma	If at least 18 spaces remain on the line, printing will start at the current carrier position. If fewer than 18 spaces remain on the line, printing will start at the beginning of the next line. If the end of the line is encountered before the data item is exhausted, printing of the remaining characters will begin on the next line.	The carrier will be moved past any remaining spaces in the full print zone. If the end of the line is encountered, the carrier will be moved to the beginning of the next line.
	Semicolon or Null (Not end of statement)	Printing will start at the current carrier position. If the end of the line is encountered before the data item is exhausted, printing of the remaining characters will begin on the next line.	The carrier will be left at the print position immediately following the packed print zone.
	Null (End of statement)	Printing will start at the current carrier position. If the end of the line is encountered before the data item is exhausted, printing of the remaining characters will begin on the next line.	The carrier will be moved to the beginning of the next line.
Null	Comma	No printing will occur.	The carrier will be moved 18 spaces. If the end of the line is encountered, the carrier will be moved to the beginning of the next line.
	Semicolon	No printing will occur.	The carrier will be moved three spaces. If the end of the line is encountered, the carrier will be moved to the beginning of the next line.
	Null	No printing will occur.	If the null data item is the first item on the list, the carrier will be moved to the beginning of the next line. Otherwise, no movement of the carrier will occur.

Figure 6. Carrier Positions in PRINT Statement

Examples:

Statement	Printed Output
`10  PRINT 'A','B'`	`A --17 blanks-- B`
`20  PRINT 'A';'B'`	`AB`
`30  LET A$="B"`	
`40  PRINT 'A' A$`	`AB`
`50  PRINT A$ 'A',A$;A$`	`BA --17 blanks-- BB`
`60  PRINT A$;'A'`	`BA`
`70  LET A$ = ''`	
`80  PRINT 'A';A$;'A'`	`AA`
`90  PRINT 'A';'';'A'`	`A --18 blanks-- A`

In the following example, assume that there are 18 spaces from the beginning of one print zone to the next and that the line width set at the terminal is 126.

```
20   MAT READ A(15)
30   DATA 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
40   MAT X(2,2) = (1)
50   MAT PRINT A,X
                .
                .
                .
1   2   3   4   5   6   7
8   9   10  11  12  13  14
15
1   1
1   1
```

The following example illustrates the use of the TAB control specification with the PRINT statement.

```
10 W = 6
20 PRINT TAB(7) 'X'; TAB(4*W), Z
```

results in the literal X being printed starting at line position 7, and the value Z being printed starting at line position 24.

PRINT USING Image

Function:

This form of the PRINT statement allows the BASIC user to have program values printed at the terminal in a format of his own choosing. The PRINT statement specifies the values to be printed and the statement number of the Image statement to be used. The Image statement provides explicit format specifications for the values to be printed. In addition, literal data (non-quoted character strings) can be included in the Image statement. For a discussion of how these statements are used, see "PRINT Using Image and FORM" in Part I of this publication.

General Format:

PRINT Statement with USING Image Clause:

[MAT] PRINT USING *s* [,*output-list*] [;] [REM *comment*]

where *s* is the line number of the Image statement to be used, *output-list* is defined under "Syntax Definition," and *comment* is one or more EBCDIC characters. (**Note:** Consecutive scalar expressions within the output-list of a PRINT statement with the USING Image clause can be separated by either a comma or a semicolon.)

Image Statement:

$$: \left\{ \begin{array}{l} c_1 \\ f_1 \\ c_1\ f_1[c_2\ f_2]...[c_n] \\ f_1\ c_1[f_2\ c_2]...[f_n] \end{array} \right\}$$

The symbol for an Image statement is a colon (:) following the statement number. The term *c* represents a string, without enclosing quotes, of any EBCDIC characters other than the pound sign (#), and the term *f* represents one or more format specifications, as described below.

Action:

When a PRINT statement with a USING Image clause is executed, the output list items are evaluated, and their values are edited, in order of appearance in the PRINT statement, into the corresponding format specifications in the Image statement. The EBCDIC characters represented by *c* in the general format description are printed exactly as entered in the Image statement itself.

Each array specified in a MAT PRINT statement with a USING Image clause is printed by rows at the terminal according to the format defined by the associated Image statement.

When printed, the first row of each array begins at the start of a new line and is separated from the preceding line by two blank lines. Each succeeding array row begins at the start of a new line and is separated from the preceding row by one blank line. After the last row has been printed, the carriage is repositioned to the beginning of the next line.

As stated above, consecutive scalar expressions in the output-list of the PRINT statement with USING Image clause can be separated by either a comma or a semicolon delimiter. The delimiter used determines how output will be printed when the number of scalar expressions in the PRINT statement exceeds the number of format specifications in the corresponding Image statement. If the expression using the last format specification is followed by a comma delimiter, the current output is printed, the carrier is positioned to the beginning of the next line, and the remaining expressions are formatted according to the format specifications from the beginning of the corresponding Image statement. If the expression using the last format specification is followed by a semicolon delimiter, the current output is not printed, and output from the remaining expressions (formatted according to the format specifications from the beginning of the corresponding Image statement) is added to the current output print line.

If the number of members in a row of an array, specified by a MAT PRINT statement with a USING clause, exceeds the number of format specifications in the corresponding Image statement, output is printed, the carrier is positioned to the beginning of the next line, and the remaining row members are formatted according to the format specifications from the beginning of the corresponding Image statement.

If either the number of scalar expressions or the number of row members is less than the number of format specifications in the corresponding Image statement, the output of the current print line ends at the first unused format specification.

Format Specifications: For each occurrence of the pound sign (#) in an Image statement, a single space is reserved in the print line for a character in the corresponding data reference of the associated PRINT statement. The pound sign may represent either character or arithmetic data. For arithmetic data,

decimal points and the plus and minus signs, like the characters represented by *c* in the general format description, are printed as entered, provided the values are appropriate to the specified signs. (See "Conversion of Data Reference Values" below for a discussion of the printing of signs in the Image statement.) For character data, the character string will override any format descriptors.

1. The various format specifications are:
   a. *Character-format*—one or more # characters.

      General Format: #[#] ...

   b. *Integer-format*—an optional sign followed by one or more # characters.

      General Format: [+ | -]#[#]...

   c. *Fixed-Decimal format*—an optional sign followed by either:

      (1) no #'s, a decimal point, one or more # characters

      (2) one or more #'s, a decimal point, no #'s

      (3) one or more #'s, a decimal point, one or more # characters

      General Format: [+ | -] [#]... .#[#]... | #[#]... .[#]...

   d. Floating-Point format— either the integer or fixed-point format (given above) followed by four | characters

   General Format: Integer-format Fixed-decimal format | | | |

2. The following rules define the start of a format specification:
   a. A # character is encountered and the preceding character is not a # character, decimal point, plus sign, or minus sign.
   b. A plus sign or a minus sign is encountered, which is followed by:

      (1) A # character or

      (2) A decimal point that is followed by a # character.

   c. A decimal point is encountered, which is followed by a # character and:

      (1) The preceding character is not a # character, plus sign, or minus sign, or

      (2) The preceding character string is a fixed-decimal format specification.

3. The following rules define the end of a format specification that has been started:
   a. A # character is encountered and:

      (1) The following character is not a # character, or

      (2) The following character is not a decimal point, or

      (3) The following character is a decimal point and a decimal point has already been encountered, or

      (4) The following four consecutive characters are not | characters.

   b. A decimal point is encountered and:

(1) The following character is not a # character, or

(2) The following character is another decimal point, or

(3) The following four consecutive characters are not | characters.

c. Four consecutive | characters are encountered.

Conversion of Data Reference Values: When the data referred to is a character value, the characters contained in it are edited into the line replacing characters in the format specification including sign, pound sign, decimal point, and | | | |.

If an edited character value is shorter than its format specification, blank padding occurs on the right. If an edited character value is longer than its format specification, it is truncated on the right. A character constant containing no characters (null) causes blank padding of the entire format specification.

An arithmetic expression is converted in accordance with its format specification as follows:

1. If the format specification contains a plus sign, and the expression value is positive, a plus sign is edited into the line.
2. If the format specification contains a plus sign, and the expression value is negative, a minus sign is entered into the line.
3. If the format specification contains a minus sign, and the expression value is positive, a blank is edited into the line.
4. If the format specification contains a minus sign, and the expression value is negative, a minus sign is edited into the line.
5. If the format specification does not contain a sign, and the expression value is negative, the negative number will be printed with a minus sign provided that the format specification is long enough to contain both the number and the sign. If the format specification is not long enough, a minus sign and asterisks are edited into the line instead of the negative expression value.
6. The expression value is converted according to the type of its format specification as follows:

Integer-format: The value of the expression is converted to an integer, rounding any fraction.

Fixed-Point-format: The value of the expression is converted to a fixed-point number, rounding the value or extending it with zeros in accordance with the format specification.

Floating-Point format: The value of the expression is converted to a floating-point number, either rounding the value or extending it with zeros and adjusting the exponent in accordance with the format specification.

If the length of the integer portion of the arithmetic expression value is less than or equal to the length of the integer portion of the format specification, the expression value is edited, right-justified, into the line. If the length of the integer portion of the format specification is less than the length of the integer portion of the expression value, asterisks are edited into the line instead of the expression value.

Some examples of reference values and the way they are printed under various format specifications are as follows:

Format Specification	Reference Value	Printed Form
###	123	123
###	12	ß12
###	1.23	ßß1
+###	123	+123
+###	-123	-123
-###	123	ß123
-###	-123	-123
-###	-	****
+###	1234	****
##.##	123	*****
##.##	1.23	ß1.23
##.##	1.23456	ß1.23
##.##	.123	ß0.12
##.##	12.345	12.35
###||||	123	123E+00
###||||	12.3	123E-01
###||||	.1234	123E-03
##.##||||	123	12.30E+01
##.##||||	1.23	12.30E-01
##.##||||	.1234	12.34E-02
##.##||||	1234	12.34E+02

Rules:

1. The maximum number of Image statements permitted in a single BASIC program is dependent on the number of FORM statements also present. Together, they may not exceed 50.
2. If the PRINT statement output list contains at least one item, there must be at least one format specification in the corresponding Image statement.
3. If all of the items in the output list of the PRINT statement are array references (that is, array names preceded by the keyword MAT), the MAT can be dropped from each of the array references and specified once before the keyword PRINT.
4. Image statements are nonexecutable and may be placed anywhere in a BASIC program, either before or after the PRINT statements that refer to them. However, they may not appear within a multiline function definition (that is, between a DEF statement and its associated FNEND statement).
5. An array cannot be redimensioned in a PRINT USING statement.
6. A comma ending an output data list on a PRINT statement with a USING Image clause will be ignored, and treated as a null.

Examples:

```
30  PRINT USING 40, X,Y
40  :RATE OF LOSS #### EQUALS ####.## POUNDS
```

Printed Output:

```
RATE OF LOSS   342      EQUALS    42.02   POUNDS
               Value              Value
               of X               of Y
```

For the following example:

```
10  DIM A(4,3)
20  : ### ##.## ##.##!!!!
30  MAT A = (1)
40  PRINT USING 20, MAT A
```

the output would appear as follows:

```
1    1.00 10.00E-01
1    1.00 10.00E-01
1    1.00 10.00E-01
1    1.00 10.00E-01
```

Use of the semicolon delimiter within the output list of a PRINT statement with USING Image clause:

```
10    A=123
20    B=234
30    C=345
40    D=456
50    PRINT USING 70, A,B,C,D
60    PRINT
70    :  ###  ###
80    PRINT USING 70, A,B;C,D
```

results in the following output:

```
123   234
345   456

123   234    345    456
```

For the same values for A,B,C,D as above, the following statements:

```
100   PRINT A,B;
110   PRINT USING 120,C,D
120   :  ###  ###
```

results in:

```
123   234    345    456
```

Use of the optional semicolon at the end of a PRINT statement with USING Image clause:

```
140   PRINT USING 150;
150   : TWO LINES
160   PRINT USING 170
170   : ARE JOINED
```

results in:

```
TWO LINES ARE JOINED
```

PRINT USING With FORM Clause

Function:

This form of the PRINT statement allows the BASIC user to have values printed at the terminal in a format of his own choosing, with the additional capabilities of inserting edit characters in numeric data items, and controlling line skipping, blank field insertion, and line positioning. The PRINT statement specifies the values to be printed and the statement number of the FORM statement to be used. The FORM statement contains data form specifications and control specifications that govern the printing operation. For a discussion of how these statements are used, see "PRINT Using Image and FORM" in Part I of this publication.

General Format:

PRINT Statement:

[MAT] PRINT USING *s*[,*output-list*] [**REM** *comment*]

FORM Statement:

FORM $\left\{ \begin{matrix} \text{"}lit_1\text{"} \\ c_1 \\ [n_1*]d_1 \end{matrix} \right\} \left[\begin{matrix} \text{"}lit_2\text{"} \\ ,c_2 \\ ,[n_2*]d \end{matrix} \right]$ [**REM** *comment*]

where *s* is the number of the FORM statement to be used, and *comment* is one or more EBCDIC characters. In the FORM statement, *"lit"* is literal specified by a quoted character constant, *c* is a control specification as defined below, *d* is a data form specification as defined below, and *n* is an unsigned, nonzero, integer constant, or an arithmetic variable whose value is greater than zero, indicating the number of consecutive times that the associated data form specification is to be interpreted before the next data form or control specification is examined.

A control specification can be any of the following:

X[*n*]
POS[*n*]
SKIP[*n*]

where *n* is an unsigned, integer constant or an arithmetic variable whose value must be greater than zero.

A data form specification can be either of the following:

C[*w*]

PIC($\left\{ \begin{matrix} ds_1 \\ ic_1 \end{matrix} \right\} \left[\begin{matrix} ds_2 \\ ic_2 \end{matrix} \right]$ ... [| | | |][*r*])

where *w* is an unsigned, nonzero, integer constant, *ds* is one of the digit specifiers #, Z, *, $, +, or -, *ic* is one of these insertion characters:

,	comma
/	slash
B	blank
.	decimal point

and *tr* is one of these trailing characters:

+	trailing plus
-	trailing minus
$	trailing dollar sign
CR	trailing credit sign
DB	trailing debit sign

Action:

When a PRINT statement with a USING clause referring to a FORM statement is executed, the output list items are evaluated, and their values are displayed, in order of appearance in the PRINT statement, according to the data form specifications and the control specifications in the FORM statement. Each item in the PRINT statement is converted, if necessary, to the form and length of the corresponding data form specification in the FORM statement, and placed in the position specified by the control specification. Array references in the PRINT statement are converted row by row. The literal, enclosed in quotation marks, is printed exactly as entered in the FORM statement.

If there are more items in the PRINT statement than data form specifications in the FORM statement, the FORM statement is reused from its beginning until all of the items in the PRINT statement are exhausted. When the PRINT list is exhausted, any control specifications immediately following the last data form specification used are interpreted and executed.

Control Specifications: Control specifications may be interspersed with data form specifications in the FORM statement to control line skipping, insertion of blank fields, and line position of the next value to be displayed.

If POS *n* is specified, the line position for the next displayed value is set to position *n*. The value *n* must be between 1 and the defined line width of the terminal. If an arithmetic variable is used to specify *n*, the truncated integer portion of the value is used to determine line position. A value less than 1 causes the control specification to have no effect. A value greater than the defined line width causes the current line to be displayed and the line position to be reset to position 1 of the next line. If POS alone is specified (without a position number), it is assumed that *n* has a value of 1.

If X*n* is specified, *n* positions are spaced over. If an arithmetic variable is used to specify *n*, the truncated integer portion of the value is used to determine the number of spaces. A value less than 1 causes the control specification to be ignored. If X alone is specified (without a number), it is assumed that *n* has a value of 1.

If SKIP *n* is specified, the current line is displayed and *n* line advances occur. If an arithmetic variable is used to specify *n*, the truncated integer portion of the value is used to determine the number of line advances. The value of *n* must be between 1 and 255; if the control specification is greater than 255, it is set back to 255. Regardless of the number of line advances that occur, the line position is always at the initial position of a line. If SKIP alone is specified (without a number), it is assumed that *n* has a value of 1.

Output Line Display: An output line is displayed only when one of the following occurs: a SKIP control specification is encountered in the FORM statement or the position of the next value to be included in the display line is beyond the line length of the terminal. After an output line is displayed, the line position is reset to the initial position. If the defined line length of the

terminal is insufficient to display a value, the portion that fits is included in the display line, the line is displayed, and the remaining portion of the value begins at the initial position of the next line. In general, it is good practice to include a SKIP control specification at the end of a FORM statement.

Data Form Specifications: Data form specifications control the exact form in which each item in the PRINT statement is to be displayed. A single data form specification can be interpreted *n* consecutive times by prefixing it with an *n** replication factor. If an arithmetic variable is used to specify *n*, the truncated integer portion of the value is used to determine the number of consecutive interpretations of the data form specification.

The specification C[*w*] indicates that the item to be displayed is a character value. The integer constant *w* specifies the length of the display field that is to contain the character value. If the actual character string is shorter than the display field, the value displayed is padded on the right with blanks. If the character string is longer than the display field, the value displayed is truncated on the right to the length of the field. If *w* is not specified, the length of the display field is equal to the length of the character string.

The PIC specification is a string of digit specifiers and/or insertion characters, optionally followed by an exponent specifier. It can refer to arithmetic or character data. When used with arithmetic data, the field is edited as indicated by the PIC specification. When used with character data, the total number of positions in the PIC specification indicates the length of the character data. The PIC specification can be used for output operations only.

With respect to the printing of arithmetic data, the following rules apply:

1. Digit specifiers can be conditional or unconditional. They are:

   \# This character is placed in any position that must always contain a numeric digit.

   Z This character causes a leading zero in the associated position to be replaced by a blank.

   * This character causes a leading zero in the associated position to be replaced by an asterisk.

   $ This character is placed in each position that can potentially be occupied by a floating dollar sign—that is, a dollar sign to the immediate left of the first significant digit. Nonsignificant zeros are suppressed.

   \+ This character is placed in each position that can potentially contain a floating high-order sign. The appearance of either a plus sign or a minus sign is guaranteed in the printed field. Nonsignificant zeros are suppressed.

   \- This character is placed in each position that can potentially contain a floating high-order minus sign if the value in the field is negative. Nonsignificant zeros are suppressed.

2. Insertion characters are conditional or unconditional. They are:

   B This unconditional character always causes a blank to be inserted in the corresponding position of the display field.

   , This character is inserted in the corresponding position of the display field unless zero suppression is in effect and no significant digits appear

to the left of the comma in the display field. In this case, the comma will be replaced by a floating or zero suppression character.

/ This character is inserted in the corresponding position of the display field unless zero suppression is in effect and no significant digits appear to the left of the slash in the display field. In this case, the slash will be replaced by a floating or zero suppression character.

. This character is inserted in the corresponding position of the display field unless zero suppression has been specified for every digit position and the value to be displayed is zero. In this case, the decimal point will be replaced by a floating or zero suppression character.

Note: When the INVP option is specified, the commas and decimal points will be converted during program execution.

3. The exponent specifier | | | | causes the following sequence of chacters to be placed in the corresponding positions of the display field:

   a. The letter E

   b. The exponent sign (plus or minus)

   c. Two digits representing the value of the exponent.

These characters do not appear in the display field when zero suppression is in effect and the value to be displayed equals zero.

4. Trailing characters are conditional. They are:

   trailing +
   This character causes a plus sign or a minus sign to be inserted in the corresponding position of the display field unless zero suppression is in effect and no significant digits appear to the left of the sign in the display field. In this case, the sign will be replaced by an asterisk or a blank.

   trailing -
   This character is inserted in the corresponding position of the display field if the value to be displayed is negative, unless zero suppression is in effect and no significant digits appear to the left of the minus sign in the display field. In this case, the minus sign will be replaced by an asterisk or a blank.

   trailing $
   This character is inserted in the corresponding position of the display field unless zero suppression is in effect and no significant digits appear to the left of the dollar sign in the display field. In this case, the dollar sign will be replaced by an asterisk or a blank.

   trailing credit CR
   These alphabetic characters are inserted in the corresponding position of the display field if the value to be displayed is negative. The two alphabetic characters are replaced by two blanks if the value is positive.

   trailing debit DB
   These alphabetic characters are inserted in the corresponding position of the display field if the value to be displayed is negative. If the value is positive, the two alphabetic characters are replaced by two blanks.

Rules:

1. The maximum number of FORM statements permitted in a single BASIC program is dependent on the number of Image statements also present. Together, they may not exceed 50.
2. There must be at least one data form specification in the FORM statement if there is an output list.
3. FORM statements are nonexecutable and may be placed anywhere in a BASIC program, either before or after the PRINT statements that refer to them. However, they may not appear within a multiline function definition (that is, between a DEF statement and its associated FNEND statement).
4. An array cannot be redimensioned in a PRINT USING statement.
5. A PIC specification in a FORM statement cannot contain both the Z and * digit specifiers.
6. A PIC string must be from 1 to 32 characters long.
7. A single $, +, or - as the leftmost character in a PIC string is treated as a static character. Two or more $, +, or - signs at the leftmost end of a PIC string are treated as floating characters. The same character cannot appear as both a static character and part of a floating character string in a single PIC string.
8. A string of floating characters must contain at least one more floating character than the maximum number of digits expected in the output field.
9. A PIC string cannot end with a B, slash (/), or comma (,) insertion character.9
10. A PIC string cannot begin with a slash (/) or comma (,) insertion character.
11. There cannot be more than one decimal point (.) insertion character in a PIC string.
12. A PIC string must include at least one #, Z, *, or floating character string.
13. No # digit specifiers can appear to the left of a zero suppression character or a floating character.
14. A # digit specifier cannot appear in a PIC string that contains a decimal point followed by zero suppression or floating characters.
15. The symbols + and - cannot appear in the same PIC string.
16. The trailing signs CR, DB, +, and - cannot appear in the same PIC string.
17. A trailing character cannot appear in a PIC string in which that trailing character is used as either a non-floating character or as part of a floating character string.
18. In order to edit signed numbers properly, a sign must appear in the picture clause, either as a floating character or a trailing character.
19. Blanks replace any portion of the PIC specification that is not used as a result of a character data assignment.

20. If the length of the character string exceeds the length of the PIC specification, the character string is truncated to the length of the PIC specification.

21. A comma or semicolon at the end of an output list on a PRINT USING statement that refers to a FORM statement will be ignored, and treated as a null.

Examples:

```
50  PRINT USING 55, "NAME", "LOCATION", "HEIGHT", "RANK"
55  FORM POS6, C, POS31, C, POS46, C, POS60, C, SKIP2
```

The character constants NAME, LOCATION, HEIGHT, and RANK are to be printed beginning in print positions 6, 31, 46, and 60, respectively. After the line is printed, the carrier skips two lines.

```
60  PRINT USING 65, N$, L$, S, R
65  FORM X5, C,POS31, C,POS46, PIC(ZZZ,###)
    POS62, PIC(Z#), SKIP1
```

Two character variables and two numeric variables are to be printed. The first character variable, N$, is printed after five print positions are skipped; the second, L$, is printed beginning in print position 31. The first numeric variable, S, is printed beginning in print position 46 according to the PIC specification, and the second variable, R, is printed beginning in print position 62, also according to a PIC specification. After the line is printed, the carrier skips to the next line.

Assuming that N$, L$, S, and R contained appropriate values, these two examples could be combined to produce this output:

```
NAME            LOCATION        HEIGHT     RANK
EVEREST         NEPAL-CHINA     29,141     1
```

The PRINT TO Statement

Function:

The PRINT TO statement directs output normally printed at the terminal to an entry-sequenced record-oriented file or to the terminal.

General Format:

PRINT TO *filename* [REM *comment*]

where *filename* is a character expression, and *comment* is one or more EBCDIC characters.

Action:

If the filename on the PRINT TO statement has not been previously opened by an OPEN FILE statement, the PRINT TO statement opens and activates this file for output. The subsequent [MAT]PRINT[USING] statements cause output to be placed on that file. The file produced contains print line images.

If the value of the character expression specified for the filename consists of one or more blanks, the output from subsequent [MAT]PRINT[USING] statements is printed at the user's terminal.

Any errors that occur on the PRINT TO or subsequent [MAT]PRINT[USING] statements can be handled by the general ON ERR statement (See the ON statement discussion).

If space in the output file is exhausted before all print line images in a [MAT]PRINT[USING] statement are placed in the file, program execution is terminated unless a general ON ERR statement is active.

Rules:

1. The PRINT TO statement, which includes the filename of the data file, causes subsequent [MAT]PRINT[USING] statements to add print line images to an entry-sequenced record-oriented file. The print line added is identical to the print line as it would have been printed at the terminal. The record length of the file must be large enough to accommodate the total number of characters in the widest line.
2. A PRINT TO statement can only be operational on one file at a time. That is, only one data file can receive data from a subsequent PRINT statement at one time.
3. A PRINT TO statement referring to a filename of a data file that is already open for output and is currently active as the file print is directed to, is ignored.
4. An input file must be explicitly closed before it can be referred to by a PRINT TO statement.
5. CLOSE FILE, RESET FILE, and PRINT are the only BASIC statement that can access a record-oriented file opened for accessing by PRINT TO statements. Conversely, any record-oriented file opened by a WRITE FILE/REWRITE FILE statement cannot be accessed by a [MAT]PRINT[USING] statement.
6. A data file opened by a PRINT TO statement is positioned at its end and subsequent [MAT]PRINT[USING] statements append records to the end of the file. To write over data existing on a file, the REUSE clause with the OPEN statement must be used.

7. In a file where the first item in the record is line numbers, the line number is picked up as part of the print line.

Note: Files created using PRINT TO can be printed using Access Method Services. Appendix A contains a summary of the VS BASIC requirements for using these services to print such files.

Example:

```
 10      PRINT    'ENTER FILE NAME FOR PRINTING OF REPORT'
 20      INPUT A$
 30      OPEN FILE A$ OUT
          .
          .
          .
 90      PRINT TO A$
100      PRINT    'RESULTS OF TEST CASES 120-400'
          .
          .
          .
200      PRINT TO '   '
210      PRINT    'YOUR REPORT PRINTED ONTO FILE   ' ;A$
```

results in the output at line 10 printing at the terminal, the report header at line 100, and all subsequent [MAT]PRINT[USING] statement writing onto the data file designated by A$ up to the execution of the statement at line 200. The output at line 210 is printed at the terminal.

The PUT Statement

Function:

The PUT statement causes data values to be placed in a specified stream-oriented file. For a discussion of how this statement is used, see "Stream-Oriented Files" in Part I of this publication.

General Format:

[MAT] PUT *filename, output-list* [,EXIT *es*] [[,EOF *s*][,IOERR *s*]] [REM *comment*]

where *filename* is a character expression, *es* is the number of an EXIT statement, *s* is the number of any executable statement and *comment* is one or more EBCDIC characters.

Action:

If it has not been otherwise specified in an OPEN statement or a RESET statement with an END clause, a file is activated for output by the first execution of a PUT statement specifying its file name. The file is positioned at its beginning and values are placed in it from the output list items in the PUT statement. Values of array members are written row by row. Subsequent PUT statements for the same file cause values to be placed in it beginning at the current file position.

If the EOF clause is included in the PUT statement, control is transferred to the specified statement if space in the output file is exhausted before all items in the PUT statement have been placed in it.

If the IOERR clause is included, control is transferred to the specified statement if a device malfunction prevents the placing of an item in the output file.

If the EXIT clause is included, the specified EXIT statement is examined for the appropriate error condition if either of the above problems occurs, and control passes to that statement.

A file is deactivated in response to a CLOSE statement or at the end of program execution.

Rules:

1. A file currently activated as an input file may not be specified in a PUT statement. It must first be closed, either by a CLOSE statement or by a RESET statement with an END clause.
2. If space in the output file is exhausted before all items in the PUT statement are placed in the file, program execution is terminated unless an EOF clause, or an EXIT clause pointing to an EXIT statement with an EOF clause, is specified.
3. The EOF and IOERR clauses can be specified in any order. The presence of either of these clauses in the PUT statement precludes the specification of an EXIT clause.
4. In the DOS/VS environment, the EOF clause cannot be specified.
5. If all of the items in the output list of the PUT statement are array references (that is, array names preceded by the keyword MAT), the MAT

can be dropped from each of the array references and specified once before the word PUT.

6. An array cannot be redimensioned in a PUT statement.

7. Specifying PUT for a file that has not been previously opened positions the file to its beginning, and causes the existing data in the file to be deleted and the new data to be entered.

8. To add records to an existing file, open the file using the RESET statement with the END clause before executing a PUT statement.

9. Naming conventions for files depend upon the environment in which they are used. See Appendix A. "Implementation Considerations."

Examples:

```
30 PUT 'ABF',Z3,5*X-7,MAT A,D$,9.005
60 MAT PUT 'BCD', A,M,Q,B$
```

The READ Statement

Function:

The READ statement assigns values to variables and arrays from the data table created by DATA statements. For a discussion of how these statements are used, see "Getting Data Into the Computer" in Part I of this publication.

General Format:

[MAT] READ *input-list* [REM *comment*]

where *comment* is one or more EBCDIC characters.

Action:

When a READ statement is executed, successive values from the data table are assigned to the items in the input list, beginning at the current position of the data table pointer. Values are assigned to arrays row by row.

Subscripted references to array members in the input list are evaluated as they occur; thus, an assigned variable in a READ statement may be used subsequently as the subscript of another variable in the same statement.

If a redimension specification follows an array name in the input list, the truncated integer portion of each expression value is used to redimension the array before data values are entered.

If a character constant in the data table is shorter than the variable in the input list, its value is padded on the right with blanks to the length of the variable before being assigned. If a character constant is longer than the corresponding variable, its value is truncated on the right to the length of the variable before being assigned. A character constant containing no characters (null) is assigned as all blank characters.

Rules:

1. Each data value read from the table must be of the same type (character or arithmetic) as the variable to which it is assigned.
2. A READ statement is invalid if there are no DATA statements in the program.
3. If the data table is exhausted and unassigned variables, arrays, or array members still remain in the READ statement input list, an error condition results.
4. If all of the items in the input list are array references (that is, array names preceded by the keyword MAT), the MAT can be dropped from each of the array references and specified once before the keyword READ.

Examples:

```
05      DIM A$8,B$8,Z(5,3)
10      DATA 'JONES', 15.00, 'SMITH', 20.50
20      READ A$, A1, B$, B1
30      DATA 1,2,3,4,5,6
40      READ A,B,C,X(A),X(B),X(C)
50      DATA 5*1.0, 5*2.0
60      MAT READ Z(10)
```

After execution of the above statements, the character variables A$ and B$ will contain the character strings JONES and SMITH, respectively, each padded on the right with blanks to a length of eight. The arithmetic variables A1 and B1 will contain the numeric values 15.00 and 20.50, respectively. The

arithmetic variables A, B, and C will contain the integer values 1, 2, and 3, respectively, and the first three members of the one-dimensional array named X will contain the integer values 4, 5, and 6, respectively. The two-dimensional array Z, with 5 rows and 3 columns, will be redimensioned as a one-dimensional array with 10 members. The first five members will be assigned the value 1.0; the remaining five members will receive the value 2.0.

The READ FILE Statement

Function:

The READ FILE statement causes values to be assigned to variables from a specified record-oriented file. For a discussion of how this statement is used, see "An Entry-Sequenced File," "A Key-Sequenced File," and "A Relative Record File" in Part I of this publication.

General Format—Entry-Sequenced Files:

[MAT] READ FILE [USING *sn*] *filename, input-list*

[,EXIT *es*
[,EOF *s*][,IOERR *s*][,CONV *s*][REM *comment*]]

General Format - Key-Sequenced Files:

[MAT] READ FILE [USING *sn*] *filename* [,*KEY r exp*] ,*input-list*

[,EXIT *es*
[,EOF *s*][,IOERR *s*][,CONV *s*][,NOKEY *s*][REM *comment*]]

General Format - Relative Record Files:

[MAT] READ FILE [USING *sn*] *filename* [,REC = *e*], *input-list*

[,EXIT *es*
[,EOF *s*][,IOERR *s*][,CONV *s*][,NOREC *s*][REM *comment*]]

where *sn* is the number of the FORM statement; *filename* is a character expression; *r* is the relational operator = or .EQ., ≥, >=, or .GE.,; *exp* is a character expression; *e* is an arithmetic expression; *es* is the number of an EXIT statement; *s* is the number of an executable statement; and *comment* is one or more EBCDIC characters.

Action:

Execution of the READ FILE statement causes a record to be placed in a buffer associated with the file specified. From this buffer, values are assigned to the items in the input list. Values are assigned to arrays row by row. When all of the items in the input list have been assigned a value, execution of the program resumes.

If the USING clause is present, values are converted as indicated in the specified FORM statement. Otherwise, values are assigned from the record without conversion; thus, numeric data must exist in the record in internal format and in the correct precision.

For key-sequence files, if the KEY clause is included, the first record satisfying the specified condition is retrieved. The character expression is evaluated and truncated on the right, if longer than the record key, before being compared with the actual record key. A character string shorter than

the record key is compared with the first *n* characters of the key, where *n* is the number of characters in the character string.

If the KEY clause is not present, the next sequential record is retrieved from the file. If the NOKEY clause is included, and no record with a key satisfying the KEY clause condition exists in the file, control is transferred to the specified statement.

For relative record files, if the REC clause is included, the record is retrieved by the relative record number specified. The arithmetic expression is the integer part of a positive number which gives the relative record number of a particular record.

If the REC clause is not present, the READ FILE statement retrieves the next non-null record following the last record successfully referred to.

If the NOREC clause is included, and the relative record specified does not exist in the record-oriented file, control is transferred to the statement specified.

The internal variable, &REC, will contain the number of the last relative record successfully referred to.

For all files, if the EOF clause is included, control is transferred to the specified statement if the KEY or the REC clause is not present and the last record in the file has been read.

If the IOERR clause is included, control is transferred to the specified statement if a device malfunction prevents reading of the record.

If the CONV clause is included, control is transferred to the specified statement if an item of data in the record cannot be converted to the type of the corresponding variable in the input list, or if the end of the record is encountered before all of the variables in the input list are filled.

If the EXIT clause is included, the specified EXIT statement is examined for the appropriate error condition if any of the above problems occur.

Subscripted references to array members in the input list are evaluated as they occur, from left to right. Thus, an assigned variable in a READ FILE statement may be used subsequently as the subscript of another variable in the same READ FILE statement.

If a redimension specification follows an array name in a READ FILE statement, the truncated integer portion of each expression value is used to redimension the array before data values are assigned from the record.

Rules:

1. Each value assigned in a READ FILE statement should be of the same type (arithmetic or character) as the corresponding item in the input list.
2. If the input record is exhausted before all items in the input list are filled, program execution is terminated unless a CONV clause, or an EXIT clause pointing to an EXIT statement with a CONV clause, is specified.
3. The EOF, IOERR, CONV, NOKEY or NOREC, clauses can be specified in any order. The presence of any of these clauses in the READ FILE statement precludes the specification of an EXIT clause.
4. If all of the items in the input list are array references (that is, array names preceded by the keyword MAT), the MAT can be dropped from each of the array references and specified once before the keywords READ FILE.

5. Naming conventions for files depend upon the environment in which they are used. See Appendix A. "Implementation Considerations."

Examples:

```
 50  READ FILE 'ABC', KEY='05', X,Y,MAT Z, EXIT 200
 70  READ FILE 'FILE6', A,B,C$,D$
 90  MAT READ FILE USING 100 'FILE3',X,Y$,EOF 210,
     CONV 250
100  FORM NC5.1,X5,C10
```

The REM Statement

Function:

The REM statement allows the BASIC user to insert comments in the program listing. For a discussion of how this statement is used, see "Comments in Your Program" in Part I of this publication.

General Format:

REM [*comment*]

where *comment* is one or more EBCDIC characters.

Action:

The REM statement is nonexecutable. It appears in the program listing, but has no effect on program execution.

Rule:

A REM statement may appear anywhere in a BASIC program.

Example:

```
10   REM THIS PROGRAM DETERMINES THE COST PER UNIT
```

REM As A Keyword

The REM keyword allows the BASIC user to insert comments on all VS BASIC statements except the DATA and Image statements. See "Comments in Your Program" in Part I of this publication for further discussion.

Action:

The REM keyword, like the REM statement, is nonexecutable. It appears in the BASIC statement to indicate that the data that follows it is a comment. It has no effect on program execution.

Rule:

A comment cannot appear on the DATA or Image statement.

The REREAD FILE Statement

Function:

The REREAD FILE statement causes the last record read from a record-oriented file to be reaccessed and the values in it reassigned to variables. For a discussion of how this statement is used, see "Rereading Records" in Part I of this publication.

General Format:

[MAT] REREAD FILE [USING *sn*] *filename, input-list*

$$\left[\begin{array}{l}\text{,EXIT } es \\ \text{,CONV } s\end{array}\right] \text{ [REM } comments\text{]}$$

where *sn* is the number of a FORM statement, *filename* is a character expression, *es* is the number of an EXIT statement, *s* is the number of any executable statement, and *comment* is one or more EBCDIC characters.

Action:

Execution of the REREAD FILE statement causes the record that was last read from the specified file to be accessed from its buffer. Values are assigned to the items in the input list. Values are assigned to arrays row by row. When all of the items in the input list have been assigned a value, execution of the program resumes. Note that no actual I/O transmission occurs from the file to the buffer.

If the USING clause is present, values are converted as indicated in the specified FORM statement. Otherwise, values are assigned from the record without conversion; thus, numeric data must exist in the record in internal format and in the correct precision.

If the CONV clause is included, control is transferred to the specified statement if an item of data in the record cannot be converted to the type of the corresponding variable in the input list, or if the record does not contain sufficient values to fill all of the variables in the input list.

If the EXIT clause is included, the specified EXIT statement is examined for the CONV error condition if the above problem occurs.

Subscripted references to array members in the input list are evaluated as they occur, from left to right. Thus, an assigned variable in a REREAD FILE statement may be used subsequently as the subscript of another variable in the same REREAD FILE statement.

If a redimension specification follows an array name in a REREAD FILE statement, the truncated integer portion of each expression value is used to redimension the array before data values are assigned from the record.

Rules:

1. Each value assigned in a REREAD FILE statement should be of the same type (arithmetic or character) as the corresponding variable in the input list.
2. The CONV and EXIT clauses cannot both be specified in the REREAD FILE statement.
3. The last I/O operation on the file specified in a REREAD FILE statement must have been the result of a READ FILE or REREAD FILE statement.

4. If all of the items in the input list are array references (that is, array names preceded by the keyword MAT), the MAT can be dropped from each of the array references and specified once before the keywords REREAD FILE.

5. Naming conventions for files depend upon the environment in which they are used. See Appendix A. "Implementation Considerations."

Examples:

```
 40   REREAD FILE 'ABC', A, B, MAT C, CONV 300
 70   MAT REREAD FILE USING 100 'FILE2',Q,A$,EXIT 280
100   FORM NC6,POS10,C20
```

The RESET Statement

Function:

The RESET statement causes an input or output file to be repositioned. A stream-oriented file can be positioned at its beginning or its end. A record-oriented file can be positioned at its beginning or at a record whose key or relative-record number satisfies a specified condition. For a discussion of how this statement is used, see "Repositioning Files" in Part I of this publication.

General Format—Stream-Oriented Files:

RESET *filename*$_1$ [**END**] [,*filename*$_2$ ...] [,**EXIT** *es* | ,**IOERR** *s*] [**REM** *comment*]

General Format - Entry-Sequenced Files:

RESET FILE *filename*$_1$ [,*filename*$_2$ ...] [,**EXIT** *es* | ,**IOERR** *s*] [**REM** *comment*]

General Format - Key-Sequenced Files:

RESET FILE *filename*$_1$ [,**KEY** *r exp*] [,*filename*$_2$ ...] [,**EXIT** *es* | ,**IOERR** *s*] [,**NOKEY** *s*]
[**REM** *comment*]

General Format - Relative Record Files:

RESET FILE *filename*$_1$ [,**REC** = *e*] [,*filename*$_2$ ...] [,**EXIT** *es* | [,**IOERR** *s*] [,**NOREC** *s*]]
[**REM** *comment*]

General Format Using &BUFF:

RESET &BUFF [REM *comment*]

where *filename* is a character expression; *r* is the relational operator = or .EQ., ≥, >= or .GE.; *exp* is a character expression; *e* is a positive scalar arithmetic expression; *es* is the number of an EXIT statement; *s* is the number of an executable statement; and *comment* is one or more EBCDIC characters.

Action:

Execution of a RESET statement for a specified file positions an internal pointer so that a subsequent GET, PUT, READ FILE, or WRITE FILE reference to the file will refer to a specific item in it. For stream-oriented files, the file is positioned at its beginning.

The error condition specified in the RESET statement causes program control to be transferred to a designated statement if the error condition occurs.

If the END clause is specified, the file is positioned at its end, closed, and activated for output.

If the KEY clause is specified, the file is positioned at the first record whose key satisfies the condition set forth in the KEY clause.

If the NOKEY clause is included, and the record with the specified key does not exist on the file, control is transferred to the specified statement.

If the REC clause is specified, the file is positioned to the relative record identified by the arithmetic expression (that is, the integer part of a positive number).

If the NOREC clause is included, and a record with the relative record number specified does not exist on the file, control is transferred to the specified statement.

If IOERR is included, a device malfunction occurs, and control is transferred to the specified statement.

The RESET statement is used with the &BUFF internal variable for buffered-ahead terminal input (see INPUT statement, Part II). RESET &BUFF results in the following actions taken:

1. The value of &BUFF is reset to zero.
2. All groups of data values (identified by a preceding semicolon) not yet accessed by an INPUT statement are discarded. For example, if the following input line were entered in response to an initial INPUT statement request,

   1,2,3;4,5,6;7,8,9

   the execution of RESET &BUFF before another INPUT statement is executed will result in the data value groups of 4,5,6 and 7,8,9 being discarded.

Rules:

1. If a file specified in a RESET statement is not currently active, its name in the RESET statement will be ignored if no clause is specified for that file. If a KEY or REC clause is specified, the file is activated for input operations. If an END clause is specified, the file is activated for output operations.
2. To add records to an existing stream-oriented file, open the file by specifying RESET with the END clause. This specification causes the file to be positioned to its end. Do not use OPEN with the OUT clause; OUT positions a file to its beginning, effectively deleting the file contents.
3. The END clause cannot be specified in the DOS/VS environment.
4. If the EXIT clause is specified for a particular file, then no other error clause can be specified.
5. In a multiple RESET statement, if an error causes transfer of program control to the statement designated, subsequent files specified are not processed.
6. Naming conventions for files depend upon the environment in which they are used. See Appendix A. "Implementation Considerations."

Examples:

```
 60  GET 'IN', X, Y, Z
          .
          .
          .
120  RESET 'IN'
130  GET 'IN', A, B, C
```

Statement 60 reads the first three values of IN into X, Y, and Z, respectively. The RESET statement in 120 repositions the file named IN to the beginning. The GET statement in 130, therefore, reads the first *three* values of IN into A, B, and C, respectively.

```
200       RESET FILE A$, REC=4, NOREC 300
220       RESET FILE 'ABLE,' KEY=B$, NOKEY 600
```

The RESTORE Statement

Function:

The RESTORE statement causes the next READ statement executed to begin assigning values from the first item in the first DATA statement of the program. For a discussion of how this statement is used, see "Getting Data Into the Computer" in Part I of this publication.

General Format:

RESTORE [[REM] *comment*]

where *comment* is one or more EBCDIC characters.

Action:

The RESTORE statement returns the data table pointer from its current position to the beginning of the table. The optional comment is a character string that does not affect the execution of the statement.

Rule:

A RESTORE statement in a program containing no DATA statements is ignored.

Example:

After the following series of statements is executed, the variables A and C will each have a value of 1, and the variables B and D will each have a value of 2.

```
10  DATA 1,2
20  READ A,B
30  RESTORE
40  READ C,D
```

The RETURN Statement

For use of the RETURN statement in the creation of subroutines, see "The GOSUB and RETURN Statements."

For use of the RETURN statement within a multiline function definition, see "The DEF, RETURN, and FNEND Statements."

The REWRITE FILE Statement

Function:

The REWRITE FILE statement causes the alteration of a record that already exists in a record-oriented file. For a discussion of how this statement is used, see "An Entry-Sequenced File", "A Key-Sequenced File" and "A Relative-Record File" in Part I of this publication.

General Format—Entry-Sequenced Files:

[MAT] REWRITE FILE [USING *sn*] *filename, output-list*

[,EXIT *es*
[,EOF *s*][,IOERR *s*][,CONV *s*]] [REM *comment*]

General Format - Key-Sequenced Files:

[MAT] REWRITE FILE [USING *sn*] *filename* [,KEY *r exp*], *output-list*

[,EXIT *es*
[,EOF *s*][,IOERR *s*][,CONV *s*][,NOKEY *s*]] [REM *comment*]

General Format - Relative Record Files:

[MAT] REWRITE FILE [USING *sn*] *filename* [,REC = *e*], *output-list*

[,EXIT *es*
[,EOF *s*][,IOERR *s*][,CONV *s*][,NOREC *s*]] [REM *comment*]

where *sn* is the number of a FORM statement, *filename* is a character expression, *r* is the relational operator = or .EQ., ≥, >=, or .GE., *exp* is a character expression, *es* is the number of an EXIT statement, *s* is the number of an executable statement and *comment* is one or more EBCDIC characters.

Action:

Execution of the REWRITE FILE statement causes a record to be written over an already existing record in a record-oriented file. Values from the variables in the output list are placed in a buffer associated with the specified file. Values from arrays are placed in the buffer row by row.

If the USING clause is present, values are converted as indicated in the specified FORM statement. (**Note:** An X or POS control specification in the FORM statement causes a spacing over of record positions, permitting a selective rewriting of portions of the record.) In the absence of a USING clause, values are assigned to the buffer without conversion.

If the KEY clause is included, the first record satisfying the specified condition is written over. The character expression is evaluated and truncated on the right, if longer than the record key, before being compared with the actual record key. A character string shorter than the record key is compared

with the first *n* characters of the key, where *n* is the number of characters in the character string.

If the NOKEY clause is included, and no record with a key satisfying the KEY clause condition exists in the file, control is transferred to the specified statement.

If the REC clause is included, the record whose relative record number matches the relative record number specified is written over. The record length of the rewritten record must not be greater than the original record.

If the NOREC clause is included and no record with the relative record number satisfying the REC clause condition exists in the file, control is transferred to the specified statement.

If the EOF clause is inclused, and there is no room for the rewritten record in the file, and control is transferred to the specified statement.

If the IOERR clause is included, and a device malfunction prevents reading or writing of the record, control is transferred to the specified statement.

If the CONV clause is included, control is transferred to the specified statement if a value cannot be converted to the format defined for the item in an associated FORM statement, or if a record larger than the maximum permitted size is written.

If the EXIT clause is included, the specified EXIT statement is examined for the appropriate error condition if any of the above problems occur.

Rules:

1. A file specified in a REWRITE FILE statement must be activated for both input and output operations (that is, ALL must be specified in the OPEN statement for the file).
2. If the KEY clause is omitted from the REWRITE FILE statement for a key-sequenced file, the last I/O operation of the specified file must have resulted from a READ FILE or REREAD FILE statement for the record to be rewritten.
3. If the REC clause is omitted from the REWRITE FILE statement for a relative-record file, the last I/O operation on the specified file must have resulted from a READ FILE or REREAD FILE statement for the record to be rewritten.
4. If the specified file is entry-sequenced, the length of the rewritten record may not exceed the original length of the record. The rewritten record will be the same length as the original record.
5. If the specified file is keyed, the length of the rewritten record can be equal to or greater than the original length of the record; however, the contents of the record positions occupied by the record key may not be altered.
6. If the specified file is relative record, the length of the rewritten record may not be greater than the length of the original record.
7. The EOF, IOERR, and CONV clauses and the NOKEY or NOREC clauses can be specified in any order. The presence of any of these clauses in the REWRITE FILE statement precludes the specification of an EXIT clause.

8. If all of the items in the output list are array references (that is, array names preceded by the keyword MAT), the MAT can be dropped from each of the array references and specified once before the keyword REWRITE.
9. An array cannot be redimensioned in a REWRITE FILE statement.
10. Naming conventions for files depend upon the environment in which they are used. See Appendix A. "Implementation Considerations."

Example:

```
70    REWRITE FILE USING 75 "ABC", KEY='05', M1, M2, M3
75    FORM X5, S, POS30, S, POS40, PIC (ZZZ.ZZ)
```

The STOP Statement

Function:

The STOP statement causes program execution to be terminated. For a discussion of how this statement is used, see "Subroutines" in Part I of this publication.

General Format:

```
          ┌ [REM] comment          ┐
STOP      └ RC = exp [REM comment] ┘
```

where *exp* is a numeric expression which returns the integer part of a non-negative number, and *comment* is one or more EBCDIC characters.

Action:

When a STOP statement is executed, program processing is terminated and all open files are automatically closed. Unlike the END statement, which functions identically at execution time, the STOP statement has no effect on the compilation of the program.

If RC= *exp* is specified a completion code is returned to the host system at the end of program execution.

Rule:

1. A STOP statement may appear anywhere in a BASIC program.

   Example:

```
 75    STOP
```

   Return/Completion example:

```
500 STOP RC=16REM RETURN TO HOST SYSTEM
```

The USE Statement

See "The CHAIN and USE Statements."

The WRITE FILE Statement

Function:

The WRITE FILE statement adds a new record to a record-oriented file. For a discussion of how this statement is used, see "An Entry-Sequenced File", "A Key Sequenced File", and "A Relative Record File" in Part I of this publication.

General Format—Entry-Sequenced Files:

[MAT] WRITE FILE [USING *sn*] *filename ,output-list*

[,EXIT *es*
[,EOF *s*][,IOERR *s*][,CONV *s*]] [REM *comment*]

General Format - Key-Sequenced Files:

[MAT] WRITE FILE [USING *sn*] *filename ,output-list*

[,EXIT *es*
[,EOF *s*][,IOERR *s*][,CONV *s*][,DUPKEY *s*]] [REM *comment*]

General Format - Relative Record Files:

[MAT] WRITE FILE [USING *sn*] *filename* ,REC = *e ,output-list*

[,EXIT *es*
[,EOF *s*][,IOERR *s*][,CONV *s*][,DUPREC *s*]] [REM *comment*]

where *sn* is the number of a FORM statement; *filename* is a character expression; *r* is the relational operator =, or .EQ.; ≥, >=, or .GE.; *e* is a positive scalar arithmetic expression; *es* is the number of an EXIT statement; *s* is the number of an executable statement; and *comment* is one or more EBCDIC characters.

Action:

Execution of a WRITE FILE statement causes a record to be added to the specified file, which can be entry-sequenced, keyed, or relative record. For an entry-sequenced file, the record is added to the end of the file. For a keyed file, the record is inserted in the file at a point determined by its record key, which is one of the fields within the record itself. For a relative-record file, the record is written into the file slot number specified in the REC clause.

When the WRITE FILE statement is executed, values from variables in the output list are placed in a buffer associated with the specified file. Values from arrays are placed in the buffer row by row.

If the USING clause is present, values are converted as indicated in the specified FORM statement. (Note: Record fields that are not filled with data — as the result of a POS or X control specification—are filled with binary zeros (interpreted as blanks), thus creating a "null field" within the record.) In the absence of a USING clause, values are assigned to the buffer without conversion.

For relative record files, the record identified in the REC clause is written.

If the EOF clause is included, control is transferred to the specified statement if there is no room in the file for the new record.

If the IOERR clause is included, control is transferred to the specified statement if a device malfunction prevents writing of the record.

If the CONV clause is included, control is transferred to the specified statement if a value cannot be converted to the format defined for the item in an associated FORM statement, or if a record larger than the maximum permitted size is written.

If the DUPKEY clause is included, control is transferred to the specified statement if the key field of the record to be written matches the key field of a record already in the file.

If the DUPREC clause is included and the relative record number of the record to be written matches a slot in which a record is already written, control is transferred to the specified statement.

If the EXIT clause is included, the specified EXIT statement is examined for the appropriate error condition if any of the above problems occur.

Rules:

1. For a keyed file, the output list must refer to a character value which is to be used as the record key. The length and position of the record key vary with each file and are defined outside of the BASIC program.
2. The EOF, IOERR, DUPKEY, or DUPREC and CONV clauses can be specified in any order. The presence of any of these clauses in the WRITE FILE statement precludes the specification of an EXIT clause.
3. If all of the items in the output list are array references (that is, array names preceded by the keyword MAT), the MAT can be dropped from each of the array references and specified once before the keyword WRITE.
4. An array cannot be redimensioned in a WRITE FILE statement.
5. When referring to a relative-record file, the WRITE FILE statement can only write a record into a null slot that has a length equal to or greater than the record's length.
6. Naming conventions for files depend upon the environment in which they are used. See Appendix A. "Implementation Considerations."

Example:

```
50   WRITE FILE USING 100 "NEW", K$, X1, X2,
     X3, X4
```

APPENDIX A. IMPLEMENTATION CONSIDERATIONS

Time-Sharing Environments

- VSPC:

1. *VSPC Internal Files:* A VS BASIC user, under VSPC, can create, access, and delete VSPC files as either sequential files or as direct files. Sequential files are used for either stream-oriented access (GET/PUT) or as entry-sequenced record-oriented files (READ FILE/WRITE FILE or INPUT FROM/PRINT TO). Direct files are defined for relative-record, record-oriented files (READ FILE/WRITE FILE with a REC clause).

   VSPC internal files are not key-sequenced, so may not be accessed with any statement containing a KEY, DUPKEY, or NOKEY clause.

2. *VSPC External Files:* External files must be defined through the use of VSAM's Access Method Services. They are typically created by the data processing installation personnel. The files are always record-oriented, and are defined in a system library. The VS BASIC programmer must know the library number of that library in order to write programs which can access the data files defined there.

   When entry-sequenced files are defined, the name of the file, the number of records in the file, and the maximum size record must be defined.

   When key-sequenced files are defined, the length and position of the key in the record must also be defined.

   When relative-record files are defined, only the record length and file size are specified.

   For all external files, the DD/DLBL card for each file must be in the VSPC start-up deck.

3. VSPC filenames have the form:

   library number filename/password

   where *filename* is 1 to 8 characters, the first of which must be alphabetic.

   When using VSPC internal files in your own library, the library number may be omitted. When using VSPC external files, or VSPC internal files in another library, the library number must be included.

For example:

```
OPEN FILE '8ABC' IN
```

will open file ABC in library 8.

In addition, a password may be required for data access. See *VS BASIC for VSPC: Terminal User's Guide* for further discussion of files and filenames.

- CMS:

  1. A filename consists of between one and eight alphabetic (including the alphabet extenders) or numeric characters. For example:

     ```
     50   PUT '111A',B1,B2
     ```

- TSO:

  1. A filename consists of between one and eight alphabetic (including the alphabet extenders) or numeric characters, with the first character always alphabetic. For example:

     ```
     100   GET 'FILE1',C1,C2
     ```

  2. A stream-oriented file which has been permanently allocated (either by a DD statement in the LOGON procedure or through an ALLOCATE command issued at the terminal) may not be repositioned to its end by a RESET statement with the END clause.

  3. All record-oriented files must be defined by the data processing installation personnel. For an entry-sequenced file, you must provide the name of the file, the number of records in the file, and the maximum size record the file will hold. For a key-sequenced file, you also must supply the length and position of the key in the record. For a relative-record file you must provide the length of the record and the number of records in the file.

  4. If you have not previously allocated an output stream data file, VS BASIC will dynamically allocate a data set for your output using the following default values: LRECL=562, BLKSIZE=1690, RECFM=VB.

- Both CMS and TSO:

  1. The PAUSE statement causes program execution to be interrupted until you press the carrier return key at the terminal.

Batch Environments

- DOS/VS:

  1. The filename for a stream-oriented file must be a logical unit name, in the form SYS*xxx*, where *xxx* is a number from 001 to the maximum permitted by the installation. For example:

     ```
     100   GET 'SYS010',A,B,C$
     ```

     The filename for a record-oriented file can be any name, consisting of up to seven alphabetic and numeric characters, the first of which must be alphabetic. For example:

     ```
     20   LET B$ = 'GRADE'
     30   WRITE FILE B$, X,Y,Z
     ```

     The filename in this example is GRADE.

     To link the BASIC program to operating system resources, the filename specified in the input/output statement must be specified in the appropriate TLBL or DLBL job control statement. For example:

     ```
     // TLBL  SYS010 ...
     // DLBL  GRADE ...
     ```

2. Positioning to the end of a file is not supported. Therefore, END cannot be used with RESET.
3. The EOF clause on the PUT statement is not supported.
4. The CPU intrinsic function always returns a value of 1.
5. All record-oriented files must be defined by the data processing installation personnel. For an entry-sequenced file, you must supply the name of the file, the number of records in the file, and the maximum size record the file will hold. For a key-sequenced file, you also must provide the length and position of the key in the record. For a relative-record file, you must provide the length of the record and the number of records in the file.

- OS/VS (VS1 and VS2):

  1. A filename can be any name consisting of up to eight alphabetic (including the alphabet extenders) and numeric characters, the first of which must be alphabetic. For example:

     ```
     200   READ FILE 'A12296', MAT G
     ```

     To link the BASIC program to operating system resources, the filename specified in the input/output statement must be specified in the DD job control statement that defines the file. For example:

     ```
     //A12296  DD  ...
     ```

- Both DOS/VS and OS/VS:

  1. The PAUSE statement prints a message but does not cause program execution to be interrupted.

The Separable Library

Installation-written functions can be made available to all users in the same manner as processor-supplied intrinsic functions, by installing them in the library of functions. This separable library facility provides programming convenience and enables VS BASIC users to share such functions. See your installation personnel for details before attempting to write or install such functions.

Printing Output Created on a PRINT TO File

Output from files created using the PRINT TO facility may be printed on a system printer by using the REPRO command of Access Method Services. You may want to contact your installation personnel to use these services; briefly, here are some points to remember when using Access Method Services to print the output:

- Use the REPRO command, not the PRINT command, of Access Method Services.
- Specify the filename used in the VS BASIC program as the name of the input file to the REPRO request.
- Include a DD card for that file.
- Define an output file, with a destination of the normal systems output device.

- The record-format parameter of the output file must be specified as fixed-blocked in order to override the normal system use of the first character of each record as a forms control character.
- Use that output file as the outfile parameter to the REPRO request.

For an OS/VS installation, the following JCL will print the contents of the file:

```
//LISTVSAM  JOB    ACCOUNTING INFORMATION,REGION=256K,
                   MSGLEVEL=1
/*SETUP            SETUP INFORMATION IF REQUIRED
//STEP1     EXEC   PGM=IDCAMS
//SYSPRINT  DD     SYSOUT=A
//FILENAME  DD     DSNAME=DATASET-NAME,DISP=OLD
//PRTFILE   DD     SYSOUT=A,DCB=(RECFM=FB)
//SYSIN     DD    *
   REPRO -
            INFILE(FILENAME) -
            OUTFILE(PRTFILE)
/*
```

APPENDIX B. COLLATING SEQUENCE OF THE VS BASIC CHARACTER SET

The EBCDIC collating sequence of the VS BASIC character set is ordered in ascending sequence according to the internal representation of each character.

Character	Internal Hexadecimal Representation	Name
	40	Blank
.	4B	Point or period
<	4C	Less-than symbol
(	4D	Left parenthesis
+	4E	Plus sign
|	4F	OR sign or vertical bar
&	50	AND sign or ampersand
!	5A	Exclamation symbol
$	5B	Currency symbol
*	5C	Asterisk or multiplication symbol
)	5D	Right parenthesis
;	5E	Semicolon
—	60	Minus sign
/	61	Slash or division symbol
,	6B	Comma
>	6E	Greater-than symbol
?	6F	Question mark
:	7A	Colon
#	7B	Number or pound sign
@	7C	Commercial "at" sign
'	7D	Single quotation mark
=	7E	Equal sign or assignment symbol
"	7F	Double quotation mark
↑	8A	Up-arrow or exponentiation symbol
≤	8C	Less-than-or-equal-to symbol
≥	AE	Greater-than-or-equal-to symbol
≠	BE	Not-equal symbol

Character	Internal Hexadecimal Representation
A,a	C1
B,b	C2
C,c	C3
D,d	C4
E,e	C5
F,f	C6
G,g	C7
H,h	C8
I,i	C9
J,j	D1
K,k	D2
L,l	D3
M,m	D4
N,n	D5
O,o	D6
P,p	D7
Q,q	D8
R,r	D9
S,s	E2
T,t	E3
U,u	E4
V,v	E5
W,w	E6
X,x	E7
Y,y	E8
Z,z	E9
0	F0
1	F1
2	F2
3	F3
4	F4
5	F5
6	F6
7	F7
8	F8
9	F9

GLOSSARY

Alphabet extender. Any one of the following three special characters: #, @, and $.

Alphabetic character. Any of the 26 letters (A through Z) of the English alphabet or any of the alphabet extenders (#, @, and $).

Alphabetic equivalent. The alphabetic character equivalent logical and relational operators that can be used interchangeably with the corresponding special character operators in logical subexpressions. The alphabetic equivalents must be enclosed in periods.

See also logical operators and relational operators. The alphabetic equivalents are:

.EQ.	equal to
.NE.	not equal to
.GT.	greater than
.LT.	less than
.GE.	greater than or equal to
.LE.	less than or equal to
.CAT.	concatenation
.OR.	or
.AND.	and

Alphameric character. A numeric or alphabetic character.

Argument. An arithmetic expression appearing in parentheses following a function name, either in a function reference (either a user-written function or an intrinsic function) or in a pseudo variable. The expression represents a value that the function is to act upon. The function name may or may not be followed by arguments.

Arithmetic array. A named table of arithmetic data items. An array may be implicitly declared through usage or explicitly declared in a DIM statement. VS BASIC allows one- and two-dimensional arithmetic arrays.

Arithmetic constant. A constant with a numeric value. The four forms of arithmetic constants permitted in VS BASIC are: integer, fixed-point, floating-point, and internal.

Arithmetic data item. Data having a numeric value.

Arithmetic expression. An arithmetic constant, a simple arithmetic variable, a scalar reference to an arithmetic array, an arithmetic-valued function reference, or a sequence of the above appropriately separated by arithmetic operators and parentheses.

Arithmetic operator. A symbol representing an operation to be performed upon arithmetic data. The arithmetic operators are:

+	addition and unary plus sign
-	subtraction and unary minus sign
*	multiplication
/	division
↑ or **	exponentiation

Arithmetic variable. The name of an arithmetic data item whose value is assigned and/or changed during program execution. The name consists of a single alphabetic character or an alphabetic character followed by a digit.

Array. A named list or table of data items, all of which are the same type—arithmetic or character. VS BASIC allows one- and two-dimensional arrays.

Array declaration. The process of naming an array and assigning dimensions to it either explicitly (by the DIM statement) or implicitly through usage.

Array element. *See* array member.

Array expression. An arithmetic expression or a character expression representing an array of values rather than a single value. It may be used only in an array assignment statement.

Array member. A single data item in an array; its position is indicated by a subscripted array reference.

Array variable. The name of an entire array. The name consists of an alphabetic character (for arithmetic arrays) or an alphabetic character followed by the dollar sign character, $, (for character arrays).

Assignment. The process of giving values to variables; for example, via LET statements, READ statements, INPUT statements, etc.

Assignment symbol. The symbol =, which is used in an assignment statement to give a value to one or more variables.

BASIC. A programming language designed for interactive systems and originally developed at Dartmouth College to encourage non-programmers to use computers for simple problem-solving operations. The word BASIC is an acronym for **B**eginners' **A**ll-purpose **S**ymbolic **I**nstruction **C**ode.

Binary operator. A symbol representing an operation to be performed upon two data items, arrays, or expressions. The four types of binary operators are: arithmetic, character, logical, and relational.

Branching. Executing a statement other than the next sequential one; for example, via the GO TO statement.

Built-in function. *See* intrunsic function.

Carriage return. *See* carrier return.

Carrier return (CR). The process of ending a line by pressing the appropriate key(s) on your terminal.

Character array. A named table of character data items. An array may be implicitly declared through usage or explicitly declared in a DIM statement. VS BASIC allows one- and two-dimensional character arrays.

Character constant. A constant with a character value. It is always enclosed by a pair of single or double quotation marks.

Character data. Data having a character value, as opposed to a numeric value.

Character expression. A character constant, a simple character variable, a scalar reference to a character array, a character-valued function reference, or a sequence of the above separated by the concatenation operator (| |) and parentheses.

Character operator. A symbol representing an operation to be performed upon character data. The concatenation operator (| |) is the only character operator in VS BASIC.

Character string. A sequence of characters which represents an item of character data.

Character variable. The name of a character data item whose value is assigned and/or changed during program execution. The name consists of an alphabetic character followed by the dollar sign character ($).

Comment. A remark or note included in the body of a program by the programmer. It has *no* effect on the execution of the program; it merely documents the program. Comments are written as a string of characters and may appear as a part of any program statement that has no operands (for example, REM, STOP, END, RESTORE, etc.), with or without the REM keyword, or by using the REM keyword, may be included on all BASIC statements except the DATA and Image statements.

Concatenation. The joining of two character data items by the symbol | | or its alphabetic equivalent .CAT.

Concatenation operator. The symbol | | or .CAT., the alphabetic equivalent, is, used to concatenate, or join, two character data items.

Constant. A value that never changes. VS BASIC has two types of constants: arithmetic and character.

Control specification. (1) One of the specifications X or POS, used in the FORM statement to specify formatting of records in record-oriented files. (2) One of the specifications X, POS, or SKIP, used in the FORM statement to control print line formatting at a terminal.

Data file. *See* file.

Data form specification. (1) One of the specifications B, C, NC, PD, S, L, or PIC, used in the FORM statement to specify formatting of character and arithmetic values in record-oriented files. (2) One of the specifications C or PIC, used in the FORM statement to format character and arithmetic values on a printed line.

Data item. A single unit of data; that is, a constant, a variable, an array element, or a function reference.

Data table. The values contained in the DATA statements of your program. DATA statements are processed in statement number sequence (lowest to highest). The values in each DATA statement are collected and placed in a single table in order of their appearance (left to right).

Data table pointer. An indicator that moves sequentially through the data table, pointing to each value as it is assigned to a corresponding variable in a READ statement. Initially, the indicator refers to the first item in the table. It can be repositioned to the beginning of the table at any time by the RESTORE statement.

Declaration. *See* explicit declaration and implicit declaration.

Delimiter. A character that groups or separates data items.

Digits. 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Dimension specification. The specification of the size of an array and the arrangement of its members into one or two dimensions.

Direct access. The storage or retrieval of data independently of other data in a file, that is, regardless of its location relative to other data.

Dummy variable. A simple variable enclosed in parentheses and placed after the name of a user-written function in a DEF statement. The function performs its defined calculation on the expression value substituted for each dummy variable when the program is executed.

EBCDIC collating sequence. The ordering of character data items according to the Extended Binary Coded Decimal Interchange Code.

Entry-sequenced file. A record-oriented file whose records are stored and accessed in the order in which they are entered.

Error message. A message generated by the computer when an error has been detected.

Executable statement. A program statement that causes an action to be performed by the computer.

Execution error. An error discovered during execution of a VS BASIC program (for example, dividing by zero branching to a non-existing statement number, etc.)

Explicit declaration. The use of a DIM statement to specify the number of members in an array, the number of dimensions in an array, or the length of a character variable.

Exponent (of floating-point format number). An integer constant specifying the power of ten by which the base (mantissa) of the decimal floating-point number is to be multiplied.

Exponentiation. Raising a value to a power.

Expression. A representation of a value, for example, variables and constants appearing alone or in combination with operators. Three forms of expressions are defined in VS BASIC: scalar (arithmetic or character), array (arithmetic or character), and logical.

Extended alphabet. The 26 letters of the English alphabet and the 3 alphabet extenders.

File. A named group of related data items that are stored together. In VS BASIC there are two types of files: stream-oriented and record-oriented.

Filename. A character expression whose value is the name of a file.

Fixed-point constant. An arithmetic constant consisting of one or more digits and a decimal point, and optionally preceded by a sign.

Fixed-point format. The form used to express a fixed-point constant.

Floating-point constant. An arithmetic constant consisting of an integer or fixed-point constant followed by the letter E, followed by an optionally signed one- or two-digit integer constant.

Floating-point format. The form used to express a floating-point constant.

Full print zone. Eighteen horizontal print positions. In a PRINT statement, a comma is used to indicate that a full print zone should be used.

Function. A named expression that computes a single value. *See also* intrinsic function and user-written function.

Function reference. The appearance of an intrinsic function name or a user-written function name in an expression.

Generic key. An argument specified in the KEY clause of a record I/O statement that is less than the full key length defined for a corresponding file.

Implicit declaration. (1) The specification of the number of members in an array or the number of dimensions in an

array, either by a reference to a member of an array or by context (without the array being explicitly specified in a DIM statement). (2) The specification of the length of a character variable by context (without the variable being explicitly defined in a DIM statement).

Input. The transfer of data from an external medium to internal storage.

Input list. A list of variables to which values are assigned from input data; the list can be made up of scalar variables, array member references, pseudo variables, array references, and array references with redimensioning.

Input/output. The transfer of data between an external medium (that is, the terminal typewriter or a file) and internal storage.

Integer constant. An arithmetic constant containing one or more digits, optionally preceded by a sign.

Integer format. The form used to express an integer constant.

Internal constant. An arithmetic constant whose value is supplied by VS BASIC. The name of the internal constants are: &PI, &SQR, &E, &INCM, &LBKG, and &GALI.

Internal storage. A computer's main storage.

Internal variable. A character or numeric value set and changed by VS BASIC that is available to the user in certain operations. The internal variables are &BUFF, &CODE, &ERR, &FILE, &LINE, and &REC.

Interrupt. The suspension of program execution.

Intrinsic function. A function supplied by VS BASIC (for example, SIN, COS, SQR, etc.)

Key. One or more consecutive characters used to identify a particular record in a key-sequenced file.

Key-sequenced file. A record-oriented file whose records are stored and accessed according to keys embedded in the records.

Literal. A symbol or quantity in a source program that is itself data, rather than a reference to data.

Logical expression. A logical subexpression, or two logical subexpressions joined by a logical operator (& or |). Its value is either true or false.

Logical operator. An operator that is used in a logical expression. The logical operators are: & (And) and | (Or).

Long-form precision. Precision whereby, externally, values printed with I-format and F-format have a maximum of 15 significant digits, and values printed with floating-point format have a maximum of eleven significant digits in the mantissa.

Loop. A sequence of instructions that is executed repeatedly until a terminating condition is reached. The FOR statement identifies the beginning of a loop; the NEXT statement identifies the end of it.

Mantissa. In floating-point notation (floating-point format), the number that precedes the E. The value represented is the product of the mantissa and that power of ten specified by the exponent.

Matrix (mathematical). A two-dimensional arithmetic array.

Multiline function. A user-defined function that is defined with more than one statement.

Nesting. (1) The occurrence of a FOR/NEXT loop within another FOR/NEXT loop. (2) The occurrence of a GOSUB statement when one or more GOSUB statements are already active. (3) The use of more than one set of parentheses to indicate the order of evaluation in a complex arithmetic expression.

Nonexecutable statement. A program statement that specifies information necessary for program execution.

Null character string. Two adjacent single or double quotation marks that specify a character constant of 18 blank characters.

Null delimiter. One or more blanks or no characters at all (that is, one data item directly following another data item with no intervening space or delimiter) used in a PRINT statement to specify a packed print zone. data items is a character constant.

Numeric character. Any of the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Operand. A constant, a variable, an array member reference, a function reference, or a subexpression on which an operation is to be performed.

Operator. A symbol specifying an operation to be performed. *See also* arithmetic operator, binary operator, concatenation operator, logical operator, relational operator, and unary operator.

Output. The transfer of data from internal storage to an external medium.

Output list. A list of variables from which values are written to an output file; the list can be made up of scalar expressions and array references.

Packed print zone. A section of a printed line, consisting of a number of horizontal print positions, whose size is determined by the type (arithmetic or character) and length of the data being printed. In the PRINT statement, a semicolon or null delimiter is used to indicate that a packed print zone is to be used.

Padding. The addition of one or more blanks to the right of a character string to extend the string to a required length.

Precision. The number of digits for which significance can be expressed.

Print zone. *See* full print zone and packed print zone.

Priority. A rank assigned to an arithmetic operator; it is used when evaluating an arithmetic expression. The order of priorities, from high to low, is: exponentiation, unary operations, multiplication and division, addition and subtraction. Operations at the same priority level are evaluated as they are encountered (from left to right in the expression).

Program. A logically self-contained sequence of BASIC statements that can be executed by the computer to attain a specific result.

Programmer-defined function. *See* user-written function.

Pseudo variable. The use of an intrinsic function as a receiving variable. STR is the only pseudo variable in VS BASIC.

Record. A collection of related data items treated as a unit.

Record-oriented file. A file in which items are stored in records.

Redimension specification. The assignment of a new dimension specification to an already existing array, via an array assignment statement, a READ statement, an INPUT statement, a GET statement, a READ FILE statement, or a REREAD FILE statement.

Redimensioning. The changing of the number of dimensions or the number of members in each dimension of a previously declared array.

Relational operator. An operator used in a logical subexpression. The relational operators are:

.EQ. or =	equal to
.NE. or ≠ or <>	not equal to
.GT. or >	greater than
.LT. or <	less than
.GE. or >= or ≥	greater than or equal to
.LE. or <= or ≤	less than or equal to

Relative-record file. A file whose records are loaded into fixed-length slots.

Relative-record number. A number that identifies not only the slot in a relative-record data set but also the record occupying the slot.

Remark. *See* comment.

Scalar. A single data item (as opposed to an array of items).

Scalar expression. An arithmetic expression or a character expression representing a single value rather than an array of values.

Sequential access. The retrieval of data according to the order in which the data is stored in a file.

Short-form precision. Precision whereby, externally, values printed with integer format and fixed-point format have a maximum of seven significant digits, and values printed with floating-point format have a maximum of seven significant digits in the mantissa.

Significant digits. All the digits of a number starting with the leftmost non-zero digit.

Simple variable. A scalar variable (but not an array member).

Single-line function. A user-defined function that is defined in one statement (that is, the DEF statement).

Slot. The space for a data record in a relative-record data set.

Special characters. Any characters allowed in VS BASIC that are not alphameric characters.

Statement number. The number which prefaces a VS BASIC statement. It can be up to five digits in length (in the range 00000 to 99999).

Stream-oriented file. A file in which items are stored as a stream of data and retrieved in sequential order.

Subexpression. A group within an arithmetic expression and used by the computer to evaluate that expression.

Subroutine. A program segment (sequence of statements) branched to by a GOSUB statement. The last statement of a subroutine must be a RETURN statement which directs the computer to return and execute the statement following the GOSUB statement.

Subscript. Any valid arithmetic expression (whose truncated integer value is greater than zero) used to refer to a particular member of an array.

Substring. A part of a character string.

System-supplied constants. *See* internal constants.

Terminal. A device resembling a typewriter that is used to communicate with the computer.

Truncation. The deletion of one or more characters on the right of a character string to shorten the string to a required length.

Unary operator. An operator that precedes, and thus is associated with, an arithmetic expression. The unary operators are + (plus) and — (minus).

User. Anyone utilizing the services of a computing system.

User-written function. A function defined by the user in a single-line or multiline function definition.

Variable. A name used to represent a data item whose value may change during execution of a program.

VS BASIC. A language processor derived from the BASIC language and developed by IBM for virtual storage systems. It can be used in time-sharing environments (VSPC, CMS, TSO) and batch environments (OSVS1, OSVS2, DOS/VS, CMS Batch).

Zero suppression. The elimination of leading nonsignificant zeros in a number.

INDEX

(Where more than one page reference is given, the major reference appears first.)

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

VS BASIC Language
GC28-8303-2

Reader's Comment Form

Your comments about this publication will help us to improve it for you. Comment in the space below, giving specific page and paragraph references whenever possible. All comments become the property of IBM.

Please do not use this form to ask technical questions about IBM systems and programs or to request copies of publications. Rather, direct such questions or requests to your local IBM representative.

If you would like a reply, please provide your name and address (including ZIP code).

Fold on two lines, staple, and mail. No postage necessary if mailed in the U.S.A. (Elsewhere, any IBM representative will be happy to forward your comments.) Thank you for your cooperation.

Fold and Staple

First Class Permit
Number 6090
San Jose, California

Business Reply Mail

No postage necessary if mailed in the U.S.A.

Postage will be paid by:

IBM Corporation
P.O. Box 50020
Programming Publishing
San Jose, California 95150

Fold and Staple

VS BASIC Language Printed in U.S.A. GC28-8303-2

IBM®

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)

VS BASIC Language Printed in U.S.A. GC28-8303-2

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)