Xerox EASY

Sigma 6/ 7/ 9 Computers

Language and Operations
Reference Manual

90 18 73A

XEROX

Xerox EASY

Sigma 6/7/9 Computers

Language and Operations Reference Manual

FIRST EDITION

90 18 73A

September 1972

File No.: 1X23
XM29, Rev. 0
Printed in U.S.A.

NOTICE

This publication documents the A00 version of Xerox EASY for Sigma 6/7/9 computers.

RELATED PUBLICATIONS

Title	Publication No.
Xerox Sigma 6 Computer/Reference Manual	90 17 13
Xerox Sigma 7 Computer/Reference Manual	90 09 50
Xerox Sigma 9 Computer/Reference Manual	90 17 33
Xerox Universal Time-Sharing System (UTS)/TS Reference Manual	90 09 07
Xerox Universal Time-Sharing System (UTS)/OPS Reference Manual	90 16 75
Xerox Mathematical Routines/Technical Manual	90 09 06
Xerox FLAG/Reference Manual	90 16 54
Xerox Sigma Multipurpose Keyboard Display/Reference Manual (Models 7550/7555)	90 09 82
Xerox Sigma Message-Oriented Communications Equipment/Reference Manual (Models 7601-7604)	90 15 68
Xerox Sigma Character-Oriented Communications Equipment/Reference Manual (Models 7611-7620-7623)	90 09 81

Manual Content Codes: BP – batch processing, LN – language, OPS – operations, RBP – remote batch processing, RT – real-time, SM – system management, TS – time-sharing, UT – utilities.

The specifications of the software system described in this publication are subject to change without notice. The availability or performance of some features may depend on a specific configuration of equipment such as additional tape units or larger memory. Customers should consult their Xerox sales representative for details.

CONTENTS

APPENDIXES

FIGURES

TABLES

1. INTRODUCTION

EASY is a shared processor operating under the Universal Time-Sharing System (UTS). It enables the user to create, edit, execute, save, and delete program files written in BASIC or FORTRAN. EASY also allows the user to create and manipulate EBCDIC data files. Although intended primarily for teletypewriter operations, EASY can be used with any type of on-line terminal supported by UTS.

To log-on, the user dials the UTS computer and waits for the log-on request

```
UTS AT YOUR SERVICE
ON AT 23:45 NOV 01, '72
LOGON PLEASE:
```

The user then types his account number, identifier, and account password (if any), followed by a carriage return. If automatic association with EASY has been established by the UTS system, control passes directly to EASY. See the Xerox UTS/SM Reference Manual, 90 16 74 for an explanation of automatic association with a processor. Otherwise the user must call EASY by typing

```
EASY
```

followed by a carriage return, in response to the ! from UTS. When EASY takes control of the computer, it prints

```
NEW OR OLD--
```

at the terminal. The user then types NEW if he wants to create a new file, OLD if he wants to work with an existing file.

EASY normally interprets any line of terminal input as a command if it begins with an alphabetic character, and as a program statement if it begins with a digit. Each input line must end with a carriage return.

Depressing the RUBOUT key erases the last unerased character in the current input line. A backslash (\) is printed to indicate each RUBOUT. Depressing the CONTROL and X keys at the same time (or ESC and X in sequence) erases the entire line; a left-arrow (←) or underscore (_) is then printed, the carriage goes to the beginning of the next line, and EASY waits for a new line to be typed by the user. Depressing the BREAK key at any time halts the operation in progress; EASY prints STOP and READY, then waits for further input.

The EASY file system is described in Chapter 2 of this manual, EASY commands are discussed in Chapter 3, and Chapters 4 through 7 explain BASIC programming under EASY. FORTRAN programming is covered in the Xerox FLAG/Reference Manual, 90 16 54. The FORTRAN library routines OPENF and CLOSEF, added to allow FLAG users to open, close, and delete files, are explained in Appendix E of this manual. Additional reference information is presented in the appendixes.

2. EASY FILE SYSTEM

A file is a storage area in secondary memory. It may contain a source program or data records. Each file must have a unique name comprising one to seven alphanumeric characters. A file may also have an optional password of up to four alphanumeric characters. The first character of a file name or password must be alphabetic, and embedded blanks are not allowed.

Files may be permanent or temporary. A temporary file exists only for the duration of the current session and is destroyed when the user logs off. A permanent file remains in secondary storage until its expiration date (see "Disk File ASSIGN Command" in the UTS/BP Reference Manual, 90 17 64) for a discussion of the EXPIRE parameter) or until deleted by the user (see "UNSAVE" in Chapter 3).

A list of all EASY files in a given account is maintained by the UTS system. This information can be listed on the user terminal by means of the CATALOG command (see "CATALOG" in Chapter 3).

To assist the user in creating and editing his program and data files, each user is provided with a temporary "workfile". Program lines or data lines are stored in this area as they are received from the user terminal or retrieved from secondary storage. The contents of the workfile can be listed, edited, executed, and/or copied and saved in permanent file storage for future use.

3. EASY COMMANDS

The user may abbreviate any EASY command word (except DSMOFF) to the first three letters. For example, CAT may be substituted for CATALOG. Commands may be typed at the beginning of any line, unless EASY is in the data storage mode (see "DSM" below). The syntax of EASY commands is indicated in this chapter by the symbology explained in Appendix A.

Wherever the term "file" is used in defining the syntax of EASY commands, this should be understood to represent a file identifier having the general form

[account:] filename[,password]

For example, the command format

OLD [file]

should be understood as though written

OLD [[account:] filename[,password]]

In addition to the other EASY commands, nine editing commands allow the workfile to be modified in a variety of ways. In general, editing operations are performed in the same way for any program file. However, resequencing a BASIC program causes all references to a line number to be changed, while the resequencing of a FORTRAN program changes only the line numbers themselves. None of the editing commands automatically save files after editing; this must be done by the user (see "SAVE" and "REPLACE", below).

NEW The NEW command has the form

NEW [file]

If no file is specified, EASY prints

ENTER FILE NAME--

and the user must respond by specifying the file he wants to create. EASY deletes the current workfile and creates a new, empty workfile having the specified name.

Numbered program statements typed by the user are stored in the workfile in ascending numerical order regardless of the order in which they are typed. Line numbers must be integers in the range of 1 through 99999. Numbers need not be contiguous; that is, gaps may be left to allow for future insertions. Lines are replaced by retyping, and typing just the line numbers causes an existing line to be deleted from the workfile.

Note that a new file is not saved permanently unless a SAVE command is given (see "SAVE", below).

OLD The OLD command has the form

OLD [file]

If no file is specified, EASY prints

ENTER FILE NAME--

and the user must respond by specifying an existing file. The account and password must be specified, if applicable. If the account is not specified, the user's own account is assumed. EASY deletes the current workfile and loads the specified file into the workfile area. The workfile may then be modified by terminal input or via EASY commands. Note, however, that a modified workfile does not replace an old file unless a REPLACE command is given (see "REPLACE", below).

CATALOG The CATALOG command has the form

CATALOG

EASY lists the names of all files saved in the UTS file system under the user's account, with the exception of any names not conforming to the rules for EASY files (e.g., more than seven characters or nonalphanumeric characters).

DSM The DSM command has the form

DSM

EASY responds by entering the data storage mode. In this mode, lines of data may be input without line numbers. Line numbers are supplied by EASY, beginning with the next available line number in the workfile and incrementing by 1. If data input is to be from paper tape, a TAPE command must be given just before typing DSM (see "TAPE", below). A return to the normal input mode is made by depressing the BREAK key or by use of the DSMOFF command (see "DSMOFF", below).

DSMOFF The DSMOFF command has the form

DSMOFF

EASY responds by leaving the data storage mode. DSMOFF is the only EASY command that may be used in the data storage mode. Note that unlike other EASY commands DSMOFF may not be abbreviated.

TAPE The TAPE command has the form

TAPE

This command must be given before activating the paper tape reader. EASY responds by entering the paper tape input mode. This mode is terminated by use of the KEY command (see "KEY", below).

KEY The KEY command has the form

KEY

This command terminates the paper tape input mode (see "TAPE", above). EASY returns to the normal keyboard input mode.

SAVE The SAVE command has the form

SAVE [file]

EASY copies the current workfile into permanent storage. If no file is specified, the name used is that currently associated with the workfile (e.g., via a previous NEW or RENAME command). The workfile is not affected.

The name of the permanent file must be unique within the user's account. Otherwise EASY ignores the SAVE command and prints

FILE ALREADY SAVED-TYPE REPLACE TO OVERWRITE

and the user must either rename the workfile (see "RENAME", below) and give another SAVE command, or use the REPLACE command (see "REPLACE", below) to overwrite the existing file. SAVE cannot be used to create a file in another account.

UNSAVE The UNSAVE command has the form

UNSAVE [file]

EASY deletes the specified file from the user's account. If no file is specified, the name currently associated with the workfile is assumed. The workfile is not affected. UNSAVE cannot be used to delete a file from another account.

RENAME The RENAME command has the form

RENAME [filename]

EASY replaces the current name of the workfile with the name specified. If no name is specified, EASY prints

ENTER FILE NAME--

and the user must specify a file name. This command can be used to create multiple copies of a file. That is, a file can be retrieved from secondary storage via the OLD command, modified as desired, renamed, and saved under a new name with the SAVE command.

REPLACE The REPLACE command has the form

REPLACE [filename][,[new password]][,old password]

EASY replaces the specified file with the contents of the workfile. If no file is specified, the name currently associated with the workfile is assumed. The workfile is not affected. The replaced file must be in the user's account.

If an old password exists, it must be specified. To delete an old password without establishing a new one, the old password is preceded by two commas. If the old password is to be retained, it is preceded by a single comma.

SCRATCH The SCRATCH command has the form

SCRATCH

EASY deletes the contents of the workfile. The name associated with the workfile remains the same and any permanent file of that name is not affected.

SORT The SORT command has the form

SORT [file]

If no file is specified, EASY sorts the workfile. Otherwise, EASY deletes the current workfile and loads the specified file into the workfile area. If duplicate line numbers exist in the designated file EASY renumbers each duplicate line, using the next available larger line number. The specified file is not affected.

LENGTH The LENGTH command has the form

LENGTH

EASY prints the number of records currently in the workfile.

LIST The LIST command has the form

LIST (NH) [line-line / line[,] / file]

If no options are typed, EASY prints the contents of the current workfile preceded by a header line containing the file name, date, and time. The listing may be terminated at any point by depressing the BREAK key. The NH option can be used to supress the header line.

A designated block of lines may be specified, or a line at which listing is to begin (a comma following the line number causes that line only to be listed).

If a file is specified, the entire file is listed. Neither the designated file nor the workfile are affected.

EDIT DELETE The EDIT DELETE command has the form

EDIT DELETE line [-line][,. . .]

EASY deletes the specified lines or blocks of lines from the workfile.

EDIT EXTRACT The EDIT EXTRACT command has the form

EDIT EXTRACT line[-line][,. . .]

EASY deletes all but the specified lines or blocks of lines from the workfile.

EDIT FIND The EDIT FIND command has the form

EDIT FIND "string"[line-line][,column-column][:. . .]

EASY lists all lines in the workfile that contain the specified character string. Any pair of the characters ! @ $ % # ¢ $ / may be used as string delimiters in addition to the quotation marks shown above.

The question mark may be used to indicate embedded character positions that are to be ignored in the target string. For example,

EDIT FIND "A?B"

would select all lines containing an A, followed two columns later by a B (e.g., A+B, A-B, AAB, etc).

The user may specify a block of lines to be searched and/or a block of columns to be searched. The colon may be used as a delimiter to allow a combination of strings to be selected. For example,

EDI FIN "AB",7-8:"C?D",14-16

would select all lines containing the string AB in columns 7 and 8 and characters C and D in columns 14 and 16 (e.g., 150 Y=ABS(X)+C*D).

EDIT REPLACE The EDIT REPLACE command has the form

EDIT REPLACE [limit]"string1" string2" ——┐
└——[line-line][,column-column][:. . .]

EASY replaces string1 with string2. If no limit is specified, an unlimited number of replacements may be made in each line.

The use of string delimiters and line and column specifications is the same as for the EDIT FIND command, above.

EDIT INSERT The EDIT INSERT command has the form

EDIT INSERT file[;file[,line]]. . .

EASY deletes the current workfile and loads the specified files into the workfile area. From two to nine files may be combined, and a point of insertion may be specified for each file. For example,

EDIT INS ONE;TWO,50

would cause file TWO to be inserted into the workfile following line 50 of file ONE. No resequencing of line numbers is done by this command, and only the workfile is affected.

EDIT MERGE The EDIT MERGE command has the form

EDIT MERGE file[;file[,line]]. . .

EASY deletes the current workfile and loads the specified files into the workfile area. From two to nine files may be combined, and a point of insertion may be specified for each file (see "EDIT INSERT", above).

After the specified files have been loaded, the line numbers in the workfile are resequenced (see "EDIT RESEQUENCE", below). Only the workfile is affected.

EDIT PAGE The EDIT PAGE command has the form

EDIT PAGE [file[;page]][;. . .]

If no file is named, EASY prints the contents of the workfile in page format with each 11-inch page containing 50 lines.

Up to nine files may be listed in the order specified. If no page number is specified for a file, the first page printed for that file is numbered as page 1.

EDIT WEAVE The EDIT WEAVE command has the form

EDIT WEAVE file[;. . .]

EASY deletes the current workfile and interweaves the specified files into the workfile area. Up to nine files may be specified. No resequencing of line numbers is done by this command. If duplicate line numbers exist, the most recently loaded lines replace the previous ones. Loading is done in the order specified, and only the workfile is affected. The name of the workfile is unchanged.

EDIT RESEQUENCE The EDIT RESEQUENCE command has the form

EDIT RESEQUENCE [line1][,[line2][-line3] ——┐
└——[,increment]]

EASY resequences the workfile. If line 1 is omitted, the new starting line number is 100. If line2 is omitted, resequencing begins with the first line of the workfile. If line3 is omitted, resequencing continues through the last line of the workfile. If no increment is specified, 10 is assumed. Line numbers must be in ascending order in the file.

SYSTEM The SYSTEM command has the form

SYSTEM [name]

If no system name is specified, EASY prints

NEW SYSTEM NAME--

and the user must type either BASIC or FORTRAN. EASY assumes BASIC when called (e.g., following an !EASY command). The SYSTEM command must be given prior to a RUN command (see below) if the program to be executed is not written in the language currently assumed by EASY.

RUN The RUN command has the form

RUN[MOD][NH] [file]

If no options are specified, the contents of the workfile are compiled by the current system (see "SYSTEM", above). A header line is printed prior to running, and execution terminates when the program halts or when the BREAK key is depressed.

The MOD option causes EASY to weave the contents of the workfile with those of the specified file before compilation. In the event of duplicate line numbers, those of the current workfile are retained. The workfile is cleared when program execution terminates.

The NH option inhibits the printing of the header line. If a file is specified, the current workfile is cleared.

TIME The TIME command has the form

TIME

EASY prints the current time.

TTY The TTY command has the form

TTY

EASY prints the current account number, current workfile name, current system language, and elapsed terminal time.

UTS and TEL

These commands have the form

UTS

TEL

EASY exits to the UTS system, allowing the user to give any TEL command such as a !PLATEN command to inhibit UTS page headings. (See the Xerox UTS/TS Reference Manual, 90 09 07 for an explanation of TEL commands.) The user may return to EASY by giving an !EASY command.

GOODBYE, BYE, HELLO, and RESTART These commands have the form

GOODBYE

BYE

HELLO

RESTART

EASY exits and the user is logged out of the system with the option of logging in under a different account or user name.

4. BEGINNING BASIC

INTRODUCTION

To use a computer, the user must learn a language the computer understands. Xerox Sigma 5-9 computers understand several languages. Most of these are meant for some special purpose such as the solution of scientific, engineering, or business problems. BASIC is intended as an all-purpose language. Although BASIC is often called a "beginner's language", the computational power of a given BASIC program depends a great deal on the experience of the user. An experienced BASIC user should have no difficulty in creating very powerful programs.

Because of its similarity to ordinary English, BASIC is a good language for users who are not professional programmers and who may have no particular interest in the internal workings of the computer. Many BASIC programmers never see the computer they are programming, but communicate by means of a teletypewriter terminal at a remote location. To use a terminal, the user must dial the telephone number of a Sigma 5-9 time-sharing computer and wait for a log-on request to be printed on the teletypewriter. The user must then type his account number, identifier, and account password (if any), followed by a carriage return. An example is shown below.

```
UTS AT YOUR SERVICE
ON AT 12:10 NOV 03, '71
LOGON PLEASE: ACCT9876,ABERNATHY

12:10 11/03/71 ACCT9876 ABERNATHY 38B-B [1]

!EASY
```

As shown above, the computer types a page heading and then an exclamation mark to indicate that it is ready for an executive-level command. The user types the word EASY to indicate that he wants to use the EASY subsystem. The computer is now ready to accept input from the user terminal.

SYMBOLIC NAMES

BASIC recognizes symbolic names representing mathematical variables. Such names may consist of a single letter of the alphabet or a letter followed by a single digit from 0 through 9:

```
X
Y
B4
```

The following are not valid names in BASIC:

```
XX
Y23
4D
```

ADDITION AND SUBTRACTION

Suppose you want to add a series of numbers such as 27.3, 14.1, 6.0, 3.5, and 36.25. One way of doing this is by typing a PRINT statement expressing the desired addition. BASIC will respond by computing the indicated sum and printing the total when Runnh is typed

```
100 PRINT 27.3+14.1+6+3.5+36.25
RUNNH

 87.1500
```

Since BASIC statements cannot be continued from one line to the next, this method will work only if all of the numbers to be added can be typed on a single line of 132 characters. The consequences of this restriction can be avoided by letting part of the sum be represented by a symbolic variable such as the letter P:

```
100 LET P=27.3+14.1+6
110 PRINT P+3.5+36.25
RUNNH

 87.1500
```

The symbolic variable P could have been redefined to represent the final sum before typing the PRINT statement

```
100 LET P=27.3+14.1+6
110 P=P+3.5+36.25
120 PRINT P
RUNNH

 87.1500
```

The statement

```
100 P=P+3.5+36.25
```

is not a mathematical equation in the usual sense. A LET statement in BASIC is actually an "assignment" statement specifying that the current value of the symbol to the left of the "equals" sign is to be replaced by the value of the expression to the right of the equals sign. Note that the word LET in an assignment statement is optional.

Quantities can be subtracted by using a minus sign rather than a plus sign

```
100 PRINT 10-13
RUNNH

-3
```

As in addition, symbols may be used to represent values in an expression involving subtraction:

```
100 LET A=10, B=13
110 PRINT A-B
RUNNH

-3
```

Note that more than one value assignment may be made in a single LET statement, as shown above, if a comma is used to separate each such assignment. The above assignment could have been written

```
100 LET B=A+3, A=10
```

In this case the value of B would be unpredictable, because BASIC executes LET statements from left to right, and A is not assigned the value 10 until the second assignment of the LET statement is performed.

MULTIPLICATION AND DIVISION

Multiplication and division can be done in much the same way as addition and subtraction. The asterisk is used to indicate multiplication, and the slash is used to indicate division. Thus, the product of 2 and 4 could be obtained as shown below.

```
100 PRINT 2*4
RUNNH

 8
```

Parentheses can be used to group two or more quantities:

```
100 PRINT 3*(4+5)
RUNNH

 27
```

Without the parentheses, the above PRINT statement would have produced the value 17 rather than 27, because BASIC would then assume that the value 5 was to be added to the product of 3 and 4. It would not be possible to avoid the use of parentheses by a rearrangement of the above expression to put the addition to the left of the multiplication, as in:

```
100 PRINT 4+5*3
```

This statement would produce the value 19 rather than the desired 27, since BASIC performs any indicated multiplication or division before doing addition or subtraction unless the order of precedence is indicated explicitly by means of parentheses.

Nested parentheses are evaluated from the innermost to the outermost:

```
100 PRINT 2*(3+4*(5+6))/7
RUNNH

 13.4286
```

In the above example the "innermost" subexpression is 5+6. This sum is evaluated first and the result is multiplied by 4 and then added to 3 before the multiplication by 2 is performed. The final operation before printing is the division by 7. Note that the result is rounded to 6 significant digits.

EXPONENTIATION

Exponentiation is indicated by use of the up-arrow operator (↑) or the double asterisk (**):

```
100 PRINT 10**2
RUNNH

 100
```

Within the same level of parenthesization, exponentiation takes precedence over any other indicated operation. That is, it is performed before multiplication, division, addition, or subtraction unless this would conflict with the grouping indicated by parentheses:

```
100 PRINT 2**2*3**(4-2)
RUNNH

 36
```

In the above example, the first operation performed by BASIC is the raising of 2 to the second power. The sum of 4 and -2 is then computed and the quantity 3 is raised to this power. The final operation before printing is the multiplication of 4 (the square of 2) by 9 (the square of 3).

INDEXED REPETITION

Many applications of BASIC require a series of operations to be performed more than once. To make this as easy as possible, two special statements are provided: FOR and NEXT. The FOR statement specifies the conditions under which the repetition is to be done and the NEXT statement indicates the end of the series of BASIC statements that is to be repeated. Each line of the program must begin with a unique number ranging from 1 to 99999. Line numbers need not be contiguous, but lines are executed in ascending order. Many BASIC programmers prefer to begin each program with line 100 and make each line number a multiple of 10, allowing room for changes and additions to the program at a later time.

In the program shown below, FOR and NEXT statements are used to cause BASIC to print three lines.

```
10 FOR I=1 TO 3
20 PRINT I
30 NEXT I
RUN


NAME       UTS/EASY      14:21 AUG 24,'72

 1
 2
 3
```

In the above example, the indexed variable "I" in the FOR statement is assigned an initial value of 1 (by the number 1 following the equals sign). Instead of the letter I, any

letter of the alphabet could have been used, but the letters I through N have become traditional in FOR statements. When the NEXT statement is executed following the printing of the first line, the value of I is incremented by 1 automatically and the loop is executed again. Note that the FOR statement initiates execution of the loop but is not a part of it. When the NEXT statement is executed following the printing of the second line, I is again incremented by 1 and the loop is executed for the third time. When the NEXT statement is executed following the printing of the third line, I is again incremented by 1, giving I a value of 4. Since 4 is greater than 3, the limiting value following the word TO, the loop is not executed again. The message 30 HALT indicates that line 30 was the last line executed in the program.

Often in programming FOR and NEXT loops, one may take advantage of the fact that the indexed variable changes in value as the loop is repeatedly executed. This is illustrated by the following example.

```
10 FOR I=1 TO 4
20 PRINT I**3
30 NEXT I
RUN


NAME        UTS/EASY        14:22 AUG 24,'72

 1
 8
 27
 64
```

In the above example, the cube of the indexed variable is printed each time the loop is executed.

The values of symbolic variables used within a loop can be changed during execution of the loop, as illustrated by the following program which prints the first 8 terms of a Fibonacci series. Note that the statements within the loop are typed indented, to make the extent of the loop more readily apparent. This optional practice is especially recommended for nested loops.

```
10 LET J=0, K=1
20 FOR I=K TO 8
30   PRINT J
40   M=J, J=K, K=K+M
50 NEXT I
RUN


NAME        UTS/EASY        14:24 AUG 24,'72

 0
 1
 1
 2
 3
 5
 8
 13
```

Although the value of K varies as the above program is executed, this does not affect I, since the initial value of I is determined only when the FOR statement is executed and the loop is first entered. Note the use of an optional form of the LET statement, without the word LET, in line 40 of the program.

You may use a LET statement to alter the value of the indexed variable within a loop, in addition to the increment added automatically whenever the NEXT statement is executed.

```
10 FOR I=1 TO 10
20   LET I=2*I
30   PRINT I
40 NEXT I
RUN


NAME        UTS/EASY        14:26 AUG 24,'72

 2
 6
 14
```

In the above program, the initial value of I is 1, as specified by the FOR statement. The LET statement doubles this value and the NEXT statement adds 1 to it. Thus, I has the value 3 at the beginning of the second execution of the loop and 7 at the start of the third execution of the loop. When the NEXT statement has been executed for the third time, I has the value 15. This exceeds the limit of 10 set by the FOR statement, and loop execution stops.

PRINT FORMATTING

The statement

```
100 PRINT
```

causes BASIC to print a single blank line. The statement

```
100 PRINT X
```

causes the value of X to be printed, beginning in column 2. Column 1 is reserved for a possible minus sign. The statement

```
100 PRINT X,Y
```

causes BASIC to print the value of X, beginning in column 2, and the value of Y on the same line, beginning in column 16.

If closer spacing is wanted, a semicolon can be used in place of a comma. The statement

```
100 PRINT X;Y
```

causes BASIC to print the value of X, beginning in column 2, followed by the value of Y with 3 or 4 columns separating the two, so that the value of Y will begin in an even-numbered column.

If a value is negative, a minus sign precedes it.

```
100 X=5, Y=-10, Z=300, N4=20.5
110 PRINT X;Y;Z;N4
RUNNH

 5  -10     300    20.5000
```

A combination of formats can be used.

```
110 PRINT X,Y;Z
RUNNH

 5             -10     300
```

If the user does not want the first value to print in column 2 he can use a comma or semicolon following the word PRINT.

```
110 PRINT ,X;Y
RUNNH

                  5  -10
```

The statement

```
100 PRINT 'THESE WORDS'
```

could be used to print THESE WORDS beginning in column 1.

TABBING

The TAB function is used in PRINT statements to advance the output device to a specified column. For example, the statement shown below causes the teletypewriter to advance to column 12 and print the word HERE.

```
100 PRINT TAB(12)"HERE"
RUNNH

           HERE
```

Note that HERE is a literal text string, identified as such by enclosure in quotes. Either single or double quotes (i.e., 'THIS' or "THIS") may be used to enclose a literal text string.

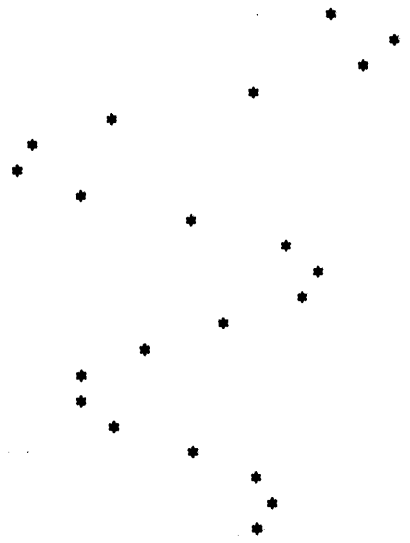
A symbolic or literal value may be used in the same PRINT statement as a literal text string (either with or without a TAB expression). However, since an expression must not follow another expression immediately, a symbolic or literal value must not be used next to a TAB expression. To avoid this difficulty, one can use an empty text string (e.g., "") to separate two expressions in a PRINT statement.

```
100 X=5
110 PRINT TAB(6)""X" APPLES"
RUNNH

     5 APPLES
```

A TAB expression may contain symbolic as well as literal values, allowing great flexibility in line format. This capability is very useful in programming for graphic output.

For example, the following program produces a graphic plot of a damped sine wave:

```
100 X=X+.7, K=EXP(-X/15)
110 PRINT TAB(15+15*K*SIN(X))"*"
120 IF INT(X)=14 THEN 140
130 GOTO 100
140 STOP

READY
RUNNH
```



```
     140 HALT
```

In this example, the EXP intrinsic function returns the value of e (the quantity 2.7183...) raised to the power of the argument. The IF statement causes BASIC to return to statement 100 unless the current value of X is equal to or greater than 15.

DATA INPUT FROM A TERMINAL

The INPUT statement is used to solicit input via the user terminal. A question mark is printed by BASIC to prompt the user.

```
100 PRINT "ENTER LENGTH & WIDTH"
110 INPUT L,W
120 PRINT 'AREA='L*W
130 END
RUNNH

ENTER LENGTH & WIDTH
?5, 10
AREA= 50

     130 HALT
```

In the above example BASIC prints a request to type values for length and width. The values 5 and 10 are typed following the prompt character. Note that the comma after the value 5 is optional. A single blank would be sufficient to separate the two values.

ERROR MESSAGES

BASIC messages to the user are explained in Chapter 7. Most of these messages inform the user of a syntax error in a program line, a logical error in program structure, or a pragmatic error in program execution. Syntax and logical errors are detected at compile time, and pragmatic errors are detected at run time.

The program shown below contains a FOR statement without a corresponding NEXT statement.

```
100 FOR I=1 TO 5
110 PRINT ,'X='I;'Y='I**3
120 END
RUNNH

 MISSING NEXTSTMT
```

When the above program is compiled, BASIC prints the message

```
MISSING NXTSTMT
```

The user can correct the program by typing

```
115 NEXT I
```

and then recompiling by typing another RUN command.

In executing the following program, a divisor becomes zero, causing BASIC to print the message shown below.

```
100 FOR X=1 TO 2
110 FOR Y=1 TO 4-X
120 Z=1/(X**2+3*X*Y-X-Y**2)
130 PRINT X,Y,Z
140 NEXT Y
150 NEXT X
RUNNH

 1             1             .500000
 1             2             .500000

     120 DIV BY ZERO
```

The user can correct this by typing

```
115 IF X**2+3*X*Y-X-Y**2=0 THEN 117
116 GOTO 120
117 Z="  INF."
118 GOTO 130
```

Note that line 116 above assigns a six-character literal text string to the name Z. Z then denotes an alphanumeric constant, or "aconst", as discussed in Chapter 2.

PROGRAM MODIFICATION

A line in a program can be changed by retyping the entire line. A new line can be added to a program by typing it, giving it any unused line number within the desired area of the existing program.

```
100 R=12, Y=-12.5
110 Y=Y+1, X=SQR(R**2-Y**2)
120 PRINT TAB(36-X)"**"TAB(36+X)"**"
130 IF INT(Y)=11 THEN 150
140 GOTO 110
150 STOP
```

The user could add a line between 110 and 120, using any line number from 111 through 119.

```
115 X=1.7*X
```

If the user wanted to combine lines 110 and 115, in the above example, he could do so by retyping line 110 and deleting line 115 by typing the line number followed by a carriage return.

```
110 Y=Y+1, X=1.7*SQR(R**2-Y**2)
115
```

In the following chapters, certain conventions have been adopted for defining the BASIC commands. Capital letters indicate command words that are required in the literal form shown. Lowercase letters are figurative representations of constants, step numbers, etc. Command parameters enclosed by braces ({ }) indicate a required choice. Parameters enclosed by brackets ([ ]) are optional. Ellipsis marks (...) signify optional repetitions of the preceding bracketed parameter. BASIC recognizes the period as a decimal point, not as a terminator.

ELEMENTS OF A BASIC PROGRAM

There are a number of elements common to most BASIC programs. These are: line numbers, simple constants and variables, arithmetic operators, expressions, and intrinsic functions. In addition to these, BASIC programs involving text manipulation often use alphanumeric constants as well as string literals, variables, and expressions.

LINE NUMBERS

Every line in a BASIC program must begin with a unique integer. Line numbers may range from one through 99999 but need not be contiguous, allowing for insertions. Lines are executed in ascending sequence, except where the sequence of execution is modified by branching or looping. Leading zeros are permitted in line numbers but are not required.

NUMBER RANGES

Because BASIC converts all input values to an internal double precision floating-point binary format, the appearance of input values and output values may differ due to rounding during input conversion.

INPUT

BASIC handles input numbers within a range of 5.398*10 -79 through 7.237*10↑75, and zero. Up to 16 significant decimal digits can be input.

OUTPUT

Output numbers are printed in fields of varying widths across the page according to the following rules.

1. Numbers are left-justified in their fields.
2. Positive numbers are preceded by a blank, negative numbers by a minus sign.
3. If the number is a whole number (integer) whose magnitude is less than 1,000,000,000 (10^9), it is printed in from 1 through 9 positions after a blank or a minus.
4. If the number is nonintegral or its magnitude is greater than or equal to 10^9 (for example, -10.5, .5, 123.45, or 10^{12}), its most significant part will be rounded to 6 or 16 digits according to the PRC function and will be treated as follows:

   a. If, after rounding, the absolute value of the number is greater than or equal to 0.1 but less than 10^6 or 10^{16}, the number is printed, in 8 or 18 print positions, in fixed-point notation; that is, its form will be a blank or minus, a maximum of 6 or 16 integer digits, followed by the decimal point and a maximum of 6 or 16 digits.

   b. If, after rounding, the absolute value of the number is less than 0.1 or greater than 10^6 or 10^{16} the number is printed in floating-point (scientific) notation; that is, its form will be a blank or minus, the most significant integer digit, a decimal point, 5 or 15 decimal digits, followed by the letter E, a plus or minus, and a 2-digit exponent.

PRECISION CONTROL

Output precision can be controlled by use of the PRC intrinsic function in a PRINT statement. Used by itself or embedded in a series of other PRINT elements, PRC(1) sets the output precision to 16 significant figures. This precision remains in effect until reset to the default value of six by a PRC(0). An example is shown below.

```
100 LET I=1/30
110 PRINT I
RUNNH

 3.33333E-02

     110 HALT

USED      .3 SECS

READY
110 PRINT PRC(1),I
RUNNH

 3.333333333333333E-02
```

SIMPLE CONSTANTS

A simple constant (that is, a nonvarying quantity) is composed of digits that stand alone, have an embedded decimal point, or are preceded or followed by a decimal point. For example, 2, 7.8, .5, and 12 are simple constants in BASIC.

Simple constants may be modified by floating-point notation, as in 2.5E-15, where the E denotes that the number that precedes it is to be multiplied by 10 to the plus or minus power following the E. Accordingly, the number 2.5E-15 is

really the number .0000000000000025. The plus sign is optional for positive powers.

SIMPLE VARIABLES

A simple variable is denoted either by a single letter or by a letter and a digit from 0 through 9. This convention allows the programmer a total of 286 simple variables. For example, A and W3 are simple variables. Note that if the letter-and-digit combination is used, the letter must precede the number.

ARITHMETIC OPERATORS

BASIC uses common mathematical symbols to denote arithmetic operations. These arithmetic operators are shown in Table 1 below. Note that either the up-arrow (or circumflex) or double asterisk is allowed as an exponentiation operator.

Table 1. Order of Arithmetic Operations

Order	Symbol	Explanation
1	↑ or **	Exponentiate
2	* and /	Multiply and Divide
3	+ and -	Add and Subtract

The table also shows the order of precedence of the various operations. When no operation takes precedence over another, the computer will perform operations from left to right. The order of operations may be altered by use of parentheses. Use of parentheses is advised if the sequence of operations seems questionable.

Note that an operator of order 1 or 2 may be followed by an operator of order 3, but that no other cases of consecutive operators are permitted.

INTRINSIC FUNCTIONS

BASIC provides a total of 34 intrinsic functions. The functions are listed in Appendix C. When a function is used in a statement, the three-letter function name must be followed by an expression or value enclosed in parentheses. This expression or value is called the "argument" of the function. The value of the argument is either used directly in the function calculation, or signals the computer to perform the calculation in a predetermined manner. The purpose of most of the functions is obvious and familiar. The INT function is often used to acquire the integer part of a calculated number. For example, INT(A), where A is computed to be 2.675, would produce the number 2. The INT function may also be used to obtain three significant digits (with rounding) as in the following example:

```
50 LET S=INT((A*100)+.5)/100
```

When statement 50 is executed, S is assigned the value 2.68.

ARITHMETIC EXPRESSIONS

The term "expression", often abbreviated "expr", represents a simple constant, simple or subscripted variable (see Chapter 3), or function reference that may stand alone or may be used in any combination when separated by the symbols for addition, subtraction, multiplication, division, and exponentiation. The components may also be enclosed by parentheses. The symbols + and - may also be the initial character of an expression and may immediately follow a left parenthesis. Some typical expressions are

```
A + 1
(B-X)/D
```

and

```
(2↑X) + SIN(Y)
```

STRING LITERAL

A string literal in a BASIC program is any sequence of text characters, including blanks, enclosed by single or double quotes. If a string literal is enclosed by single quotes, a single quote must not appear in the string. A parallel rule applies to the use of double quotes. Examples of string literals are shown below.

```
100 PRINT 'THIS IS A STRING LITERAL'
110 PRINT "THIS IS 'ANOTHER ONE'"
120 PRINT 'AND "THIS" IS A THIRD'
```

ALPHANUMERIC CONSTANTS

Besides the simple constants previously mentioned, BASIC allows symbolic names such as X, R, or K2 to be assigned string literals of up to six characters. Such symbols are then called "alphanumeric constants" or "aconsts". For example,

```
100 A='ACONST'
110 Z="SPACE3"
120 X6=' X RAY'
```

An "aconst value" can be assigned via a LET, INPUT, READ, GET, MAT READ, MAT INPUT, or MAT GET statement and can be tested, for equality or inequality only, via an IF statement.

```
100 IF A='ALPHA' THEN 200
110 IF Z="OMEGA" THEN 300
```

An aconst value may occur in unquoted form as an element of a DATA statement or in the response to an INPUT request. If an element begins with a digit, plus or minus sign, or decimal point it is assumed to be a number. If it starts with any other character, or is enclosed in quotes, it is assumed to be an aconst value.

Example:

```
100 DATA 5, '5', .5, "FIVE"
```

The first and third elements in the above example are treated as numbers. The second and last are treated as aconst values.

Similar rules apply to elements entered in response to an INPUT request, except that in UTS BASIC if the input is assigned to string variables (see below) all elements are interpreted as strings.

STRING SCALARS

String scalars are denoted by a letter and dollar sign. A string scalar consists of up to 72 characters comprising a single string. A string vector is a one-dimensional array, each element of which is a single string. A string matrix is a two-dimensional array of such elements. String vectors and matrixes are discussed in Chapter 3. To avoid conflict, the same letter must not be used to designate a string scalar and a string (or numeric) array.

Examples of string scalars are

```
100 A$='A STRING'
110 B$="ANOTHER STRING"
```

String scalars can be compared for relative magnitude as well as equality.

```
100 A$='THIS'
110 B$="THAT"
120 IF A$=B$ THEN 200
```

In this example, the comparison of A$ and B$ fails on the third character. Thus, since the character "A" is lower in the EBCDIC code than "I" (see the UTS/TS Reference Manual, 90 09 07, Appendix A) a branch to line 300 is taken.

ASSIGNMENT STATEMENTS

This section discusses the assignment of simple variables, alphanumeric constants, and string scalars. The assignment of values to vectors and matrixes is explained in Chapter 3.

LET The LET statement replaces the current value of the variable(s) on the left of the equals sign with that of the expression on the right of the equals sign.

LET statements have the general form

line [LET]variable[,variable]... = expression

[,variable][,variable]... = expression

where

variable is either a simple variable, an alphanumeric constant, or a string scalar.

expression is an arithmetic expression (see Arithmetic Expressions, above) if the variable is a simple variable. For alphanumeric constants, expressions are either string literals of up to six characters or else string expressions assigning strings of up to six characters (see "Character String Manipulation", Chapter 3).

For string scalars, expressions are either string expressions or string literals of up to 72 characters. Strings of excess length are truncated to the maximum permitted.

Arithmetic operations can be performed by use of the LET statement.

```
100 A=A+1
110 B=A**2
```

Lines 100 and 110 above could be combined into a single statement.

```
100 A=A+1, B=A**2
```

Such serial assignments are executed from left to right. Thus, if A is initially 2 then B will be assigned a value of 9. Parallel assignments can also be made.

```
100 A,B,C=D*E
110 F,G=F+1
```

Note that line 110 above is equivalent to

```
110 F=F+1, G=F
```

Other examples of LET statements are shown below.

```
100 P,A1,Q3=4*ATN(1), A=1, B=5
110 C='STRING'
120 D$='A LONG STRING'
130 E$=C, F=D$
```

In executing line 130 above, aconst F is assigned the string A LONG because of the aconst length limitation.

BRANCHING

Normally BASIC executes program lines in ascending order, beginning with the lowest numbered line. The statements discussed below cause BASIC to alter the normal order of execution, either conditionally or unconditionally. (See also, "Branching to a Subroutine", in Chapter 6.)

IF . . . THEN The IF ... THEN statement provides a conditional branching capability. If the test condition specified in the IF ... THEN statement is true, then the next line executed is that specified in the IF ... THEN statement. Otherwise, the statement following the IF ... THEN statement in the normal sequence is executed.

The form of the IF ... THEN statement is

line IF expr operator expr{THEN / GOTO} line

The condition to be tested is specified between the IF and the THEN (or GOTO) of the statement. The line to which BASIC is to branch on a true test is specified after THEN (or GOTO). An expression may be a simple constant, variable, alphanumeric constant, literal string or string scalar, or a compound arithmetic expression. The operators that may be specified are given in Table 2.

Table 2. Condition Operators

Operator	Explanation
=	Equal to
>< or <>	Not equal to
<	Less than
>	Greater than
< = or = <	Less than or equal to
> = or = >	Greater than or equal to

Examples of IF ... THEN statements are given below:

```
100 IF X=2 THEN 120
110 IF Y="EUREKA" THEN 130
120 IF A$=Z$ THEN 140
130 IF SIN(X+J)=COS(X*K) THEN 100
140 IF B=C GOTO 110
```

ON . . . GOTO If many different branches are to be taken according to the value of some expression, the use of a separate IF ... THEN statement (see above) for each branch becomes unwieldy. To overcome this inconvenience, BASIC provides the ON ... GOTO and GOTO ... ON statements.

ON ... GOTO takes the form

line ON expr{GOTO / THEN} line,line,...

where

expr is any arithmetic expression.

line,line,... is a list of program line numbers.

When the statement is executed, the expression is evaluated and, if necessary, truncated to an integer. If the resulting value is 1, a branch is made to the first line specified in the list. If the value is 2, a branch is made to the second line specified, and so on. If the value is less than 1 or greater than the total number of lines specified in the list, no branch is taken and the next statement in the normal sequence is executed.

Example:

```
100 ON SGN(X)+2 GOTO 150,250,200
```

In the above example, if X is negative a branch to line 150 is taken, if X is 0 a branch to line 250 is taken, and if X is positive a branch to line 200 is taken.

GOTO . . . ON This statement is identical in operation to ON ... GOTO (see above). It has the form

line GOTO line,line,...ON expression

Examples:

```
100 GOTO 140,160,180 ON Y
110 GOTO 200,250,300 ON Z+3
```

GOTO The GOTO statement can be used to alter the normal sequence of program execution unconditionally. The GOTO statement has the form

line GOTO line

GOTO is generally used in conjunction with a conditional branch such as IF ... THEN (see above). An example is shown below.

```
100 IF TIM(1)=12 THEN 120
110 GOTO 100
120 PRINT "LUNCHTIME"
130 IF TIM(1)=12 THEN 130
140 GOTO 100
```

DATA OUTPUT

PRINT The PRINT statement tells the computer to print out the current value of a variable, the results of a calculation, a message, or any combination of these items. The PRINT statement has the general form

[line]PRINT expression(s), or text string(s) with

commas or semicolons

where the word PRINT is usually followed by the name of the item that is to be printed. Two sample PRINT statements are

```
60 PRINT X1,X2
70 PRINT 'NO REAL ROOTS'
```

Line 60 will print out the calculated values of the variables X1 and X2. Line 70 will cause the message enclosed by single quotation marks to be printed out. Note that a string of text must be enclosed by either single or double quotation marks. Blanks, which usually enhance the appearance of text, may be freely interspersed within the string and will be reproduced in the output as presented. More than one text string may be present in a PRINT statement. Each separate string, however, must be enclosed by quotation marks.

A PRINT statement may contain a reference to an intrinsic function. For example

```
1220 PRINT SQR(X)
```

will calculate and print the square root of the variable X, while

```
1230 PRINT X;SQR(X)
```

will print the current value of the variable, followed by its square root.

PRINT can also contain variables, providing that the variables have been defined in statements preceding the PRINT. The following statement is an example of this use of PRINT.

```
1250 PRINT B*B-4*A*C
```

Similarly, the statement

```
1260 PRINT (7/8)**14
```

will give the value of the fraction 7/8 raised to the 14th power.

The word PRINT, used alone in a statement, causes the printer to advance the paper by one line. An example is shown below.

```
450 PRINT
```

PRINT FORMATS

Punctuation marks in the PRINT statement (commas and semicolons) define the desired appearance, or format, of the printed output. The punctuation marks tell the print device at which position to start printing. BASIC has two types of output format, regular and packed. Regular format is specified by using commas to separate elements in the PRINT statement; packed format is specified by using semicolons.

Regular Format. When the regular format is specified by using commas to separate the elements in the PRINT statement, the print line is thought of as consisting of a series of 14-character fields. Each comma causes a shift to the next field. For example,

```
4040 PRINT A,B,C
```

will cause the values of A, B, and C to be printed at 14-character intervals, as in

```
1             5             4
```

When the regular format option is specified, at least two blanks follow the last printed character. In some cases, this spacing may cause an extra field shift.

Packed Format. Packed format, which is specified by using semicolons to separate the elements in the PRINT statement, causes the printed output to be compressed on the page by reducing the spacing between fields. Each semicolon causes a field shift that is either two or three positions in length, so that the shift reaches an even-numbered position. For example, statement 4040 above could be written as

```
4040 PRINT A;B;C
```

with the resultant output of

```
1  5  4
```

Additional Format Considerations. It is important to note the difference between the TAB function and the output format characters. TAB causes output to be printed at a specified position, and is most useful in providing columnar output. The output format characters (, and ;) cause the

output to be printed at intervals that depend on the number of preceding (printed) characters.

If a PRINT statement terminates with a TAB to a column to the left of the current print position, e.g., TAB(0), the line is buffered but not printed until a subsequent PRINT is executed. This allows effective continuation of PRINT statements.

An expression may not follow another expression in a PRINT statement, but a text string literal may be used anywhere. Thus, a null string (two quotes) may be used to separate two expressions. For example,

```
5 PRINT A(1)""B(2)
```

If a PRINT statement ends with a punctuation mark, the appropriate field shift takes place and subsequent printing starts at that point on the same print line. Otherwise, subsequent printing starts on a new line.[†] All printing is left-justified in its field. If a field shift places a field in a position to extend beyond the last allowed print position, a new line is generated and the field is printed on the new line.

This procedure is modified when printing text. If a text string overflows the last position, the string is truncated at that point and the remainder is printed on the next line.

Text strings may extend beyond any number of field boundaries. If neither a comma nor a semicolon appear on either side of a text string in a PRINT statement, no spacing will occur before or after the string in the printed output.

Format characters may be used alone in PRINT statements, or they may be used in any number and combination to cause appropriate field shifts.

PRINTUSING and :(Image) As an alternative to use of the PRINT statement (see above), BASIC provides another method of specifying the format of printed output. This method makes use of a PRINTUSING statement and an associated Image statement. The PRINTUSING statement contains parameters to be inserted into the print positions specified by the referenced Image statement.

PRINTUSING takes the general form

line PRINTUSING line [,expression(s) or text string(s)]

where the line number that follows the command word designates the Image statement into which this PRINTUSING's parameters will be embedded. A sample PRINTUSING statement is shown below.

```
50 PRINTUSING 75,X,SQR(X),'SQ. ROOT'
```

The parameters of line 50, that is, the current value of X, the square root of X, and the text string 'SQ. ROOT', will be embedded, from left to right, in the designated fields of line 75 (a field is a group of character positions that is treated as a distinct unit). Note that commas are used to separate parameters in PRINTUSING.

The Image statement (identified as such by a colon after the line number) complements the PRINTUSING statement in that Image statements depict the final printed appearance of PRINTUSING parameters. An Image statement has the form

line :[#s and/or characters to 132 max.]

where the characters that follow the required colon are governed by the following rules.

1. Each digit position is designated by a # symbol. Also, text strings to be derived from the PRINTUSING statement are indicated by # symbols. For example, the statements

```
50 PRINTUSING 75,X,SQR(X),'SQ. ROOT'
75 :IF X=#, # IS ITS ########
```

will generate the following output (assuming X is currently 4):

```
IF X=4, 2 IS ITS SQ. ROOT
```

2. If a field is preceded by a plus sign, positive values are preceded by a plus sign and negative values by a minus sign. On the other hand, if a field is preceded by a minus sign, positive values are preceded by a blank, negative values by a minus sign. For example, the statements

```
5 :-##, -##, +##, +##, ##
14 PRINTUSING 5,-19,+20,-21,20,99
```

will generate

```
-19,  20, -21, +20, 99
```

3. The decimal point is denoted by a . symbol. For example, the statements

```
3 :#.# AND -.## ALSO +###.
23 PRINTUSING 3, 1.2, -1/4, 100.435
```

will generate

```
1.2 AND -.25 ALSO +100
```

[†] In BASIC, a line ending with a punctuation mark is buffered but is not printed until execution of the next PRINT statement that does not end with a punctuation mark.

4. If a field contains a decimal point, the user may also append four trailing exclamation points to signify floating-point notation. (If more or fewer than four exclamation points are shown, they will be printed literally in the output.) BASIC determines the need for floating-point notation according to the rules given for the PRINT statement. The four !'s provide for a letter E, a plus or minus sign, and a two-digit exponent. Note that a decimal point may be placed anywhere in the field, but that, on printing, it will follow the first digit. The position of the floating-point notation remains unchanged. For example, the statements

```
98 PRINTUSING 99, 1/30, 2/30
99 :VALUES ARE #.##!!!! AND +###.!!!!
```

will generate

```
VALUES ARE 3.33E-02 AND +6.67E-02
```

5. Except for the #, period, and ! symbols, characters that follow the colon will be printed exactly as shown, with spacing as provided by blanks in the Image statement.

6. Text strings may be inserted in fields containing decimal points or specifying floating-point notation. In addition, if the field is preceded by an algebraic sign, its position will be preempted by the text string. For example, the statements

```
11 :THE VALUE IS -#.####
47 PRINTUSING 11,'TOO BIG'
```

will generate

```
THE VALUE IS TOO BIG
```

If a text string is larger than its corresponding field, it will be truncated on the right.

In addition to the above rules, printing is subject to the following conventions.

1. If the field to the left of a decimal point is not large enough to contain a numeric value, asterisks are inserted in the printed output as a warning to the programmer. For example, the statements

```
50 PRINTUSING 75, X, SQR(X), 'SQ. ROOT'
75 :IF X=#, # IS ITS ########
```

will generate, assuming an X value of 25,

```
IF X=*, 5 IS ITS SQ. ROOT
```

Also, if a negative value is associated with a field containing no sign position, a minus will appear in the first position, and any remaining field positions will contain asterisks.

2. If an Image field is larger than necessary, the printed output will show blanks preceding expression values and following text string values up to the required number of positions in the field.

3. If a PRINTUSING statement specifies more values than there are fields in the complementary Image statement, the Image statement is repeatedly used until all PRINTUSING values are printed. For example, the statements

```
8 :N=#####
90 PRINTUSING 8, 1, 4, 90, 81777
```

will generate

```
N=     1
N=     4
N=    90
N=81777
```

4. If a PRINTUSING statement specifies fewer values than there are fields in the complementary Image statement, the printout will be terminated at the first unused field of the Image statement. For example, the statements

```
9 :######## CASES ##.# RESULTS
103 PRINTUSING 9, 'NO MORE'
```

will generate

```
NO MORE  CASES
```

Whenever a PRINTUSING statement is executed, printing starts at the left of a new line. Values are rounded approximately prior to printout. Although the programmer may specify numeric fields greater than 16 characters in length, only 16 significant digits are output (with trailing zeros to fill out the field) for the fractional part of values.

Note that PRINTUSING will accept up to 132 character positions.

PAGE The PAGE statement can be used to advance the paper to the top of the next page. The following example shows how PAGE and PRINT statements might be used to produce a page of tabular data (see Figure 1).

Note that before EASY is called, a PLATEN command is used to set the page length to 18 lines rather than the standard 54 lines per page. A PLATEN,0 command given prior to calling EASY would have caused both PAGE statements to be ignored.

```
!PLATEN ,18

!EASY

NEW OR OLD--NEW EXAMPLE

READY
100 PAGE
110 PRINT ,'X', 'X SQUARED', 'X CUBED'
120 PRINT
130 X=X+1
140 PRINT ,X, X**2, X**3
150 IF X=10 THEN 170
160 GOTO 130
170 PAGE
RUNNH



10:33 08/25/72 356101 2F-35       (13)


                X              X SQUARED      X CUBED

                 1              1              1
                 2              4              8
                 3              9              27
                 4              16             64
                 5              25             125
                 6              36             216
                 7              49             343
                 8              64             512
                 9              81             729
                 10             100            1000




10:34 08/25/72 356101 2F-35       (14)


     170 HALT
```

Figure 1. Use of PAGE Statements

DATA INPUT

DATA and READ The data values used in the execution of a program may be contained in a DATA statement. They are called into use at appropriate times by the READ statement. READ and DATA are used in combination with each other.

DATA statements form a chained list of constants that the READ statement accesses from left to right, top to bottom.† DATA takes the form

line DATA [constant] [,[constant]]...

Simple constants may be preceded by a plus or minus sign, an empty field after DATA, as in

```
1250 DATA
```

or an empty field between commas or after a terminating comma, as in

```
1260 DATA 1,2,3,,5
1270 DATA 6,7,8
```

imply a value of zero. DATA statements may appear anywhere in a BASIC program, and do not have to be consecutive. However, it is good practice to group the DATA statements at the end of the program, thereby making it possible to add as many statements as are needed to contain the data values without disrupting the order of the preceding statements.

READ assigns (in consecutive order) the values in the DATA statement(s) to the variables listed in the READ statement. The form of the READ statement is

line READ variable[,variable]...

There is no comma following the final variable in the list.

Example:

```
555 READ B,C,D
```

If a READ statement requests data after the list of constants in the DATA statement has been exhausted, execution of the program ceases and a message is output to the programmer advising him of the out-of-data condition.

The list of variables following a READ statement may include either of two special entities. A single asterisk means to take an error exit if the current DATA statement list has not been completely read. A double asterisk means to skip any unread elements in the current DATA statement.

Examples:

```
500 READ X,Y,**
510 READ A,B,C,*,D,E,F
```

†The DATA statement may also contain alphanumeric constants or text strings.

Suppose the program includes these DATA statements:

```
1000 DATA 1,2,31.5
1010 DATA 3,4,5,6
1020 DATA 7,8,9
```

When line 500 is executed, 1 is read into X, 2 is read into Y, and the 31.5 is skipped. When 510 is executed, 3 is read into A, 4 into B, and 5 into C. The single asterisk is encountered with 6 left in statement 1010 and the error message EXTRA INPUT results.

INPUT The INPUT statement requests data from a source that is external to the program, that is, teletype unit, or other input device. (INPUT differs from READ in that when using READ, the DATA statement and its data values are contained within the program itself.) Data may be stored in an external medium for two reasons: either the data is unknown when the program is written but will be supplied when the program is run, or the amount of data is too large for inclusion in the body of the program. The INPUT statement takes the form

line INPUT variable[,variable]...

There is no comma following the final variable in the list. When the INPUT statement is executed, data values are read into the computer from the external storage medium and are assigned, one at a time, to the variables designated in the INPUT statement. It should be emphasized that data is stored as it is received, and that the variables are satisfied (that is, associated with the data) in the order in which they are specified. Some sample INPUT statements are shown below.

```
100 INPUT X
110 INPUT A,B,Z,Y,R3
120 INPUT B(1,N), C(N), N
130 INPUT N, B(1,N), C(N)
```

In the above example, every time statement 100 is executed, the computer will supply a data value to the variable X. Statement 110 will supply data values to A, B, Z, Y, and R3, in that order, from the list of data supplied by the programmer. Statements 120 and 130 will very probably not be equivalent, even though the same variables are specified in both. They will not be equivalent even if the data values are supplied in the same order as the variables were given, unless the value of N is not changed by execution of either of the INPUT statements.

When data input is required, the user is signaled by a "?" character.

The data values that satisfy the variables in INPUT are contained in a list of data separated by commas or blanks. If the list begins with a comma (or in the case of commas with no intervening nonblank characters), the computer

understands that a zero value precedes the comma. For example, the computer interprets

```
,5,3 4
```

as meaning the values 0,5,3, and 4. Similarly, if the list ends with a comma, as in

```
1 2 3,
```

the computer will assign the variables in the INPUT statement the data values 1,2,3, and 0. Finally, the list

```
5 ,1 ,,5 1 2
```

will be interpreted as data values 5,1,0,5,1, and 2.

After the entire list of variables in an INPUT statement is satisfied, control passes to the next program statement. If at the time, the current line of input values has not been exhausted, the remaining values will be accessed by the next INPUT statement executed.

The list of variables in an INPUT statement may include the special entities, asterisk and double asterisk, used to act on unused fields in lines entered for input. The double asterisk means skip any unused fields. The single asterisk means take an error exit if unused fields remain in the line of input.

Examples:

```
200 INPUT A,B,**,C,D
```

means input to A and B, skip anything left in the current input line, and input to C and D from the next line.

```
210 INPUT A,B,*
```

means input to A and B. Error exit if the input line is not exhausted.

If the input lines shown above are entered in response to statement 100, etc., and N = 2 prior to executing statement 100, the result is as follows:

```
X=0

A=5 B=3 Z=4 Y=1 R3=2

B(1,2)=3 C(2)=0 N=5

N=1 B(1,1)=0 C(1)=5
```

The values 1 and 2 are pending for any subsequent INPUT statement.

LOOPING

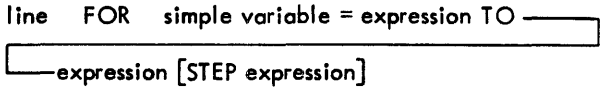
FOR and NEXT BASIC provides the programmer with still another method for specifying data values for variables. This method defines a loop using FOR and NEXT statements. A loop is a portion of a program written in such a way that it will execute repeatedly until some test condition is met. A FOR and NEXT loop causes execution of a set of steps for successive values of a variable until a limiting value would be exceeded. Such values are specified by establishing an initial value for a variable together with a limit value, and an increment or decrement that is used to modify the variable each time the loop is executed. When the limit is exceeded, an exit condition built into the loop allows the computer to proceed to the following body of the program. FOR and NEXT loops, therefore, have three main components.

1. An initial value expression for the variable used by the formula.

2. A limit value expression beyond which the variable may not be incremented (or decremented).

3. An optional increment or decrement expression value to be added to (or subtracted from) the value of the variable for each pass through the loop (except the last).

The FOR statement defines loop parameters. It gives the initial value of the variable, the expression for the limit value that the variable may not exceed and that cause the loop to terminate, and (optionally) the increment or decrement expression. If the step increment or decrement is not expressly given in the FOR statement, it is assumed to be +1. The FOR statement takes the form



The expression preceding TO specifies the initial value of the variable, the expression following TO gives the limiting value, and the expression following STEP gives the increment or decrement. The computer evaluates the initial value expression only once, when the FOR statement is executed. The other two expressions are also evaluated when FOR is executed, but, additionally, are reevaluated every time the NEXT statement is executed. A sample FOR statement is shown below in the discussion of NEXT.

The NEXT statement returns program execution to the beginning of a FOR and NEXT loop after the indexed simple variable has been incremented. NEXT has the form

```
line    NEXT    simple variable
```

Note that the simple variable in the NEXT statement must be specified exactly as it appeared in the FOR statement.

The easiest way to understand a FOR and NEXT loop is to follow one through its entire sequence of operations, as in the following statements.

```
50 FOR X=2 TO 11 STEP 3
60 PRINT X, 2**X
70 NEXT X
```

Statement 50 sets the initial value of X to 2 and specifies that X thereafter will be incremented by 3 each time the loop is performed until X has the limiting value 11.

Statement 60 causes the computer to print out the current value of the variable X and the result of 2^X. Statement 70 causes the computer to return to statement 50, where it picks up the next value of X, that is, +5. The computer then prints 5 and 32 and again goes to NEXT which returns it to FOR. When X attains the limit value of 11, statement 60 will be executed and control will pass to 70. The computer will again try to increment X by 3, but as the upper limit of variable X will have been reached, the computer will "fall through" statement 70 and control will pass to the next statement. At this point, X will have the value 11, the last value that does not exceed the terminal value.

Fractional values may be used in FOR-NEXT loops. When this is done, there is the chance that an expected iteration may not occur because of rounding, as in the following statements:

```
10 FOR I=.1 TO .4 STEP .1
.
.
.
50 NEXT I
```

This loop will be executed only for I = .1, .2, and .3 because the rounded value of I is slightly over .4 on the last try. To get four iterations in this example, use

```
10 FOR I=.1 TO .41 STEP .1
```

Loops may be contained within other loops (nested), but the loops may not "cross". This exclusion is illustrated in Figure 2.

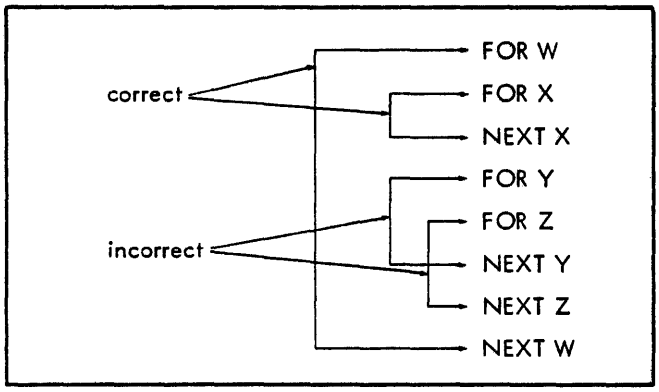


Figure 2. Nested Loops

BASIC allows loop nesting to 26 levels, that is, the BASIC program may contain no more than 26 FOR statements whose corresponding NEXT statements have not yet been encountered in compilation.

MISCELLANEOUS STATEMENTS

REM or * The REM (Remarks) statement allows the programmer to interject commentary anywhere in the program without affecting its execution. REM may be used to identify the complete program, or, more important, the function or purpose of various sections of the program. REM takes the form

line REM [commentary]

The commentary portion of the statement may include any characters up to the end of the line. If commentary is omitted, REM will produce a dummy line in the program.

An alternate form for REM is indicated by an asterisk.

Example:

```
110 *THIS IS A REMARK
```

Branching to a REM statement is allowed and is recommended when branching to a closed subroutine. Such use of a REM statement serves to identify the subroutine. It also allows statements to be inserted at the beginning of the subroutine, if unused line numbers exist between the REM statement and the first executable statement of the subroutine.

PAUSE, STOP, or END PAUSE, STOP, or END can be used to halt program execution at any point. The line number of the halt is printed when program termination occurs.

PAUSE, STOP, and END have the form

line PAUSE

line STOP

line END

Any number of these may be used in a program, or none at all. If none is used, the program will normally halt after the highest numbered line has been executed. If a branch into an infinite loop occurs, as shown in the example below, the BREAK key can be used to halt execution.

```
100 INPUT A
110 PRINT A
120 GOTO 100
```

6. ADVANCED FEATURES OF BASIC

For simplicity, explanations in previous chapters have covered only the essential features of BASIC program elements. This chapter contains additional information on these elements and explains advanced features of BASIC.

OTHER ELEMENTS OF A BASIC PROGRAM

The additional program elements presented below give the user greater flexibility in using the statements explained in Chapter 2, and also augment the capabilities of the new statements described in this chapter.

SUBSCRIPTED VARIABLES

In addition to simple variables, BASIC also provides for subscripted variables. A subscripted variable denotes an element of an array, that is, a list or table of data. The individual values within the array are called array elements. We refer to an array element by specifying the name of the array (always a single letter) and the position of the element in the array. For example, the fourth element in the array named L is denoted by L(4). The value inside the parentheses is called the subscript, and is represented by an expression that can be reduced by the computer to a single integer value. (Subscript expressions are evaluated to integer value after adding 2^{-12}.) Subscripts range from 1 through the maximum allowed dimensioned value.

Arrays can have either one or two dimensions. A one-dimension array is called a vector and is characterized by a single subscript. The subscript denotes the position of the desired array element in the list of data. Sample vector array elements are A(1) and B(J + 3).

When an array has two dimensions, it is called a matrix. Data in a matrix is thought of as being arranged in rows and columns. Each element in a matrix is identified by two subscripts separated from each other by a comma. The first subscript specifies the row number and the second specifies the column number. For example, C(K,L) and D(M+2,N+3) denote matrix array elements.

As a further example of matrix notation, consider the following table, which lists expenses for a four-day car trip.

Row \ Column	1	2	3	4
Item \ Date	June 5	June 6	June 7	June 8
1	Gas, oil	21.29	20.84	19.42	6.08
2	Tolls	1.32	.86	.40	.07
3	Food	11.18	12.83	14.39	5.06
4	Lodging	10.05	12.78	10.35	.00
5	Misc.	1.35	.44	.90	.10

If we consider the table to be a matrix called E, the amount ($10.05) spent for lodging on June 5 would be represented by E(4,1), and the amount ($5.06) spent for food on June 8 would be represented by E(3,4).

DIMENSIONING

A dimension is the largest value that a subscript may attain for a given subscripted variable (array). This limit tells the computer how many storage units of the computer's memory to allocate for the array. Dimensions are specified explicitly in the DIM statement, but the user may make array references without corresponding DIM statements. In such cases, implicit dimensions are used. Implicit dimensions are: 10 storage units for a vector and 100 storage units for a matrix (that is, a 10 by 10 matrix). If the program uses MAT statements (explained later), the dimensions of all arrays referred to in these statements must be explicitly defined in DIM statements.

DIM There are three reasons for explicitly specifying the dimensions of an array.

1. The user may wish to allocate more space for his array than allowed by implicit dimensions. Thus, DIM A(18) would reserve 18 storage units for the vector A.

2. The user may wish to restrict the reserved storage space for each array to its exact dimensions, thereby conserving space. For example, DIM B(3,4) reserves 12 storage units for matrix B, thereby leaving for other use the remaining 88 units that would have been allocated by implicit dimensions.

3. The user may wish to use a given array in a MAT statement.

The DIM statement takes the form

line DIM name(dimx[,dimx])[,name(dimx[,dim])]...

where

name is the name of the array being dimensioned.

dimx is a dimension expression that denotes the maximum number of row or column elements in the array.

Dimension expressions may not contain user-defined functions, array references, or letter digit variables, and are evaluated during compilation (not during execution) by truncating to an integer value after adding 2^{-12}. If dimensions for more than one array are specified in a DIM statement, they are separated by commas. A given array may be dimensioned only once in a BASIC program via a DIM statement. DIM statements may appear anywhere in a BASIC program. A sample DIM statement is given below.

```
10 DIM M(3,3), V(128)
```

VECTORS

Numeric Vectors. A numeric vector is a one-dimensional array containing numeric or aconst data elements. The name of a numeric vector consists of a single alphabetic character. An element is referenced by a subscript expression denoting the relative position of the desired element (see "Subscripted Variables", above).

```
100 A(1)=3.14, B(A(1))='LARGER'
110 A(2)=SIN(A(1)-1), B(2)=1
115 IF A(1)=A(B(A(1)-B(2))+1) THEN 150
120 PRINT B(A(B(2)))
```

String Vectors. A string vector is a one-dimensional array containing text string elements. The name of a string vector consists of an alphabetic character followed by a dollar sign. An element is referenced by a subscript expression denoting the relative position of the desired element. The subscript expression may be followed by a substring expression specifying the beginning and length of the desired substring (see "Character String Manipulation", below).

```
100 DIM A$(3)
110 A$(1)='ABCDEFGH'
120 A$(2)=A$(1:2)
130 A$(3)=A$(2:3,4)
```

In the above example, vector element A$(2) is assigned the string BCDEFGH and A$(3) is assigned DEFG. Note that all string arrays must be dimensioned via a DIM statement.

MATRIXES

Numeric Matrixes. A numeric matrix is a two-dimensional array containing numeric or aconst data elements. The name of a numeric matrix consists of a single alphabetic character. An element is referenced by a pair of subscript expressions, separated by a comma, denoting the row and column of the desired element (see "Subscripted Variables", above):

```
100 A(1,1)=1, A(1,2)=2
110 A(1,3)='THREE', A(2,7)=5
120 IF A(1,1)=A(1,2) THEN 155
130 PRINT A(1,A(2,7)-A(1,2))
```

String Matrixes. A string matrix is a two-dimensional array containing text string data elements. The name of a string matrix consists of an alphabetic character followed by a dollar sign. An element is referenced by a pair of subscript expressions, separated by a comma, denoting the row and column of the desired element. The subscript pair may be followed by a substring expression specifying the beginning and length of the desired substring (see "Character String Manipulation", below):

```
100 DIM A$(2,2)
110 A$(1,1)='ZEITGEIST'
120 A$(1,2)=A$(1,1:3,6)
130 A$(2,1:1,2)=A$(1,2:3)
```

In the above example, A$(1,2) is assigned the string ITGEIS and A$(2,1) is assigned GE. Note that all string arrays must be dimensioned via a DIM statement.

CHARACTER STRING MANIPULATION

BASIC permits strings up to 72 characters long and provides capability for

1. Referencing string variables.
2. Using string expressions.
3. Assigning a character string variable.
4. Assigning length or numeric value of a string variable to a simple or subscripted variable.

5. Converting a numeric value to string format.

6. Concatenating strings.

7. Comparing strings.

8. Using strings in input/output statements.

9. Generating alphanumeric constants from strings for file identification.

REFERENCING STRING VARIABLES

Strings are identified by a letter and dollar sign followed by a further identification of the type of string specified: string scalar, string array, string array element, or substring. Examples of each of these are given at the end of this discussion. Strings may also be combined in expressions for the purpose of string concatenation.

String scalars have the form

letter$ or $letter

A string scalar may not appear in a dimension statement. To avoid conflict, the same letter may not be used for both a string scalar and a string array or numeric array.

String array elements are subscripted variables. They have the form

letter$ (expr[,expr])

where the optional expression denotes a matrix element. A string with only one expression is a vector element.

String arrays may be explicitly dimensioned. The form for dimensioning a string array is

DIM letter$ (dim[,dimx])

Substrings are marked by a colon preceding an expression.

letter$ (:expr 1[,expr 2])

where

expr 1 indicates the position of the first character of the substring.

expr 2 indicates the length of the substring in number of characters. If expression 2 is omitted, the substring includes all characters from the indexed character to the end.

If a string is an element of a vector or a matrix, then the form of the substring is

letter$ (expr 1[,expr 2] : [expr 3][,expr 4])

where

expr 1 and expr 2 are the indexes of a string array element.

expr 3 is the index, or string position, or the first character of the substring.

expr 4 is the length, or number of characters in the substring. Again, if expression 4 is omitted the string consists of all characters from the indexed character to the end of the string.

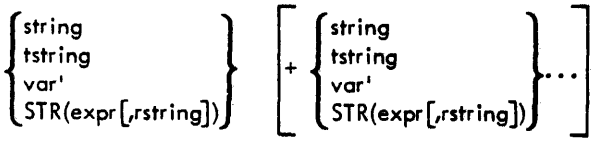
String examples are

P$	String scalar.
H$(1)	String vector element.
B$(2,3)	String matrix element.
A$(:4)	Substring consists of all characters from the fourth to the last character of A$.
A$(:4,1)	Substring consists of fourth character of A$.
B$(2,3:5,2)	Substring consists of the fifth and sixth characters of string matrix element B$(2,3).

STRING EXPRESSIONS

String expressions may be used as arguments for string functions; PUT, PRINT, and PRINTUSING statements; string concatenations; string comparisons; and as file identifiers in OPEN and CHAIN statements. They must be explicitly stated in the PUT, PRINT, and PRINTUSING statements, but may be in either implicit or explicit format in all other cases.

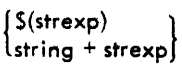
The implicit string expression (strexp) has the form



where var' is a variable containing an alphanumeric constant.

Implicit string expressions are always to the right of the relation operator in string assignment statements and comparisons.

Explicit string expressions (xstrexp) are required to avoid ambiguity on whether or not string processing is called for. The form of this expression is



where the dollar sign resolves the ambiguity that arises if the first character is a letter character (as in STR or in a variable). An implicit expression may be used only in a statement where the syntax is unambiguous in indicating string processing.

Examples:

LET Z$ = B (2,3) + A$(:5)	implicit string expression
PRINT $(B(2,3) + A$(:5))	explicit string expression
PRINT B (2,3) + A$(:5)	ILLEGAL ! ambiguous syntax
PRINT "ABC" + A$	ILLEGAL

ASSIGNING CHARACTER STRINGS TO STRING VARIABLES

Simple variables provide storage for just one doubleword; therefore, a simple variable is limited to representing an alphanumeric constant (maximum of six characters).

Character strings more than six characters long must be assigned to string variables (letter$). Strings up to six characters are considered alphanumeric constants and may be assigned to simple or subscripted variables.

STRING LENGTH AND VALUE ASSIGNMENTS

For these assignments, BASIC provides two intrinsic functions: LEN (for length) and VAL (for value). LEN and VAL may only be used in assignment statements. The assignments are made to simple or subscripted (not string) variables and have the form

line LET var[,var]... =sfunct(strexp)

where sfunct is LEN or VAL. Both assignments can be made in one statement, separated by a comma as in the example

```
25 K1=LEN(W$(2,3)), K2=VAL(W$(2,3))
```

in which the length of the matrix string element WS(2,3) is assigned to the simple variable K1, and its numerical value to K2. The arguments for both functions must be string expressions. If the character string specified for VAL does not represent a correctly formatted decimal constant, an error message is generated and execution terminates.

CONVERSION TO A STRING

The output conversion routine automatically converts an expression to string fromat, but in manipulating text it may be desirable to have the same conversion performed internally, for example to store an evaluated expression in a file or embedded as a substring within a text string. The string-conversion routine is available for this purpose. It has the form

[line] LET string = STR (expr[,rstring])

where STR is the string-conversion function.

The replaceable-string (rstring) argument is optionally used to indicate the image of the desired format. If the rstring option is not used, format is that for print output conversion.

Like the output conversion, string-conversion is governed by the setting of the precision flag. The string will have a leading blank if it is nonnegative, but will not contain trailing blanks. The minimum length for a string is two bytes; maximum length is 22 bytes for long precision and 12 bytes for short precision.

Examples of STR (conversion-to-string) statements are

```
10 A$=STR(3.5,#.#)
20 H$(1:9)=STR(SQR(X))
30 LET W$(2,3)=STR(A1+B1*COS(X))
```

STRING ASSIGNMENT AND CONCATENATION

Another string, an alphanumeric constant, a string converted expression (see above), or a concatenation of any or all of

these may be assigned to a string. The form of the string-assignment and concatenation statement is

$$\text{line } [\text{LET}]\text{string} = \begin{Bmatrix}\text{strexp}\\ \text{xstrexp}\end{Bmatrix} \left[+\begin{Bmatrix}\text{strexp}\\ \text{xstrexp}\end{Bmatrix}\right]\ldots$$

where strexp is an implicit string expression and xstrexp an explicit string expression (see "String Expressions", above).

Examples:

```
100 DIM A$(2)
110 A$(1)='ONE', A$(2)='TWO'
120 B$=A$(1)+' AND '+A$(2)
130 B$=B$+'.'
```

The left string is given a value and a length consistent with the items to the right of the equals sign. If the right contains only one term, the statement performs string assignment. If the right side contains two or more terms, concatenation occurs in the order given. If the maximum string length is exceeded, the string is truncated.

If assignment is to a substring whose current length is less than n-1, where n is the first character of the target substring, then the gap to character n-1 is filled with blanks. If target-substring length is specified and the number of characters transferred is less than this specified length, then the gap from the last character transferred to the specified length is also filled with blanks. Characters in excess of specified length are not transferred.

STRING COMPARISON

Strings are compared for identity or "magnitude" in IF ... THEN or GOTO statements. The form of the statement is

$$\text{line IF string oper} \begin{Bmatrix}\text{strexp}\\ \text{xstrexp}\end{Bmatrix} \begin{Bmatrix}\text{THEN}\\ \text{GOTO}\end{Bmatrix} \text{line}$$

where oper is a condition operator (stexp and xstrexp were explained under "String Expressions"). Examples are

```
10 IF W$(:1)='1W' GOTO 99
20 IF W$=STR(X1*Y1+3) THEN 40
30 IF R$=STR(0) GOTO 85
40 IF A$='DOG'+B$(1:9) THEN 120
```

Strings are compared from left to right as character pairs. In comparing characters, the EBCDIC collating sequence is followed. A blank is the lowest character, followed by nonalphanumeric characters. Alphabetic letters are next, in the order A B C ... YZ. Digits are the highest elements in the collating sequence. If one string is shorter than another, the shorter string is "extended" with blanks for comparison.

STRING INPUT/OUTPUT

Explicit string expressions and text strings may be used in input/output statements in the form used for expressions and alphanumeric constants, subject to the general rules governing strings. That is, a run-time error results if a text string (more than six characters) is provided as input to a nonstring variable; or nonstring input (neither a text string nor an alphanumeric constant) is provided to a string. The statements used are INPUT, PUT, READ, GET, DATA, PRINT, MAT PUT, MAT GET, MAT READ, PRINTUSING, and MAT INPUT.

Examples:

```
140 MAT GET :3, A$(2,3), B$
500 DATA 'ONCE UPON', "A TIME"
885 PUT V$, W$(1), X$(1,A)+'?'
910 READ H$(1), H$(2), H$(3)
700 GET :3;K,W$(A1,A2), X$
750 PRINT $("IT'S "+STR(A)+B$)
760 PRINT USING 100, H$(1), C$
800 PRINT :4,A$(1,1),A$(1,2)
400 MAT PUT A$, B$, C$
410 MAT GET A$(2,3), V$
420 MAT READ F$(4), G$
430 MAT INPUT W$, X$
440 MAT GET A$, B
445 PRINT :1,B$(2,3:4,5);A**2
```

Line 440 requires that the data file have the correct number of string array elements to fill A$, immediately followed by numeric data to fill numeric array B.

The examples above show cases of string input/output only. The forms are similar to those described earlier for nonstring input/output. In BASIC, statements may mix (with appropriate caution) string and nonstring items in the same statement, as shown in Appendix A.

STRING INPUT MODE CONTROL

Normally, when a string is reached in the list for an INPUT statement, the next data field in the record is accessed. Blanks and commas are treated as field separators unless they occur within quoted fields. An alternate form is provided in which an entire input line, or record, is treated as a single field.

The form for switching string INPUT mode is

$$\text{line INPUT} = \begin{Bmatrix}\$\\ \text{any other character}\end{Bmatrix}$$

INPUT = $ switches to full record input mode. Each input referenced to a string accesses a full record and treats it as a single string (as though it were enclosed in quotes). If a record has been partially input (for numeric assignment) and a reference to a string follows, the remainder of the record is treated as a single field.

INPUT = X (any character but $) switches back to normal input mode, which is the default.

String input mode is changed only by these explicit statements and remains as set through successive operations within BASIC until explicitly reset.

GENERATION OF ACONSTS FROM STRINGS

A string or string expression may be assigned to a simple or subscripted variable, but only six characters will be transferred and the rest truncated. If the string contains fewer that six characters, trailing nulls are generated to satisfy the aconst format.

Examples:

```
10 A1=P$
20 A2=B$(;4)
30 A3=$('NO. '+C$(A4))
```

This provides an indirect means to assign strings as external names or file identifiers by first assigning strings to simple variables.

STRING EXPRESSIONS AS FILE IDENTIFIERS

In BASIC, string expressions may be used to designate the name, password, and account for file identification in OPEN and CHAIN statements. The string expressions must not result in text strings exceeding 11 characters for name or 8 characters for account and password.

Examples:

```
120 OPEN 'FILE'+A$ TO :1,INPUT
340 CHAIN B$(N):'ABC';'SECRET'
```

In line 120 above, if A$ = '1234567' and I = 3 then 'FILE3' is opened.

USER-DEFINED FUNCTIONS

DEF If the programmer wants to make use of a function that is not included in the set of BASIC intrinsic functions, or if he intends to make repeated use of an involved expression, he may define the function in a DEF statement and make reference to it according to a name he designates. The form of the DEF statement is

line DEF FN letter(simple variable[,simple variable]...) = expression

where

letter provides a unique name for the function.

simple variable is a dummy argument appearing in parentheses to the left of the equals sign. These only serve to identify which of the simple variables in the expression to the right of the equal sign are arguments. These must be at least one such argument, although it is not necessary that any or all of the arguments appear in the expression. Each time the function is evaluated, current argument values will be substituted for these terms in the expression. There is no comma following the final simple variable in the list.

The following examples illustrate typical DEF formats:

```
65 DEF FNA(X)=X+B*X
100 DEF FNB(X)=X*SIN(FNA(X+C))
120 DEF FNX(X0,X1,X2)=X0*X1*X2/K
550 X=FNX(1,2.3)+FNB(Y+3.14)
```

Line 500 is an example of how the functions defined in lines 100 and 120 might be used later in the program. The variable X to the left of the equals sign is a different entity from the dummy variables X in the DEF statements.

DEF statements may appear anywhere in the BASIC program, including those cases in which the function is referenced prior to its definition.

BASIC checks DEF statements for identical simple variables in the list of dummy arguments, undefined functions, multi-defined functions, and consistency between the number of arguments supplied by the programmer when the function is called (referred to) and the number of arguments in the DEF statement. However, it is the responsibility of the programmer to avoid circular definitions in and among the DEF statements. Improper uses of DEF are shown below.

Case 1. Circular definition within statement:

```
1200 DEF FNA(X)=X+FNA(X)
```

Case 2. Circular definition among statements:

```
1400 DEF FNA(X)=X+FNB(X)
1450 DEF FNB(X)=X*FNC(X)
1500 DEF FNC(X)=FNA(X)/X
```

REREADING DATA

RESTORE The RESTORE statement alters the normal sequence of DATA statement accession. DATA statements are normally accessed as the preceding DATA statement is exhausted. For example, of the following set of DATA statements,

```
100 DATA 1,2,3,4
110 DATA 5,6,7,8
120 DATA 9,10,11,12
```

statement 110 will be accessed only after data value 4 in statement 100 has been assigned to a variable, and statement 120 will be accessed after data value 8 in the preceding statement is assigned. RESTORE allows the programmer to alter this sequence by directing the computer (via a line number) to a specified DATA statement from which data accession will proceed in the normal manner.

The RESTORE statement is frequently used for accessing data that will be used several times in the program, and eliminates the need for writing duplicate DATA statements when the same data is to be accessed more than once. The form of the RESTORE statement is

line RESTORE [line]

where the second "line" must be the line number of a valid DATA statement in the program. Some sample RESTORE statements are given below.

```
740 RESTORE 125
900 RESTORE
```

If the line number is omitted in the RESTORE statement (as in line 90 above), the computer will return to the first DATA statement in the program.

BRANCHING TO A SUBROUTINE

GOSUB and RETURN The GOSUB and RETURN statements provide subroutine capability in BASIC. A subroutine is a section of the main program that completes a specific task. GOSUB, in the main body of the program, directs the computer (via a line number) to the first statement of the subroutine. After the subroutine has been executed, RETURN directs the computer to the statement following GOSUB, where the main program continues. The form of GOSUB and RETURN are

line GOSUB line

line RETURN

where RETURN is the last executed statement of the subroutine.

Some sample GOSUB and RETURN statements are shown below.

```
10 GOSUB 500
 .
 .
 .
525 RETURN
```

The RETURN statement does not contain the line number of the statement following GOSUB. BASIC remembers its place in the program.

An attempt to execute a RETURN statement before a GOSUB statement is executed causes output of an appropriate error message. Execution of too many GOSUBs before a RETURN also causes an error message to be printed. The program may execute up to 20 GOSUB statements before a RETURN is needed.

CHARACTER CONVERSION

CHANGE The CHANGE command can be used to convert string characters to equivalent EBCDIC values and vice versa. To convert a string to EBCDIC, the command has the following form:

line CHANGE {string / xstrexp} TO letter

Examples:

```
10 CHANGE A$ TO B
50 CHANGE $('246'+C$) TO D
```

The string characters are converted to EBCDIC values stored in the vector specified by the letter. The letter must represent a vector dimensioned by a DIM statement. The current dimension of the vector is set to the number of string characters converted.

Assuming that A$ = 'A1' when line 10 above is executed, the decimal equivalent of 'A' (i.e., 193) is stored in B(1) and the equivalent of '1' (i.e., 241) is stored in B(2).

To convert a vector to a string, the following form is used:

line CHANGE letter TO string

The elements of the specified vector are converted to characters and placed in the specified string or substring. The current dimension of the vector is used. If a value has a fractional part, it is truncated.

Example:

```
15 CHANGE X TO Y$
```

Assuming X has the elements 193, 241, 90, and 87, the characters 'A', '1', '!', and 'bell' will be stored in Y$.

FILE MANIPULATION

A file is a collection of items assigned a name and treated as a single unit. It is the principal device for manipulation of blocks of data too large to process as a unit in computer memory and for storage and data. Files are made up of records, each of which may contain one or more data elements. A file may be organized as a consecutive sequence of records or as a set of records arranged according to sort keys. The data in files may exist in a "print" form (BCD) or in the form used within the computer (binary). BASIC allows operations on BCD and binary files with sequential or keyed access. Files are also used to store and fetch BASIC programs. The BASIC statements OPEN, CLOSE, GET, PUT, ENDFILE, and special forms of PRINT and INPUT provide file manipulating capability.

FILE NOMENCLATURE

Files may have names of up to 7 characters. Names may be enclosed in quotes (single or double) or may be aconsts (limited to six characters) stored in simple variables. In BASIC a name may also be any string expression of up to 7 characters.

The name may be optionally followed by a password of up to four character and/or an account identifier of up to eight characters. An account should not be specified for output operations, since output will not be permitted on other users' accounts.

The general form for file identification in an OPEN statement is

name [;password][:account]

Examples:

```
"FILEONE"
A1;'PASS':'12345678'
```

In the second example, the simple variable A1 must contain a name as an aconst.

Passwords are used for file security, and the account is used to input data created in other users' accounts. A password is preceded by a semicolon and an account number is preceded by a colon. If both account and password are specified, the order of their appearance is optional.

The term 'fileid' will be used for name [;password][:account] in describing forms of the OPEN and CHAIN statements.

I/O STREAM NUMBERS

Files are opened to specified input/output stream numbers. BASIC permits four streams. Only one file can be opened on a given stream number at one time, but the same stream number may be used to open another file later, closing the currently open file. The stream number is specified in the OPEN statement and may be any expression which evaluates to a legitimate stream number. Fractional values are truncated to integers, and expressions which do not result in integer values should generally be avoided.

KEYED AND SEQUENTIAL ACCESS

BASIC allows file access in either sequential or keyed form. All files created by BASIC are actually keyed but they may be sequentially written and read without explicit references to keys. Sequential files created without keys may only be read sequentially. This means that files created by EASY can only be read by BASIC, not updated.

The keys used in BASIC are numbers in the range 0.001 to 9999.999. Sequential files are created with keys 1.000, 2.000,... (these keys are compatible with the keys, or "sequence numbers", used in the Xerox EDIT processor).

The key value for a given I/O operation can be set by explicit reference to the key, using any arithmetic expression. If a subsequent operation on the same file does not reference an explicit key, the key value is incremented by one for each record accessed.

If an output statement with an explicit key creates more than one record, the subsequent records have keys incremented by one per record. If an input statement with an explicit key requires more than one record of data, records are read sequentially starting at the record with the referenced key.

UNKEYED I/O IN THE UPDATE MODE

When a stream is opened in the update mode, file positioning is not separately maintained for input and output operations. For example, if a PUT is followed by a GET and the GET is unkeyed, unused data read from the last record is used. Then the record accessed is the next one, in ascending key sequence, after the last record PUT. An unkeyed PUT following a GET replaces the last record accessed. An unkeyed PRINT following an INPUT replaces the last record accessed. An unkeyed INPUT following a PRINT accesses the next record after exhausting all data in a previously INPUT record.

In general, it is advisable to specify keys in all UPDATE operations.

OPEN The OPEN statement performs the following file management functions:

1. Designates that the named file is to be opened for BCD or binary input, output, or update.
2. Assigns the file to an I/O stream number.
3. Closes a file if the file is to be opened for output and a file of the same name is currently open. Closes a file if the file is to be opened for input and a file of the same name is currently open for output.
4. Indicates whether an existing file may be written over if a file of the same name is to be open for output.
5. Positions the opened file at its starting point. (A file opened for output is initialized as an empty file.)
6. Declares a file as a TFILE if the OPEN statement so indicates. TFILES are released at the end of a terminal session or, in batch operations, at the end of the JOB. The TFILE directive is ignored if the file has a password.

BINARY INPUT

The OPEN statement for binary input has the form

line OPEN fileid[,]TO :stream, GET[[,]TFILE]

Example:

```
120 OPEN 'DATA' TO :3, GET
```

This opens the file on I/O stream 3 and does not declare the file temporary.

DEFAULT FORM FOR BINARY INPUT

An abbreviated form may be used for binary input.

line OPEN fileid[,]I[any characters]

This is equivalent to

line OPEN fileid TO :1, GET

BCD INPUT

The OPEN statement for BCD input has the form

line OPEN fileid[,]TO :stream, INPUT[[,] TFILE]

BINARY OUTPUT

The OPEN statement for binary output has the form

line OPEN fileid[,]TO :stream, PUT,{ON / OVER}[[,]TFILE]

Example:

```
130 OPEN 'OUTF' TO :A(I),PUT,OVER,TFILE
```

If A(I)=4, this opens "OUTF" to stream 4 for binary output (PUT). OVER indicates that an old file named OUTF is to be written over if present. TFILE indicates this is a temporary file, to be released at end of job.

DEFAULT FORM FOR BINARY OUTPUT

An abbreviated form may be used for binary output.

line OPEN fileid[,]O[any characters]

This is equivalent to

line OPEN fileid TO :2,PUT,OVER,TFILE

BCD OUTPUT

The OPEN statement for BCD output has the f n

line OPEN fileid[,]TO :stream, PRINT[,]{ON / OVER}
[[,]TFILE]

Example:

```
140 OPEN 'BCDOUT' TO :4, PRINT ON
```

The 'ON' directive means if an old file exists with name BCDOUT, it is not to be overwitten.

BINARY FILE UPDATE

To update a binary file, use the form

line OPEN fileid[,]TO :stream, GET, UPDATE
[[,]TFILE]

This opens an existing binary file in the update mode, allowing input (GET) and output (PUT) on the file.

BCD FILE UPDATE

To update a BCD file, use the form

line OPEN fileid[,]TO :stream, INPUT,UPDATE[[,]TFILE]

This opens an existing BCD file in the update mode, allowing input (INPUT) and output (PRINT) on the file.

ENDFILE The ENDFILE statement allows the user to branch to a designated line number in his program when an out-of-data condition occurs or a specified key is not found. The form of the ENDFILE statement is

line ENDFILE :stream,{E / line number}

The "stream" may be any expression. If the expression evaluates to a legitimate stream number, the OUT OF DATA control will be applied to any GET or INPUT: via that stream. If the expression evaluates to zero, OUT OF DATA control is applied for READ statements.

The "stream" expression is followed by a "line number" (not an expression) or the letter "E". E indicates reset to normal error exit. A "line number" indicates the location (in the user's program) to transfer to on the out-of-data condition.

Example:

```
100 OPEN "FILE" TO :1, INPUT
110 ENDFILE :1, 150
120 INPUT :1, A$
130 PRINT A$
140 GOTO 120
150 CLOSE :1
```

CLOSE The CLOSE statement closes the file on the indicated I/O stream.

line CLOSE :stream

This closes the open file, if any, on the indicated stream number.

Example:

```
200 CLOSE :N
```

If N=3, stream 3 is closed.

The following forms may also be used.

line CLOSE I[any characters](equivalent to CLOSE :1)

line CLOSE O[any characters](equivalent to CLOSE :2)

Files may also be implicitly closed by an OPEN statement and are closed when leaving BASIC.

GET The GET statement retrieves binary data from files created by PUT statements. Data is assigned to specified variables as it is received from the file via the indicated I/O stream. Access may be sequential or keyed. GET has the form

line GET[:stream[;key],]variable[,variable]...

A default form can be used

GET variable...

This is equivalent to

GET :1,variable...

The variables may be simple or subscripted. There is no comma following the final variable in the list. As in READ and INPUT, the variable list may include either of the special entities * or **. The single asterisk causes an error exit if the current record has not been exhausted of data. The double asterisk causes any unused data in the current record to be discarded.

Attempts to GET on an I/O stream which is not open for binary input or to use an illegal I/O stream number will terminate the run with an appropriate error message.

The specification of a nonexistent key or an attempt to read beyond end-of-file gives an OUT OF DATA error exit and message. This exit may be modified by the user (see "ENDFILE", above).

A GET statement may require reading more than one record to satisfy the variable list. If the GET statement has the key value n, the records read are accessed sequentially starting with n.

If a keyed GET is followed by a nonkeyed GET, records are accessed sequentially starting with the keyed record.

Example:

```
100 OPEN "PUTFILE" TO :3, GET
110 ENDFILE :3, 150
120 GET :3, A1, B4, C7
130 PRINT A1*(B4-C7)
140 GOTO 120
150 CLOSE :3
```

PUT The form of the PUT statement is

line PUT[:stream[;key],]{expr / aconst}[,{expr / aconst}]...

This statement writes data into a file in internal (binary) format. The expression following the colon indicates the I/O stream (thus, because of an OPEN statement, the file to be written on). The optional expression preceded by a semicolon designates the key. If a key is designated,

the first items written will be to a record with that key value. A PUT statement may generate more than one record. If so, and if the statement contains the explicit key n, the records generated will have keys of value n, n+1,...

A default form can be used

line PUT{expr / aconst}[,{expr / aconst}]...

This is equivalent to

line PUT :2,{expr / aconst}[,{expr / aconst}]...

The data values to be entered into the file may take the form of an expression or an alphanumeric constant. There is no comma following the final expression or aconst in the list. Some sample PUT statements are shown below.

```
880 PUT "ERICEL",55,72
881 PUT :1,TIM(X),DAY(X),YER(X)
882 PUT FNH(A1)+P*Q
```

The FNH in statement 882 is an example of a user-defined function. These are explained under "User-Defined Functions", earlier in this chapter.

PUT statements can be used to write on files opened for PUT or opened for GET,UPDATE. In the case of updates, records can be inserted or replaced using the key option. In newly created files, the key option can also be used to write records in a nonsequential order and to replace previously written records by repeating a PUT with the same key.

The form of PUT and GET records is described in Appendix E. These records generally include 14 data values, but short records may be created by use of keyed PUT statements or as the last record written before closing the file or using the flushing technique (see "I/O Flushing" below).

Example of keyed PUT statement:

```
300 PUT :4;121,A,B,C,D,E,F,
    G,H,I,J,K,L,M,N,O,P
```

This statement will cause BASIC to write a record via I/O stream number 4 with the key 121 and the values contained in the simple variables A to N. The values in O and P will be the first two values written on record 122. If the next PUT statement does not include a key, writing will continue on record 122. (This example was chosen to indicate the caution that should be used in mixing keyed and nonkeyed output statements.)

If a PUT statement does not fill the current output record, that record is not normally output until it is later filled, a keyed PUT is executed, or a CLOSE is executed. Short records may be forced out by using the special expression "**" (see "I/O Flushing", below).

A PUT statement may result in an error message if the selected I/O stream is not open in the proper mode, an illegal stream number is selected, or an out-of-range key is selected.

INPUT The action of an INPUT statement for file input is analogous to that of a normal INPUT statement (see "INPUT", Chapter 2) except the BCD input is routed from an open file through a specified I/O stream rather than from the terminal. Sequential or keyed access is permitted. The form of a file INPUT statement is

line INPUT :stream ;key , input list

Example:

```
250 INPUT :3,A(1),A(2),**
```

The "input list" is the same as for normal INPUT statements. One line constitutes one record and a single INPUT statement may access several records sequentially. If the INPUT statement specifies a key, reading starts at the beginning of the keyed record.

Attempts to input a specific keyed record that does not exist, (if no ENDFILE is in effect) or to use an I/O stream not open for BCD input or an illegal I/O stream will result in an error exit and message.

An attempt to INPUT beyond the end-of-file gives an OUT OF DATA error exit. This exit may be modified by the user by means of an ENDFILE statement (see above).

PRINT The action of the PRINT statement for file output is analogous to that of a normal PRINT statement (see "PRINT", Chapter 2) except the BCD output is routed to an open file via a specified I/O stream. Sequential or keyed output is permitted. The form of the file PRINT statement is

line PRINT :stream[;key],print list

The first expression is the I/O "stream" number. The optional second expression is the "key" value. The "print list" allows any arguments acceptable in a normal PRINT statement.

A record is generated for each line of print, thus one statement may generate more than one record. If the print list ends with punctuation, a partial record is formed. The record is output if a full line is formed, a PRINT on the same stream ends without punctuation, or any PRINT is executed on a different I/O stream (or to the terminal). In general, it is bad practice to end a file PRINT statement with punctuation (comma or semicolon) or a TAB(0).

If a PRINT statement generates more than one record, the key is incremented by 1 for each record. This should be particularly noted if files are generated with later updates in mind or when in the update mode. If it is likely that multirecord PRINT statements will be used, the file should be created as a keyed file with key increments large enough to allow insertions and replacements without inadvertent overwrites.

Example:

```
400 PRINT :4;I,1,2,3,4,5,6,7,8
410 PRINT :4;I+1,9,10,11,12
```

In the above example, statement 400 generates two records with keys I and I+1. Then statement 410 generates a new record with key I+1.

If the file is created with a key interval of 10 records, it might be generated or updated as follows.

```
400 PRINT :4;10*I,1,2,3,4,5,6,7,8
410 PRINT :4;10*(I+1),9,10,11,12
```

In this case, statement 400 generates records with keys 10I and 10I+1. Statement 410 generates a record with key 10I+10.

PRINTUSING The action of the PRINTUSING statement for file output is analogous to that of a normal PRINTUSING statement (see "PRINTUSING and :(Image)", Chapter 2) except the BCD output is routed to an open file via a specified I/O stream. Sequential or keyed output is permitted. The form of the file PRINTUSING statement is

[line]PRINT :stream[;key],USING line[,{expression / text string}]...

Example:

```
100 PRINT :1;K, USING 200, X, Y4
```

I/O RESIDUE

In executing GET, INPUT, or READ statements the current record may or may not be exhausted of data, depending on

```
!EASY

NEW OR OLD--OLD LISTF

READY
LIST FILE


FILE      UTS/EASY      14:21 AUG 25,'72

"EMPL NO.      NAME          SOC SEC NO.   ADDRESS       PHONE"
10712    JACK 468-54-234 123GTR 876-0987
76540    MIKE 654-87-932 321KIU 654-6543
87654    TED 432-22-876 987PIP 876-6789

READY
LIST



LISTF      UTS/EASY      14:21 AUG 25,'72

10 DIM E$(5)
20 OPEN "FILE" TO :1, INPUT
30 INPUT N$
40 INPUT :1, B$
50 INPUT :1, E$(1)
60 IF E$(1)=N$ THEN 90
70 INPUT :1, **
80 GOTO 50
90 PRINT B$
100 INPUT :1,E$(2),E$(3),E$(4),E$(5)
110 PRINT E$(1),E$(2),E$(3),E$(4),E$(5)

READY
RUNNH

?76540
EMPL NO.       NAME          SOC SEC NO.   ADDRESS       PHONE
76540          MIKE          654-87-932    321KIU        654-6543

     110 HALT
```

Figure 3. INPUT Residue Example

the amount of data contained in that record. Normally, any residual data remaining in a record is retained for use by a subsequent GET, INPUT, or READ. The use of a single asterisk in a GET, INPUT, or READ causes BASIC to take an error exit if residue occurs. A double asterisk causes any residue to be discarded.

In the example shown in Figure 3, the user loads, lists, and runs a program named LISTFILE. This program opens FILE to stream 1 and prompts the user to type an employee number. It then inputs a header record to B$ and inputs the first field of each subsequent record to E$(1) until it finds one beginning with 76540. The INPUT :1, ** statement in line 70 causes unwanted residue to be discarded. Line 100 inputs the remaining data from the selected record.

I/O FLUSHING

If a PUT statement does not fill the current output record, that record is not normally output until it is filled by a subsequent PUT, a keyed PUT is executed, or a CLOSE is executed.

The writing of short records can be forced by use of the double asterisk (see Figure 8, Appendix E). This capability is useful primarily in the update mode.

Examples:

```
320 PUT :4;N,A(1),A(2),A(3),**
```

This ensures that a three-element record with a key value of 4 is immediately output.

```
440 PUT :1,**,A,B,C,**,D,E,F,**
```

This ensures that two short records are output, with three elements each. In this case, if a partial record was pending prior to executing the statement, it is output first as a short record.

RUNNING CONSECUTIVE PROGRAMS

CHAIN CHAIN directs the computer to acquire and run another program of a series of programs without future action by the programmer. The format of the CHAIN statement is

line CHAIN name[;password][:acct no]

where name (and, optionally, password and acct no) is the identity of a program as defined in the discussion of the OPEN statement.

When executed, CHAIN produces the following results:

1. The current program is discarded, but the values of its simple variables are retained.

2. The named program is obtained, compiled, and executed if possible.

Note that only the simple variables computed by the program are retained when CHAIN is executed; all array values and dimension information are lost. A sample CHAIN statement is shown below.

```
950 CHAIN "PART2" :A;P
```

CHAIN LINK The CHAIN LINK statement differs from the CHAIN statement in that it preserves array and string values. The CHAIN statement retains only the simple variables and discards the rest of the program.

The form of the statement is

line CHAIN [LINK]file identification

An example is

```
960 CHAIN LINK "PART3" :A;P
```

where

"PART3" is the name of the chained program.

:A is account A.

;P is the password.

A and P contain alphanumeric constants. The "LINK" part of the statement is optional. Without it, the array context of the old program would simply be discarded. With it, context is retained.

Although CHAIN LINK preserves array context, array dimensioning is not preserved. Therefore any array used in a program that is chained to must be redimensioned in that program, via a DIM statement, if explicitly dimensioned in the first program.

MATRIX OPERATIONS

Matrix operations in BASIC are controlled through use of a special set of MAT statements. In addition to the usual set of allowed matrix manipulations, BASIC provides options for input of matrixes via console or file, copying of matrixes, and the solution of simultaneous equations. Some of the

matrix operations apply to vectors as well as to matrixes. At times, vectors are treated as either row or column matrixes.

MAT statements may be specified to use variable dimensions, as long as these are within the dimension limits specified in DIM statements. In some cases, dimensions will vary because of the operations that are performed on them (for example, multiplication of nonsquare matrixes). Thus, there is current dimensioning (the result of the latest matrix operations, or as specified by the user in MAT SIZE statements), and absolute dimensioning (given in DIM statements).

Every array that is named in a MAT statement must be dimensioned in a DIM statement. This assures that absolute dimensions exist, sets initial current dimensions, and differentiates between vectors and matrixes. (Note that use of a letter to designate an array does not preclude the use of the same letter to designate a simple variable.) Current dimensioning may not exceed absolute dimensioning. Matrix operations and their corresponding MAT statements are presented below.

MAT GET The MAT GET statement reads array values from the currently open input file. It complements the MAT PUT statement, but can also read data prepared by ordinary PUT statements. MAT GET takes the form

line MAT GET[:stream[;key],]adescr[,adescr]...

The default form

MAT GET adescr[,adescr]...

is equivalent to

MAT GET :1,adescr[,adescr]...

where adescr is an array descriptor of the following format:

aname[(dimx[,dimx])]

In the array descriptor, the terms aname and dimx present the single letter array designator and dimension (subscript) expression, respectively, of the array. As shown in the general form above, therefore, an adescr term may indicate a single letter array designator, a designator followed by one subscript (a vector), or a designator followed by two subscripts (a matrix). The dimx expressions in an adescr term constitute variable dimensions; they provide a simple method for varying current dimensions during execution (not during compilation).

One example of a MAT GET statement is

```
1008 MAT GET A(3,3), B(4,8)
```

MAT PUT The MAT PUT statement enters arrays into the currently open output file. It takes the form

line MAT PUT[:stream[;key],]adescr[,adescr]...

The default form

MAT PUT adescr[,adescr]...

is equivalent to

MAT PUT :2,adescr[,adescr]...

There is no comma following the final array name in the list. In the sample statement

```
1002 MAT PUT A,B
```

array A will be completely output before array B is output. Current dimensions determine how much data is output from a given array. Current dimensions may be set as part of the adescr, which is described above for MAT GET. Matrixes are entered into output files in row major sequence, that is, with the last subscript varying most rapidly.

Note that one MAT PUT may create several records. The key for the first record may be selected, but later records will have keys incremented by 1 per record. See "I/O Flushing", above, for treatment of partial records.

MAT INPUT The MAT INPUT statement is the array counterpart of the variable-oriented INPUT statement described earlier in this chapter. Format and a sample statement are shown below.

line MAT INPUT[:stream[;key],]adescr[,adescr]...

If ":stream" is included, input is from an open BCD input file assigned to that stream.

Example:

```
1007 MAT INPUT A(3,4), B
```

When statement 1007 is executed, 12 values must be supplied for the 3 x 4 matrix A. These values are then followed by input for array B. Note that the number of values input to array B must match the current dimensions for B.

The rule for the use of commas and empty fields in the data list read by MAT INPUT is the same as described for the INPUT statement.

MAT PRINT The MAT PRINT statement prints arrays in regular or packed format. The form of the MAT PRINT statement is

line MAT PRINT[:stream[;key],]aname[{,;}]
[aname]...[{,;}]

where aname is the letter designation of an array that has been dimensioned in a DIM statement. Some sample MAT PRINT statements are shown below.

```
1000 MAT PRINT A, B; C
1010 MAT PRINT D;
```

There are two types of print formats, regular, or packed. An aname parameter followed by a semicolon causes the named array to be printed in packed format. Otherwise, regular format is used. Statement 1000 will cause array A to be completely printed before any element of array B is printed. Array A will be printed in regular format, array B in packed format, and array C in regular format. Current dimensions are used to determine how much data is printed from an array. Statement 1001 will cause array D to be printed out in packed format.

Vectors are printed in row fashion. Each row of a matrix is printed as one or more consecutive print-rows with a blank line between successive matrix-rows. Column 1 of a matrix always occurs in the leftmost print field.

Note: If MAT PRINT ;stream is used, each line of output creates one record. In general, more than one record will be created. A record containing one blank byte is created after the last MAT PRINT record is written.

MAT READ The MAT READ statement is similar to the READ statement described in Chapter 2, except that it acquires whole arrays of data, rather than just single data items. MAT READ has the form

line MAT READ adescr[,adescr]...

where adescr is an array descriptor of the same format previously explained under "GET", namely

aname [(dimx[,dimx])]

If dimension expressions are included in the array descriptor, they specify current array dimensions. If they are omitted, current dimensioning results from previous conditions. Some samples are shown below.

```
1005 MAT READ A(K/L+M,7)
1006 MAT READ B, C(3,4)
1007 MAT READ D
```

MAT SIZE The MAT SIZE statement redefines current dimensions of the named array. MAT SIZE has the form

line MAT SIZE aname (dimx[,dimx])
[,aname(dimx[,dimx])]...

Examples:

```
1003 MAT SIZE A(X+Y,Z)
1004 MAT SIZE D(4,5), B(3)
```

where

aname is the single letter designator of an array.

dimx is any expression representing a legal subscript.

ASSIGNMENT FUNCTIONS

The matrix assignment functions resemble the LET statement in form and function. An array name (aname) always appears immediately to the left of the equals sign and the named array is assigned values according to the specifications to the right of the equals sign.

ZERO

This statement zeros those elements of the named array that fall within the range of current dimensions. Its format is

line MAT aname = ZER[(dimx[,dimx])]

The optional dimx terms have the same meaning described previously under MAT GET; that is, they are dimension expressions specifying new current dimensions (subscripts) of the array, and as such constitute variable dimensions. For example, assuming the array B(5,5), that is, a 5 x 5 matrix named B, the following statement,

```
1010 MAT B=ZER(3,2)
```

will zero elements (1,1),(1,2),(2,1),(2,2),(3,1), and (3,2), leaving the remainder of the matrix unchanged. Other examples are

```
1020 MAT C=ZER(10)
1030 MAT D=ZER
```

CONSTANT

This statement is analogous to that discussed above, except that is sets matrix elements to 1 (instead of 0). Its format is given below.

line MAT aname = CON[(dimx[,dimx])]

Example:

```
2250 MAT L=CON(3,5)
```

IDENTITY MATRIX

This statement forms an identity matrix. Since the array must be a square matrix, new current dimensioning may have to be provided. Two forms are shown below.

line MAT ananme = IDN[(dimx)]

line MAT aname = IDN[(dimx, any characters to end of line]

The first form, which gives the current dimension for the row value only, is sufficient to define the square matrix. The second form is provided for those users who wish to clearly indicate that the array is a square matrix. However, it may stand alone. Appropriate samples are shown.

```
1011 MAT Z=IDN
1012 MAT B=IDN(4)
1013 MAT C=IDN(3,3)
1014 MAT D=IDN(X+Y)
```

COPY

The copy statement copies arrays, and sets current dimensioning of the array copied into to that of the array copied from. The form of the copy statement is

line MAT aname = aname

In the sample copy statement below,

```
1015 MAT A=B
```

matrix B is copied into matrix A. Assume that B is a 4 x 4 matrix with current dimensioning (3,3). Only elements (1,1), (1,2), (1,3) ... (3,3) from B are copied into A. Further, any remaining elements in array A are not changed, and the current dimensioning of A becomes (3,3).

SCALAR MULTIPLICATION

The scalar multiplication operation multiplies an array by a scalar quantity. The form of the statement is

line MAT aname = (expr)* aname

where expr is an expression representing the scalar multiplier, and the parentheses are required. In sample statement 1016, below,

```
1016 MAT A=(A)*A
1017 MAT B=(SIN(X+H))*C
```

the parenthesized A is interpreted by BASIC as a simple variable, not as an array name.

ADDITION AND SUBTRACTION

Array addition and subtraction are performed through use of the statement shown below.

line MAT aname = aname{+ / -}aname

This statement adds or subtracts the corresponding elements of the two arrays named on the right of the equals sign and stores the results in the array named on the left. A sample is given for reference.

```
1018 MAT Z=B+C
```

The ambiguity introduced by allowing addition or subtraction of two vectors with storage in a matrix, or copying, transposition, and scalar multiplication of a vector into a matrix is resolved by considering the vectors as row vectors. Current dimensions of both arrays named to the right of the equals sign must be equal for addition and subtraction. The current dimensions of the array named on the left side of the equals sign are set equivalently.

TRANSPOSITION

It is not necessary to transpose a vector array; the result is an exact copy of the argument vector. Matrixes are transposed by use of the statement shown below.

line MAT aname = TRN(aname)

Sample statements are shown below

```
1019 MAT A=TRN(A)
1020 MAT B=TRN(C)
```

Current dimensioning of the matrix named on the left side of the equals sign is set consistent with current dimensioning of the matrix named on the right.

MULTIPLICATION

In the multiplication operation, vectors are taken to be row or column matrixes as appropriate. If a vector is multiplied by a vector, the scalar (dot) product results. The form of the multiplication statement is

line MAT aname = aname*aname

The following is a sample multiplication statement.

```
1021 MAT Z=B*C
```

Notes:

1. Current dimensioning must be consistent with the usual rules of matrix multiplication.
2. The same array name may not appear on both sides of the equals sign.

INVERSION

The inverse of a square matrix is specified as shown below.

line MAT aname = INV(aname[,simple variable])

where the inclusion of the simple variable (in which to store the computed determinant of the argument matrix) is a user option. Some sample statements are

```
1022 MAT A=INV(H)
1023 MAT B=INV(I,D)
```

In calculating the inverse of the square (by current dimensioning) argument matrix, the target matrix is initially set to an identity matrix. Then the target is converted by those elementary row operations that reduce the argument matrix to the identity matrix. Upon completion of the conversion, the target matrix is approximately the inverse of the argument matrix. The values of the argument matrix are destroyed; both matrixes have current dimensioning originally applicable to the argument matrix.

Notes: 1. The argument matrix must be square (according to its current dimensions).

2. Results are approximate, not exact.

3. At the user's option, the computed determinant of the argument is stored in a simple variable.

4. The contents of the argument matrix are destroyed, but current dimensions remain.

5. The same array name may not appear on both sides of the equals sign.

SIMULTANEOUS EQUATION SOLUTION

Solution of simultaneous equations is accomplished via the statement shown below.

line MAT aname = SIM(aname[,simple variable])

where the simple variable modification is a user option.

Some sample statements are shown below.

```
1024 MAT M=SIM(E)
1025 MAT S=SIM(H,D2)
```

The target array contains one or more sets of linear equation constant column vectors. The dimensions of this array must be compatible with the square argument matrix. For example, if the argument matrix has current dimension of (n,n), the target array must be either an n-dimension vector (one solution, or else an n x m matrix (m solutions). The argument matrix contains the coefficient matrix. The solution of the simultaneous equations is arrived at by converting the target array by those elementary row operations that reduce the argument matrix to the identity matrix. Upon completion of the conversion, the values of the argument matrix are destroyed, but current dimensions for both the target and the argument arrays are unchanged. The target array contains the appropriate values that are computed by taking

$$(\text{argument})^{-1} \times (\text{target})$$

This result is equivalent to solving one or m sets of simultaneous linear equations having the same coefficient matrix, that is, the argument matrix.

Notes: 1. The argument matrix must be square (according to its current dimensions).

2. Results are approximate, not exact.

3. At the user's option, the computed determinant of the argument is stored in a simple variable.

4. The contents of the argument matrix are destroyed, but current dimensions remain.

5. The same array name may not appear on both sides of the equals sign.

ACCURACY OF INVERSION AND SIMULTANEOUS EQUATION SOLUTION

The results of matrix inversion will vary in accuracy because of precision losses during the conversion process. If, during conversion, a pivotal element is smaller in magnitude than 10^{13}, it is considered to be zero and the matrix is considered singular. If all elements of a matrix are of small magnitude (e.g., 10^{-6} or less), it should be scaled upward so the greatest magnitude of any element is near unity. If a matrix consists of elements of large magnitude, it should be scaled down again to near unity for the maximum element.

When a determinant calculation is requested in using the inversion or simultaneous equation functions, the following special situations may occur:

1. If the determinant value calculation results in a magnitude greater than 7.234×10^{75}, the value of the simple variable will be the alphanumeric value OVERFL. This does not affect the calculation of the inverse or simultaneous equation solution.

2. If the matrix is singular, the simple variable is given a value of zero; the values of the argument and target arrays are destroyed.

7. BASIC MESSAGES

This chapter lists BASIC error messages and other messages in alphabetical order. In the messages, xxxxx represents line number and x represents array name, function letter, or declared letter. Except as noted in comments, an error causes termination of program execution.

xxxxx ACONST EXPECTED

A variable contains numeric data when it should contain an aconst.

xxxx ARG NO. ERR FNx

Conflict between the number of arguments defined and the number of arguments used with the function.

ARRAY CLASS CONFLICT

The indicated letter is used for more than one type of array. Example: B used for a string vector but dimensioned as a numeric vector.

xxxxx ASN-ACS ARG ERROR

The argument is outside the allowable limits ±1.

BAD BYTE

In executing a CHANGE statement, a value was not in the range 0-255.

xxxxx BAD CHAR

Statement xxxxx contains an illegal character (for example, !, ?, @, etc.). Note, however, that all characters having the EBCDIC value of blank or greater are allowed in Image statements, text strings, and alphanumeric constants.

xxxxx BAD CONST

Line xxxxx contains an improperly formed numeric or alphanumeric constant. Probable causes are

Numeric

1. Extra decimal points.
2. More than two digits in exponent fields (for example, .001E100).
3. Underflow or overflow in conversion to floating-point form.
4. Missing operator after the constant.

Alphanumeric

1. Missing quotation mark.
2. Single (or double) closing quotation mark does not match double (or single) opening quotation mark.
3. More than six characters between quotes.
4. Contains a character having an EBCDIC value less than that of the blank character (for example, the O character).

xxxx BAD FORMAT

This message covers a wide range of syntax errors. The user should reexamine statement definition if the error is not obvious.

xxxxx BAD FORMULA

An arithmetic expression error has been detected. This message covers a wide range of error situations such as missing operators, missing operands, misspelled function names, misspelled keywords, etc.

BAD STEP # AFTER STMT xxxxx

A line number contains a nonnumeric character or more than five digits.

xxxxx BAD STMT

The type of statement is not recognizable; most frequently, the command keyword has been misspelled.

xxxxx BAD STREAM NO.

An I/O stream number is outside of the legal range (1-4).

xxxxx BAD SUBSCR

A known subscript value is too small.

BAD SUBSTRING PARAM

Run-timer error. A substring index is nonpositive or starts beyond the maximum string length.

xxxxx BAD TEXT

A text string either contains a New Line character (user probably forgot the end-quote mark), or has an unmatching quote (as in 123 PRINT "DOUBLE QUOTE'), or contains a character having an EBCDIC value lower than 15.

CANNOT OPEN

Unable to OPEN a file

DATA MIX-UP, $STRING VS NUMERIC

Either numeric data is being input to a string, or text exceeding aconst length is being input to a simple or subscripted variable (via READ, INPUT, GET, MAT READ, MAT INPUT, or MAT GET).

xxxxx DEFD TWICE

A function defined in DEF statement xxxxx was also defined by an earlier DEF statement.

xxxxx DIM ERR

A DIM statement formula contains one of the following:

1. User function call.
2. Simple variable that is not SET to some value.
3. Subscripted variable reference.

xxxxx DIMD TWICE x

Multiple-dimensioning has been attempted. Revise DIM statements.

DIM TOO BIG

Run-time error. A dimension is too large in a matrix operation.

xxxxx DIV BY ZERO

Run-time or compile error. A zero denominator was encountered in expression evaluation.

xxxxx ERROR IN KEYED I/O

Reference to illegal key or attempt to access an unkeyed file in the keyed mode.

xxxxx EXP OVERFL

Floating-point overflow during exponentiation.

xxxxx EXTRA COMMA

Error indirectly associated with bad comma. Examples:

Syntax	Explanation
X + (Y,Y)	Array reference without array designator.
SIN (A,B)	Too many arguments in intrinsic function.
M(X,Y,Z)	Too many subscripts in array reference.

xxxxx EXTRA INPUT

Contents of input record not exhausted when check symbol '*' encountered.

xxxxx FILE I/O ERROR

Monitor indication of error in attempting to write or read on file.

xxxxx FILE NOT OPEN IN PROPER MODE

I/O operation attempted on file which is closed, or open in a conflicting mode, on the I/O stream in use.

xxxxx FOR-NEXT ERR

Message covers FOR-NEXT errors illustrated below:

Case 1. Wrong Variable Reference

```
FOR A
NEXT B          error
```

Case 2. Improperly Sequenced Statements

```
FOR I
FOR J
NEXT I          error
NEXT J          error
```

Case 3. No Corresponding FOR Statements

```
NEXT A          error
```

These messages may be compounded as in

```
FOR A
FOR B
FOR C
NEXT B              error, FOR C is cancelled
NEXT C              error, FOR B is cancelled
NEXT A
```

or alternatively

```
FOR A
FOR B
FOR C
NEXT B              error
NEXT C              error
:
:
END                 error, MISSING NEXTSTMT
```

xxxxx HALT

Normal message at termination of run.

xxxxx HAS BAD STEP NO.

A GOTO, GOSUB, IF, ON, PRINTUSING, or RESTORE contains a step number having more than five digits.

xxxxx HSN-HCS OVERFL)

Hyperbolic sine or hyperbolic cosine overflow.

ILLEGAL FILE ID

File name, password, or account identifier too long.

ILLEGAL INPUT FROM FILE

Illegal input from a file.

xxxxx INCOMPAT DIMS

Dimensions not compatible in matrix operation. Examples: matrix identity or inversion on nonsquare matrix, wrong dimensioning for matrix multiplication or addition, etc.

xxxxx INPUT DATA LOST

An input record is too big or has a parity error and has been discarded.

xxxxx KEY NOT FOUND

A keyed read was attempted on a file not containing a record with the specified key.

xxxxx LINE # ERR

An illegal line number (>99999) occurred immediately after the statement at xxxxx. If the first line is incorrect, xxxxx is zero.

xxxxx LOG OF NON-POS ARG

The argument in a logarithmtic operation is not greater than zero.

MISSING NEXT STMT

At least one FOR statement occurred without a matching NEXT statement; that is, there were more FORs than NEXTs.

MISSING STEPS

No list of line numbers was found in a GOTO...ON or ON...GOTO statement.

xxxxx NEG BASE TO NON-INTEGER POWER

Fractional exponentiation was indicated but the number is negative.

NO DIMSTMT ARRAY x

An array is used in MAT statement or string statements, but not dimensioned.

NON-EXISTENT LINE #

In the execution mode, a direct statement references a line number that is not in the compiled program.

xxxxx NON-NUMERIC VAL

A VAL function argument (string expression) does not represent a number.

xxxxx NON-POS DIM

Run-time error. A zero or negative dimension was encountered in a matrix operation.

xxxxx OUT OF RANGE REF. TO ARRAY x

A matrix operation compiled in the "safe" mode has referenced an index greater than the maximum dimension for array x.

xxxxx OVERFLOW

Floating-point overflow.

xxxxx PAREN ERR

This message indicates a parenthesis imbalance.

POWER OVERFLOW

Overflow in exponentiation.

xxxxx PROG TOO BIG

The program exceeds available memory.

xxxxx RECORD NOT IN -GET-FORMAT

An attempt to GET has encountered a record not in the proper format (see Appendix E).

xxxx RESTORE A NON-DATA LINE

A RESTORE statement indicates a line other than a DATA line.

xxxxx RESTORE A NON-EXISTENT LINE

A RESTORE statement indicates a nonexistent line number.

xxxxx RETURN BEFORE GOSUB

A RETURN statement was reached but the return stack is empty. Indicates improper nesting or branching on GOSUB-RETURNs.

xxxxx RUN INTERRUPTED

This message is issued after a "break" (not an error condition).

xxxxx SEC-CSC OVERFL

Secant or cosecant operation overflow.

xxxxx SHOULD BE DATA STMT

Line xxxxx was referenced in a READ or RESTORE statement but was not a recognizable DATA statement.

xxxxx SHOULD BE IMAGE STMT

Line xxxxx was referenced in a PRINTUSING statement but was not a recognizable Image statement.

xxxxx SINGULAR MATRIX

An inversion or simultaneous equation solution was attempted on a singular matrix.

xxxxx SQR ROOT OF NEG ARG

The argument of a square root function is negative.

xxxxx $TRING EXPR ERR

An incorrectly formatted string expression has been detected.

xxxxx TOO MANY GOSUBS BEFORE A RETURN

The return stack for GOSUB-RETURN logic is full and a GOSUB has been encountered.

xxxxx UNABLE TO OPEN xxxxx

An attempt to open the named file failed. The file is probably not present or has another account number or name.

xxxxx UNDEF FNx

No DEF statement appeared for user function FNx.

xxxxx ZERO TO NEG POWER

An exponentiation operation attempted to raise zero to a negative power.

APPENDIX A. SUMMARY OF BASIC STATEMENTS

The complete set of BASIC statements is shown below. Capital letters indicate syntax that is required as shown. Lowercase letters designate general items. Command parameters enclosed by braces ({ }) indicate a required choice. Parameters enclosed by brackets ([ ]) are optional. Ellipsis marks (...) denote multiple occurrences of the preceding parameter. Unless otherwise noted, "variable" means either a simple or a subscripted variable.

Statement

line:[#s and/or characters to end of line]

line CHAIN xname [[;password][;acct no] / [:acct no ;password]]

line CHAIN LINK xname [[;password][;acct no] / [:acct no][;password]]

line CLOSE {:stream / {I / O} [characters to end of line]}

line DATA {[±]constant / aconst / tstring} [, {[±]constant / aconst / tstring}]...

line DEF FN letter (simple variable[,simple variable]...) = expression

line DIM {letter$ / letter} (dimx[,dimx]) [, {letter$ / letter} (dimx[,dimx])]...

line END

line ENDFILE:stream,{line / E}

line FOR simple variable = expression TO expression STEP expression

line GET[:stream[;key],] {variable / string} [, {variable / string}]...

line GOSUB line

line GOTO line

line GOTO line [,line]... ON expression

line IF {expr / aconst / string} {operator} {expr / aconst / strexp / xstrexp} {THEN / GOTO} line

line INPUT [:stream[;key],] {variable / string} [, {variable / string}]...

line INPUT = {$ / any other character}

line [LET] {variable ,variable ... = {expression / aconst / xstrexp} / string = strexp} [, {variable , variable... = {expression / aconst / xstrexp} / string = strexp}]...

line MAT aname = (expression)*aname

line MAT aname = aname

line MAT aname = aname ± aname

line MAT aname = aname * aname

line MAT aname = CON[(dimx[,dimx])]

line MAT aname = IDN (dimx)
(dimx ,[any characters to end of line]

line MAT aname = INV (aname[,simple variable])

line MAT aname = SIM (aname[,simple variable])

line MAT aname = TRN (aname)

line MAT aname = ZER[(dimx[,dimx])]

line MAT GET [:stream[;key],] {letter$[vardim] / adescr} [, {letter$[vardim] / adescr}]...

line MAT INPUT [:stream[;key],] {letter$[vardim] / adescr} [, {letter$[vardim] / adescr}]...

line MAT PRINT [:stream[;key],] aname [{, / ;} aname] ...

line MAT PUT [:stream[;key],] {letter$[vardim] / adescr} [, {letter$[vardim] / adescr}]...

line MAT READ {letter$[vardim] / adescr} [, {letter$[vardim] / adescr}]...

line MAT SIZE {letter$ / aname}(dimx[,dimx]) [, {letter$ / aname}(dimx[,dimx])]...

line NEXT simple variable

line ON expression {GOTO / THEN} step[,step]...

line OPEN fileid , { I / O / TO:stream, } { GET / INPUT [UPDATE] / PUT [,] {ON / OVER} / PRINT } [[,] TFILE]

line PAGE

line PAUSE

line PRINT [:stream[;key],] [[[, ;]] [xstrexp] [expression† / xstrexp / text string]] ...

line PRINT [:stream[;key],] USING line [, { xstrexp / expression / text string }] ...

line PUT [:stream[;key],] { expression / aconst / xstrexp } [, { expression / aconst / xstrexp }] ...

line { * / REM } [characters to end of line]

line READ { variable / string } [, { variable / string }] ...

line RESTORE[line]

line RETURN

line STOP

†Must not begin with a '+'.

APPENDIX B. BASIC INTRINSIC FUNCTIONS

Function	Result
SIN(arg)	Calculates sine of argument in radians.
COS(arg)	Calculates cosine of argument in radians.
TAN(arg)	Calculates tangent of argument in radians.
ATN(arg)	Calculates arctangent of unitless argument in radians.
EXP(arg)	Calculates exponential functions, that is $e^{(argument)}$.
ABS(arg)	Calculates absolute value of argument.
LOG(arg)	Calculates natural logarithm (base e) of the argument.
LGT(arg)	Calculates common logarithm (base 10) of the argument.
SQR(arg)	Calculates square root of argument.
INT(arg)	Acquires the integer part of the argument, that is, the greatest integer that is less than or equal to the argument.
SGN(arg)	Identifies algebraic sign of argument, and produces a -1 for negative arguments, a 0 for 0, and a +1 for positive arguments.
RND(arg)	Produces, for each call, the next element of a sequence of uniformly distributed random numbers that are greater than 0 but less than 1. If arg is 0 for the first RND call of a program, the identical sequence of random numbers will be generated if the program is rerun and arg is not changed. Otherwise, an unrepeatable sequence will be generated.
DAY(arg)	Supplies the calendar day. If the argument is 0, the BTM output form is mm/dd (as in 03 r March 7) and the BPM and UTS output form is mon/dd (as in MAR 07). If the argument is nonzero, the output form is a floating-point number whose integer part represents the month, and whose fractional part represents the day of the month divided by 100. For example, 3.07E0 represents March 7.
TIM(arg)	Supplies the time of day. If the argument is 0, the output form is hh:mm, as in 15:09. If the argument is nonzero, the output form is a floating-point number whose integer part represents the hour and whose fractional part represents the minutes divided by 60. For example, 15.15E0 represents 3:09 PM.
YER(arg)	Supplies the year. If the argument is 0, the output form is 19yy, as in 1969. If the argument is nonzero, the output form is a floating-point number whose value is equal to the year, as in 1969.0E0.
MAX(arg_n)	Selects the maximum value in the list of arguments.
MIN(arg_n)	Selects the minimum value in the list of arguments.
TAB(arg)	Advances the print device to the column designated by the argument, and should only be used in a PRINT statement. TAB cannot be used to backspace the print device.
PRC(arg)	Specifies the number of significant digits in printed output, and is used only in a PRINT statement. An argument of 0 specifies 6-significant-digit output format, and a nonzero argument specifies 16-significant-digit output.

Function	Result
CSC (arg)	Calculates cosecant of an argument in radians. Overflow results in an error message and termination of execution.
SEC(arg)	Calculates secant of an argument in radians. Overflow results in an error message and termination of execution.
COT(arg)	Calculates cotangent of an argument in radians. Overflow results in an error message and termination of execution.
ASN(arg)	Calculates arcsine of a unitless argument, in radians. If the absolute value of the argument is greater than 1.0, an error message is printed and execution is terminated. Resolution of results is restricted to the two quadrants from $-\pi/2$ to $\pi/2$.
ACS(arg)	Calculates the arccosine of a unitless argument, in radians. If the absolute value of the argument is greater than 1.0, an error message is printed and execution is terminated. Resolution of results is restricted to the two quadrants from 0 to π.
HSN(arg)	Calculates hyperbolic sine of an argument. Overflow results in an error message and termination of execution.
HCS(arg)	Calculates hyperbolic cosine of an argument. Overflow results in an error message and termination of execution.
HTN(arg)	Calculates hyperbolic tangent of an argument.
LTW(arg)	Calculates logarithm, base two, of an argument.
DEG(arg)	Converts argument to degrees, from radians.
RAD(arg)	Converts argument to radians, from degrees.
LEN(strexp)	Gets current number of characters in string expression, as floating-point number.
VAL(strexp)	Gets numeric value of string expression as floating-point value. Error exit if string expression not numeric.
STR(expression [, rstring])	Converts numeric value of expression to string format. Optional rstring argument permits specific formatting. If second argument is not used, standard print output format is used.
KEY(arg)	Returns the value of the key most recently accessed on the I/O stream specified by the argument.

APPENDIX C. FORMAT OF BINARY DATA FILES FOR BASIC (PUT AND GET OPERATIONS)

The PUT and MAT PUT operations in BASIC create data files in the internal format described in Table 3 with a physical record size of 120 bytes.

Table 3. Internal Format of Data Files

Byte	Coding	Meaning
0	X'3C'	Physical record.
1	Checksum	Sum of bytes in record, not counting checksum byte.
2,3	Record size	Number of bytes used (120 or less), including control bytes.
4...n	Data	Either doubleword floating-point or aconst doubleword or both.
n+1	X'3C'	End of physical or logical record.

Table 3. Internal Format of Data Files (cont.)

Byte	Coding	Meaning
n+2	X'BD'	
n+3, n+4	Physical record number	In numerical order, from 0.

Normally a record contains 112 noncontrol bytes (14 floating-point values or aconsts). The last record in a file may contain fewer used bytes but still contains 120 total bytes. The control word – bytes n+1 to n+4 – is repeated in this case as bytes 116 to 119.

Figure 4 shows a file containing three records of numeric and aconst data, with the record contents given in hexadecimal format. The values were created with the program shown in Figure 5. In Figure 4, the value 1 occupies words 1 and 2 of record 1000, the aconst ABCDEF occupies words 13 and 14 of record 2000, and the aconst 7890 occupies words 25 and 26 of the same record but followed in word 27 by an end-of-record control word forced there by the flush operation.

```
KEY= X'001000'    -       120 BYTES

00000    3C090078    41100000    00000000    41200000
00004    00000000    41300000    00000000    41400000
00008    00000000    41500000    00000000    41600000
0000C    00000000    41700000    00000000    41800000
00010    00000000    41900000    00000000    00000000
00014    00000000    00000000    00000000    41900000
00018    00000000    41800000    00000000    41700000
0001C    00000000    3CBD0000

KEY= X'002000'    -       120 BYTES

00000    3CA60070    41600000    00000000    41500000
00004    00000000    41400000    00000000    41300000
00008    00000000    41200000    00000000    41100000
0000C    00000000    0001C1C2    C3C4C5C6    0001C7C8
00010    C9D1D2D3    0001D4D5    D6D7D8D9    0001E2E3
00014    E4E5E6E7    0001E8E9    00000000    0001F1F2
00018    F3F4F5F6    0001F7F8    F9F00000    3CBD0001
0001C    00000000    3CBD0001
```

Figure 4. Contents of Sample File

```
KEY= X'003000'     -        120 BYTES

00000    3C030028   41100000   00000000   41200000
00004    00000000   41300000   00000000   41400000
00008    00000000   3CBD0002   00000000   41100000
0000C    00000000   0001C1C2   C3C4C5C6   0001C7C8
00010    C9D1D2D3   0001D4D5   D6D7D8D9   0001E2E3
00014    E4E5E6E7   0001E8E9   00000000   0001F1F2
00018    F3F4F5F6   0001F7F8   F9F00000   3CBD0001
0001C    00000000   3CBD0002
```

Figure 4. Contents of Sample File (cont.)

```
100 OPEN 'PUT',0
110 PUT 1,2,3,4,5,6,7,8,9,0
120 PUT 0,9,8,7,6,5,4,3,2,1
130 PUT 'ABCDEF','GHIJKL','MNOPQR','STUVWX','YZ'
140 PUT '123456','7890'
150 PUT**
160 PUT 1,2,3,4,**
170 CLOSE 0
180 END
```

Figure 5. Program Used to Generate Figure 4

APPENDIX D. EASY ERROR MESSAGES

ACCOUNT NUMBER ILLEGAL IN THIS COMMAND.

The parameter noted is illegal in this command.

BAD PARAMETER -- TRY AGAIN

Incorrect syntax -- check and retry.

CANNOT ACCESS NEXT FILE

The CATALOG command cannot be completed because of bad file structure.

CANNOT LOAD BINARY FILES

You have issued an OLD command for a binary file.

DELIMITER MISSING - RETYPE

Omitted delimiter in an EDIT FIND or REPLACE command.

DISC IS SATURATED: DATA LOST

The last command saturated secondary storage.

ERROR IN TARGET WORD

The target word in an EDIT FIND or REPLACE command is not valid.

FILE ALREADY SAVED - TYPE REPLACE TO OVERWRITE

A SAVE command has been given but the file already exists.

FILE IN USE - TO TRY AGAIN, RETYPE COMMAND

An attempt has been made to access a file that someone is currently updating.

FILE I/O ERROR : DATA LOST

There was an I/O error -- retry the command.

FILENAME ACCOUNT OR PASSWORD TOO LONG

The parameter entered is too long.

FILE NAME ILLEGAL IN THIS COMMAND.

The parameter noted is illegal in this command.

FILENAMES AND PASSWORDS MUST BEGIN WITH AN ALPHA CHARACTER

Self explanatory.

FIRST NUMBER OF PAIR MUST BE SMALLEST

You have used a pair of numbers to specify a range but have incorrectly entered the larger number first.

INSUFFICIENT INFORMATION TO OPEN FILE

Check to make sure you have supplied a password if needed and retry the comman

INVALID PASSWORD

The password you have used is not correct.

LINE NUMBER EXCEEDS 99999

An EDIT RESEQUENCE command has attempted to generate a line number greater than 99999.

LINE TABLE FULL - CUT FILE SIZE

There is no more core available to put file keys into.

LINE TOO LONG

An EDIT REPLACE command has tried to increase a line beyond 132 characters.

LINE TOO LONG: DATA LOST

A read operation has failed because the record was longer than 132 characters.

LINE TOO LONG -- RETYPE

The line entered is longer than 132 characters.

NO FILE

You have tried to run or edit an empty file.

NONE OF THE FILES EXIST

An EDIT MERGE, INSERT, or WEAVE command has not specified any existing files.

NO SUCH FILE: xxx

The file named does not exist.

NUMBER MISSING - RETYPE

You have not entered a number where one is required.

PARAMETER CONFLICT : TRY AGAIN

Illegal syntax -- check the command and try again.

PASSWORD ILLEGAL IN THIS COMMAND.

The parameter noted is illegal in this command.

THE NEXT AVAILABLE LINE NUMBER IS TOO LARGE

A DSM, EDIT WEAVE, INSERT, MERGE or RESEQUENCE command has attempted to assign a line number larger than 99999.

TOO MANY FILES SPECIFIED

You have specified more filenames in a command than is allowed.

WHAT?

EASY cannot interpret what was just typed in; check the syntax and try again.

xx ERROR AT yyyyyyyy

xx is an error code explained in the UTS Reference Manual (90 17 64), Appendix B. yyyyyyyy is the hexadecimal location at which the error occurred.

APPENDIX E. FLAG ADDITIONS

FILE SUBROUTINES

Two new subroutines, OPENF and CLOSEF, have been added to FLAG.

OPENF This subroutine allows the FLAG user to access a specified file in his account in UTS secondary storage. The form of the call is

CALL OPENF(u,'name'[,org[,mode[,granules]]])

where

u specifies a device unit number in the range 1-7.

'name' specifies a file name of 1-7 characters. A trailing blank is required if the name has fewer than 5 characters.

org specifies file organization (1 = consecutive, 3 = random). The default is consecutive.

mode specifies the file function mode (1 = read, 2 = write, 4 = update, 8 = write then read). The mode must be 8 for random files. The default is 4 (read or write on old file).

granules specifies the number of granules to allocate for a random file. The default is 12. This parameter is meaningful only if org = 3.

CLOSEF This subroutine allows the FLAG user to close a specified file. The form of the call is

CALL CLOSEF(u[,dis])

where

u specifies a device unit number in the range 1-7.

dis specifies file disposition (0=save, nonzero=release). The default is save

DEVICE UNIT NUMBERS

Unit numbers for FORTRAN I/O are 1-7. However, 5, 6, and 7 are defaults for FORTRAN II type statements, format-free I/O, and INPUT and OUTPUT statements. Unit number 105 is mapped into 5, 108 into 6, and 106 into 7. Unit numbers 5, 6, and 7 default to the user terminal. A CALL OPENF(6,'fileone') followed by an OUTPUT,k statement will cause output to go to a file rather than the terminal. A CALL CLOSEF(6) will close unit 6. Subsequent use of unit 6 will default to the terminal.

ON-LINE OPERATIONS

Since Batch processing is not done under EASY, the FLAG user need not be concerned with DCB assignments or job deck considerations. EASY provides all the default options listed in the FLAG Reference Manual, except that DB (debug mode) is assumed rather than NODB.

COMMENTS AND CONTINUATION LINES

Comment lines in FORTRAN programs must begin with the letter C in the first column following the line number. Continuation lines must begin with an & character in the first nonblank column following the line number.

XEROX

Reader Comment Form

We would appreciate your comments and suggestions for improving this publication

Publication No.	Rev. Letter	Title	Current Date

How did you use this publication?

☐ Learning ☐ Installing ☐ Sales
☐ Reference ☐ Maintaining ☐ Operating

Is the material presented effectively?

☐ Fully Covered ☐ Well Illustrated ☐ Well organized ☐ Clear

What is your overall rating of this publication?

☐ Very Good ☐ Fair ☐ Very Poor
☐ Good ☐ Poor

What is your occupation?

Your other comments may be entered here. Please be specific and give page, column, and line number references where applicable. To report errors, please use the Xerox Software Improvement or Difficulty Report (1188) instead of this form.

Your name & Return Address

Thank You For Your Interest (fold & fasten as shown on back, no postage needed if mailed in U S A)

CUT ALONG LINE

FOLD ALONG LINE

PLEASE FOLD AND TAPE–
NOTE: U. S. Postal Service will not deliver stapled forms

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 59153 LOS ANGELES,CA 90045

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
5250 W. CENTURY BOULEVARD
LOS ANGELES, CA 90045

ATTN: PROGRAMMING PUBLICATIONS

FOLD ALONG LINE

Honeywell

Honeywell Information Systems
In the U.S.A.: 200 Smith Street, MS 486, Waltham, Massachusetts 02154
In Canada: 2025 Sheppard Avenue East, Willowdale, Ontario M2J 1W5
In Australia: 124 Walker Street, North Sydney, N.S.W. 2060
In Mexico: Avenida Nuevo Leon 250, Mexico 11, D.F.

26025, 1.25C1179, Printed in U.S.A.

XM29, Rev. 0