HP 3000 Series II Computer System

BASIC Interpreter

Reference Manual

5303 STEVENS CREEK BLVD., SANTA CLARA, CALIFORNIA, 95050

Part No. 30000-90026
Product No. 32101B

Printed in U.S.A. 6/76

NOTICE

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced or translated to another program language without the prior written consent of Hewlett-Packard Company.

Copyright © 1976 by HEWLETT-PACKARD COMPANY

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

LIST OF EFFECTIVE PAGES

The List of Effective Pages gives the most recent date on which the technical material on any given page was altered. If a page is simply re-arranged due to a technical change on a previous page, it is not listed as a changed page. Within the manual, changes are marked with a vertical bar in the margin.

Pages	Effective Date
Title	June 1976
ii to x	June 1976
1-1 to 1-15	June 1976
2-1 to 2-65	June 1976
3-1 to 3-20	June 1976
4-1 to 4-12	June 1976
5-1 to 5-23	June 1976
6-1 to 6-12	June 1976
7-1 to 7-22	June 1976
8-1 to 8-37	June 1976
9-1 to 9-14	June 1976
10-1 to 10-16	June 1976
11-1 to 11-12	June 1976
12-1 to 12-10	June 1976
A-1 to A-3	June 1976
B-1 to B-2	June 1976
C-1 to C-10	June 1976
D-1 to D-7	June 1976
E-1 to E-4	June 1976
F-1 to F-4	June 1976
G-1	June 1976
H-1 to H-3	June 1976
Index-1 to Index-6	June 1976

PRINTING HISTORY

New editions incorporate all update material since the previous edition. Update packages, which are issued between editions, contain additional and replacement pages to be merged into the manual by the customer. The date on the title page and back cover changes only when a new edition is published. If minor corrections and updates are incorporated, the manual is reprinted but neither the date on the title page and back cover nor the edition change.

First Edition . . . . June 1976

Contents

SECTION I Introduction to BASIC	1-1
SPECIAL KEYS	1-2
PROMPT CHARACTERS	1-3
LOGGING ON AND OFF	1-4
Logging On	1-4
Entering BASIC	1-4
Leaving BASIC	1-5
Logging Off	1-5
Suspending BASIC	1-5
CORRECTING ERRORS	1-6
BASIC COMMANDS AND STATEMENTS	1-7
BASIC PROGRAMS	1-10
USER'S WORK AREA	1-11
LISTING A PROGRAM	1-12
RUNNING A PROGRAM	1-13
DELETING A PROGRAM	1-14
DOCUMENTING A PROGRAM	1-15
SECTION II Essentials of BASIC	2-1
EXPRESSIONS	2-2
Constants	2-2
Variables	2-4
Functions	2-5
Operators	2-6
Evaluating Expressions	2-8
STATEMENTS	2-10
ASSIGNMENT STATEMENT	2-11

SECTION II (Continued)

REM STATEMENT	2-13
GOTO STATEMENT	2-14
GOSUB/RETURN STATEMENTS	2-16
END/STOP STATEMENTS	2-19
LOOPING STATEMENTS	2-22
CONDITIONAL STATEMENTS	2-25
PRINT STATEMENT	2-31
Print Functions	2-36
READ/DATA/RESTORE STATEMENTS	2-39
INPUT STATEMENT	2-42
ENTER STATEMENT	2-47
>BASIC	2-49
COMMANDS	2-50
RUN	2-51
EDITING COMMANDS	2-54
LIST	2-54
SCRATCH	2-55
DELETE	2-55
RENUMBER	2-56
LENGTH	2-56
LIBRARY COMMANDS	2-59
NAME	2-59
SAVE	2-59
GET	2-61
PURGE	2-61
APPEND	2-62
CATALOG	2-62

SECTION III Arrays	3-1
DIM STATEMENT	3-3
REDIM STATEMENT	3-4
STORING DATA IN ARRAYS	3-6
MAT READ/INPUT Statements	3-6
PRINTING DATA FROM ARRAYS	3-8
MAT PRINT Statement	3-8
INITIALIZING ARRAYS	3-10

SECTION III (Continued)

ARRAY OPERATIONS 3-12

Array Copying 3-12

Array Addition/Subtraction 3-13

Array Multiplication 3-14

Array Inversion 3-16

Array Transposition 3-18

Array Scalar Multiplication 3-19

ARRAY FUNCTIONS 3-20

SECTION IV Variable Types 4-1

TYPE STATEMENTS 4-2

Numeric Constant Forms 4-2

Printing Long and Complex Data 4-6

Numeric Expressions 4-7

Conditional Statement 4-8

Numeric Assignment 4-8

Entering Numeric Data 4-9

Other Uses of Data Types 4-9

Numeric Arrays 4-10

Function Class 4-11

SECTION V Strings 5-1

LITERAL STRINGS 5-1

DIM STATEMENT WITH STRINGS 5-3

REDIM STATEMENT WITH STRINGS 5-5

STRING VARIABLE 5-6

STRING EXPRESSIONS 5-9

STRING ASSIGNMENT 5-10

STRING-RELATED FUNCTIONS 5-12

COMPARING STRINGS 5-16

STRING INPUT AND OUTPUT 5-17

Reading Strings 5-17

Inputting Strings 5-18

Entering Strings 5-18

Printing Strings 5-20

LINPUT STATEMENT 5-21

SECTION V (Continued)

STRING ARRAY OPERATIONS 5-22
CONVERT STATEMENT 5-23

SECTION VI **User-Defined Functions** 6-1
ONE-LINE FUNCTION 6-2
MULTILINE FUNCTION 6-4
CALLING A USER-DEFINED FUNCTION 6-7
Passing Parameters 6-10

SECTION VII **Debugging** 7-1
TRACE/UNTRACE COMMANDS 7-2
BREAK/UNBREAK COMMANDS 7-7
Legal Commands During Break 7-8
ABORT COMMAND 7-11
RESUME OR GO COMMAND 7-12
SHOW COMMAND 7-14
SET COMMAND 7-16
FILES COMMAND 7-18
CALLS COMMAND 7-20

SECTION VIII **Files** 8-1
CREATING A FORMATTED FILE 8-3
PURGING A FILE 8-5
OPENING FILES 8-6
FILES STATEMENT 8-7
ASSIGN STATEMENT 8-9
FILE ACCESS 8-12
SERIAL FILE PRINT 8-13
SERIAL FILE READ 8-15
FILE RESTORE STATEMENT 8-17
DIRECT FILE PRINT 8-18
DIRECT FILE READ 8-20
ASCII FILE ACCESS 8-22
FILE LINPUT STATEMENT 8-23
BINARY FILE ACCESS 8-24
DYNAMIC LOCKING 8-25

SECTION VIII (Continued)
ON END STATEMENT 8-27
ADVANCE STATEMENT 8-29
UPDATE STATEMENT 8-30
LISTING FILE CONTENTS 8-31
DUMP Dommand 8-31
FILE FUNCTIONS 8-32
TYP Function 8-32
REC Function 8-34
FILE ARRAY OPERATIONS 8-35
Serial File MAT PRINT Statement 8-35
Serial File MAT READ Statement 8-35
Direct File MAT PRINT Statement 8-36
Direct File MAT READ Statement 8-37

SECTION IX Formatted Output 9-1
PRINT USING STATEMENT 9-2
MAT PRINT USING STATEMENT 9-4
IMAGE STATEMENT 9-6
FORMAT STRINGS 9-7

SECTION X Segmentation 10-1
CHAIN STATEMENT 10-2
INVOKE STATEMENT 10-4
FILES AND SEGMENTATION 10-7
COM STATEMENT 10-9

SECTION XI Communication with Non-BASIC Programs 11-1
CALL STATEMENT 11-2
SYSTEM/RESUME COMMANDS 11-10
SYSTEM STATEMENT 11-12

SECTION XII Non-Interactive Programs 12-1
CARD READER/LINE PRINTER 12-2
PAPER TAPE 12-5
Preparing a Paper Tape 12-5
Punching Paper Tape Off-Line 12-6

SECTION XII (Continued)
Reading Paper Tape 12-7
COMMAND INPUT FROM FILES 12-9

APPENDIX SECTION

APPENDIX A ASCII Character Set A-1
APPENDIX B Error Messages B-1
APPENDIX C BNF Syntax for BASIC/3000 C-1
APPENDIX D Summary of BASIC/3000 Statements & Commands D-1
STATEMENT SUMMARY D-1
COMMAND SUMMARY D-6
APPENDIX E Built-in Functions E-1
APPENDIX F Parameter Format F-1
APPENDIX G Compatibility Between BASIC/2000 & BASIC/3000 G-1
APPENDIX H File Structure H-1

SECTION I
Introduction to BASIC

HP BASIC/3000 is a programming language designed for use at a keyboard terminal. It may also be used for batch jobs on paper tape and cards. To use BASIC at a terminal, the terminal must gain access to the BASIC/3000 Interpreter through the HP Multiprogramming Executive Operating System (MPE/3000). The BASIC/3000 Interpreter is the control program for BASIC/3000.

BASIC/3000 consists of statements for writing programs and commands for controlling program operation. This section describes how to log on and log off, how to enter commands and statements and make corrections. A few simple programs are used for illustration, but the actual programming language is not described until Section II.

This manual assumes that the user knows how to connect his terminal, and is familiar with his terminal keyboard. Special keys with particular functions in BASIC/3000 are described in this section.

In this section only, characters typed by the computer are underlined to distinguish them from user input. Subsequent sections assume that this distinction is clear to the user.

Special Keys

return	Must be pressed after every command and statement. It terminates the line and causes the teleprinter to return to the first print position. BASIC returns a linefeed.
linefeed	Advances the teleprinter one line.
CTRL	When pressed simultaneously with another key, converts the key to a control character that is usually non-printing.
CTRL H (H^c)	Deletes the previous character in a line. It prints the character \ for each character deleted.
CTRL X (X^c)	Cancels the line currently being typed. It types three exclamation points on the line and then gives a return and linefeed to the beginning of the next line.
CTRL Y (Y^c)	Suspends a particular BASIC/3000 program or command and returns to the BASIC/3000 Interpreter. To return control to a program, type GO.
BREAK	Stops all BASIC/3000 activity and returns the user to the operating system (MPE/3000). BASIC/3000 can be re-entered by typing RESUME.

Prompt Characters

BASIC/3000 uses a set of prompting characters to signal to the user that certain input is expected or that certain actions are completed.

> The prompt character for the BASIC/3000 Interpreter; a BASIC command or statement is expected.

: The prompt character for the MPE Operating System; MPE commands such as HELLO or BASIC are expected.

? User input is expected during execution of an INPUT statement.

?? Further input is expected during execution of an INPUT statement.

>> BASIC expects a continuation line when the previous line was terminated by a &.

!!! A full line has been deleted with CTRL X.

\ A single character was deleted with CTRL H.

??? A BASIC command was mistyped; re-enter it correctly.

Logging On and Off

LOGGING ON

Once the terminal is connected and ready, the user presses the carriage return. MPE responds with a colon (:) at the beginning of the line. The user may now log on. He should know his user and account identification codes, and also the user and account passwords.

To log on, type:

JOANG and STUDENT are sample user and account identification codes. A period must be typed between them. The computer types a mask over which the passwords are typed. This preserves password privacy.

```
:HELLO JOANG.STUDENT
USER PASSWORD?
XXXXXXXXX
ACCOUNT PASSWORD?
XXXXXXXXX
SESSION NUMBER = #S5
WED, MAY 16, 1973,  2:26 PM
HP32000B.Q1.25
```

The last line identifies the Multi-programming Executive Operating System (MPE/3000).

ENTERING BASIC

Following log on, the MPE/3000 Operating System signals it is ready for the next command by printing a colon. The user may now request the BASIC/3000 Interpreter by typing BASIC.

To enter BASIC, type:

```
:BASIC
```

BASIC signals that it has control with a greater-than sign at the start of the line.

```
BASIC 01.0
>
```

BASIC commands and statements can now be entered. Each command or statement is prompted by the greater-than sign at the start of a new line.

LEAVING BASIC

When the user is through, he returns control to MPE/3000 with the EXIT command.

To leave BASIC, type:	>EXIT
The computer prints:	END OF PROGRAM
and MPE/3000 signals that it has resumed control with a colon.	:

LOGGING OFF

When a session at the terminal is finished, the user logs off with the MPE/3000 command BYE. He must have already exited from the BASIC Interpreter by typing EXIT. When MPE/3000 prints a colon, the user can type BYE.

To log off, type:

```
:BYE
```

MPE/3000 records the date and the time. It also records the number of minutes the terminal was connected and the seconds of central processor time used.

```
CPU (SEC) = 3
CONNECT (MIN) = 2
WED, MAY 16, 1973,  2:28 PM
END OF SESSION
```

SUSPENDING BASIC

The user may want to return to the MPE/3000 Operating System temporarily. He can leave BASIC, return to MPE/3000 control, enter MPE/3000 commands and then return to the same point in his BASIC program. To do this, he uses the SYSTEM command or the BREAK key. For operation of the BREAK key, see Special Keys, this section.

To suspend BASIC operation:	>SYSTEM
The computer types a colon:	:

The user may then enter MPE/3000 commands. When he wishes to return to BASIC, he types the MPE/3000 command RESUME. The system responds with a > .

Correcting Errors

Several types of errors may be made while logging on. We will consider spelling mistakes, format errors and incorrect passwords or codes. The methods for correcting these errors are general and can be used in BASIC as well as under control of MPE/3000.

Corrections can be made while the line is being entered if the error is noticed before *return* is pressed. The control character *CTRL H* (H^c) can be used to correct a few characters just typed, or the control character *CTRL X* (X^c) can be used to cancel the line and start fresh.

Suppose the user misspells the command HELLO. H^c will delete the last character and print a back slash. The user retypes the character correctly and finishes the line. When he presses *return*, the line is entered correctly.

```
:HELO\LO JOANG.STUDENT
USER PASSWORD?
```

If several characters have been typed after the error, H^c must be typed for each character to be deleted.

In this case, four characters including the blank are deleted.

```
:HELO JO\\\\LO JOANG.STUDENT
USER PASSWORD?
```

Another method is to use X^c to cancel the line. X^c must be typed before *return* is pressed.

To cancel the line, type X^c
Three exclamation points are typed and the computer responds with a carriage return and linefeed. The user retypes the line:

```
:HELO!!!
HELLO JOANG.STUDENT
```

BASIC Commands and Statements

Commands

BASIC/3000 commands instruct the BASIC/3000 Interpreter to perform certain control functions. Commands differ from the statements used to write a program in the BASIC/3000 language. A command instructs the interpreter to perform some action immediately, while a statement is an instruction to perform an action only when the program is run. A statement is always preceded by a statement number; a command never is.

Any BASIC/3000 command can be entered following the BASIC prompt character > . Each command is a single word that must be typed in its entirety with no embedded blanks. If misspelled, the computer will return an error message. Some commands have parameters to further define command operation.

For instance, EXIT is a command that signals completion of a BASIC program and return to the operating system. It has no parameters. Another command, LIST, prints the program currently being entered. It may have parameters to specify that only part of the program is to be listed, or to indicate a particular list destination.

Statements

Statements are used to write a BASIC/3000 program that will subsequently be executed. Each statement performs a particular function. Every statement entered becomes part of the current program and is kept until explicitly deleted or the user exits from BASIC with EXIT.

A statement is always preceded by a statement number. This number is an integer between 1 and 9999. The statement number indicates the order in which the statements will be executed. Statements are ordered by BASIC from the lowest to the highest statement number. Since this order is maintained by the interpreter, it is not necessary for the user to enter statements in execution order so long as the numbers are in that order.

Following each statement, *return* must be pressed to inform the interpreter that the statement is complete. The interpreter generates a linefeed and prints the prompt character > on the next line to signal that the statement is accepted. If an error is made entering the statement, the computer prints an error message.

BASIC/3000 statements have a free format. This means that blanks are ignored.

For instance, all these statements are equivalent.

```
>3Ø PRINT S
>3Ø      PRINT   S
>3ØPRINTS
>  3 Ø P R I N T S
```

Any statement except REM (to introduce remarks) can continue on more than one line. Each line to be continued must end with the character &; only the first line has a statement number. When the computer expects a continuation line, it prompts with two greater-than characters.

The statement 100 PRINT 35+5 is entered on two lines:

```
>100 PRINT&
>>35+5
```

Error Messages

If an error is made in a line and the line is entered with *return*, the interpreter types a message. The message consists of the word ERROR followed by @ and a number indicating about how many non-blank characters were read before an error was detected.

If this line is entered; the computer prints a message.

```
>30 PRING S
ERROR@2
```

The user then presses *return* and enters the correct line after the BASIC prompt character > .

If the mistake is not obvious, type any character after the message instead of pressing *return.* The computer will print a diagnostic message.

For instance:

```
>30 PRING S
ERROR@2;
UNRECOGNIZABLE STATEMENT TYPE
```

Typing a semi-colon causes the diagnostic message to be printed. Any other character, except a colon, could have been typed with the same result. A colon will cause an abort.

Changing or Deleting a Statement

If an error is made before *return* is pressed, the error can be corrected with *CTRL H* (H^C) or the line may be cancelled with *CTRL X* (X^C). (See Correcting Errors, above). After *return* is pressed, the error can be corrected by deleting or changing the statement.

To change a statement, simply type the statement number followed by the correct statement.

To change this statement:	`>30 PRINT X`
retype it as:	`>30 PRINT S`

A change such as this can be made any time before the program is run.

To delete a statement, type the statement number followed by *return*.

Statement 30 is deleted:	`>30`

The DELETE command, described in section II, is useful to delete a group of statements.

BASIC Programs

Any statement or group of statements that can be executed constitutes a program.

A program can have as few as one statement.

This is an example of a program with only one statement.

```
>100 PRINT 35+5
```

100 is the statement number. PRINT is the key word or instruction that tells the interpreter the kind of action to perform. In this case, it prints the result of the expression that follows. 35+5 is an arithmetic expression. It is evaluated by the interpreter, and when the program is run, the result is printed.

The statement 100 PRINT 35+5 is a complete program since it can run with no other statements and produce a result. Usually a program contains more than one statement.

These three statements are a program:

```
>10 INPUT A,B,C,D,E
>20 LET S = (A+B+C+D+E)/5
>30 PRINT S
```

This program, which calculates the average of five numbers, is shown in the order of its execution. It could be entered in any order if the statement numbers assigned to each statement were not changed.

This program runs exactly like the program above.

```
>20 LET S=(A+B+C+D+E)/5
>10 INPUT A,B,C,D,E
>30 PRINT S
```

It is generally a good idea to number statements in increments of 10. This allows room to intersperse additional statements as needed.

User's Work Area

When statements are typed at the terminal, these statements become part of the user's work area. All statements in the user's work area constitute the current program.

Any statement in the user's work area can be edited or corrected; the resulting statement will then replace the previous version in the user's work area.

When the user exits from BASIC with the EXIT command, the work area is cleared. If, however, he only suspends BASIC operation with the SYSTEM command, the *BREAK* key, or the *CTRL Y* keys, the user's work area is not changed.

Listing a Program

At any time while a program is being entered, the LIST command can be used to produce a listing of the statements that have been accepted by the computer. LIST causes the computer to print a listing of the current program at the terminal.

After deleting or changing a line, LIST can be used to check that the deletion or correction has been made.

A correction is made while entering a program:

```
>10 U\INPUT A,B,C,D,E
>20 PR\\LET S = (A+B+C+D+E)/5
>30 PRINT S
```

To check the correction, list the program:

```
>LIST
  10 INPUT A,B,C,D,E
  20 LET S=(A+B+C+D+E)/5
  30 PRINT S
>
```

Note that the greater-than prompt character is not printed in the listing, but is printed when the list is complete to signal that BASIC is ready for the next command or statement.

Should the statements have been entered out of order, the LIST command will cause them to be printed in ascending order by statement number.

For instance, the program is entered in this order:

```
>20 LET S = (A+B+C+D+E)/5
>30 PRINT S
>10 INPUT A,B,C,D,E
```

The list is in correct order for execution:

```
>LIST
  10 INPUT A,B,C,D,E
  20 LET S=(A+B+C+D+E)/5
  30 PRINT S
>
```

Running a Program

After the program is entered and, if desired, checked with LIST, it can be executed with the RUN command. RUN will be illustrated with two sample programs.

The first program has one line:

```
>100 PRINT 35+5
```

When run, the result of the expression 35+5 is printed:

```
>RUN
 40
```

Because the program contains a PRINT statement, the result is printed when the program is run.

The second sample program averages a group of five numbers. The numbers must be input by the user:

```
>10 INPUT A,B,C,D,E
>20 LET S=(A+B+C+D+E)/5
>30 PRINT S
```

Each of the letters following the word INPUT and separated by commas names a variable that will contain a value input by the user from the terminal. When the program is run, the interpreter signals that input is expected by printing a question mark. The user enters the values following the question mark. They are entered with a comma between each successive value.

The statement LET S = (A+B+C+D+E)/5 assigns the value of the expression to the right of the equal sign to the variable S on the left of the equal sign. The expression first adds the variable values within parentheses and then divides them by 5. The result is the value of S.

When the program is run, the user enters input values and the computer prints the result:

```
>RUN
?7,5,6,8,9
 7
```

Deleting a Program

If a program that has been entered and run is no longer needed, it can be deleted with the SCRATCH command. Typing SCR or SCRATCH deletes whatever program has been entered by the user during the current session.

The first program entered was 100 PRINT 35+5. After it has run, it should be scratched before entering the next program. Otherwise both programs will run when RUN is typed. They will run in the order of their statement numbers. For instance, if both programs are currently in the user's work area, the program with numbers 10 through 30 executes before line 100.

Both programs will run when RUN is typed:

```
>100 PRINT 35+5
>10 INPUT A,B,C,D,E
>20 LET S=(A+B+C+D+E)/5
>30 PRINT S
>RUN
?7,5,6,8,9
 7
 40
```

To avoid confusing results, a program that has been entered and run can be deleted with SCRATCH:

After entering and running:

```
>100 PRINT 35+5
>RUN
 40
```

the program is scratched:

```
>SCRATCH
```

The users work area is now cleared and another program can be entered.

The second program is entered:

```
>10 INPUT A,B,C,D,E
>20 LET S=(A+B+C+D+E)/5
>30 PRINT S
>RUN
?15,25,32,11,29
 22.4
```

Unless this program is to be run again, it can now be scratched and a third program entered.

Documenting a Program

Remarks that explain or comment can be inserted in a program with the REM statement. Any remarks typed after REM will be printed in the program listing but will not affect program execution. The remarks cannot be continued on the next line, but as many REM statements can be entered as are needed.

The sample program to average 5 numbers can be documented with several remarks:

```
>5 REM  THIS PROGRAM AVERAGES
>7 REM  5 NUMBERS
>15 REM 5 VALUES MUST BE INPUT
>25 REM S CONTAINS THE AVERAGE
```

The statement numbers determine the position of the remarks within the existing program. A list will show them in order:

List of sample program including remarks:

```
>LIST
  5 REM  THIS PROGRAM AVERAGES
  7 REM  5 NUMBERS
 10 INPUT A,B,C,D,E
 15 REM 5 VALUES MUST BE INPUT
 20 LET S=(A+B+C+D+E)/5
 25 REM S CONTAINS THE AVERAGE
 30 PRINT S
>
```

When run, the program will execute exactly as it did before the remarks were entered.

SECTION II
Essentials of BASIC

The first section introduced the user to BASIC programming. This section describes the statements needed to write a simple BASIC program. It also describes the commands used to run a program, to edit a program, and to save and manipulate library programs.

The section begins with a description of expressions used in BASIC, and the constants, variables, functions and operators used in the formation of expressions.

Subsequent sections discuss particular features of more advanced BASIC.

The simple PRINT statement and RUN command used in Section I are used again in this section prior to the explanation of the full capabilities of PRINT and RUN.

Expressions

An expression combines constants, variables, or functions with operators in an ordered sequence. When evaluated, an expression must result in a value. An expression that, when evaluated, is converted to an integer, is called an integer expression. Constants, variables, and functions represent values; operators tell the computer the type of operation to perform on these values.

Some examples of expressions are:

(P + 5)/27	P is a variable that must have been previously assigned a value. 5 and 27 are constants. The slash is the divide operator. Parentheses group those portions of the expression evaluated first. If P = 49, it is an integer expression with the value 2.
(N-(R+5))-T	N, R, and T must all have been assigned values. + and - are the add and subtract operators. The innermost parentheses enclose the part evaluated first. If N=20, R=10, and T=5, the value of the integer expression is zero.

CONSTANTS

A constant is either numeric or it is a literal string.

Numeric Constants. A numeric constant is a positive or negative decimal number including zero. It may be written in any of the following three forms:

- As an integer — a series of digits with no decimal point.
- As a fixed point number — a series of digits with one decimal point preceding, following, or embedded within the series.
- As a floating point number — an integer or fixed point number followed by the letter E and an optionally signed integer.

Examples of Integers:

```
 1234
-70
 0
```

Examples of Fixed Point Numbers:

```
 1234.
 1234.56
-.0123
```

Floating Point Numbers. In the floating point notation, the number preceding E is a magnitude that is multiplied by some power of 10. The integer after E is the exponent, that is, it is the power of 10 by which the magnitude is multiplied.

The exponent of a floating point number is used to position the decimal point. Without this notation, describing a very large or very small number would be cumbersome:

1E+35 = 100000000000000000000000000000000000

1E-35 = .00000000000000000000000000000000001

Examples of Floating-Point Numbers:

1E+23	= 1×10^{23} = 100000000000000000000000
1.0E23	(same as above)
.001E26	(same as above)
1.02E+4	= 1.02×10^{4} = 10200.
1.02E-4	= .000102

Within the computer, all these constants are represented as floating-point real numbers whose precision is 6 or 7 digits and whose size is between 10^{-77} and 10^{77}.

Literal Strings. A literal string consists of a sequence of characters in the ASCII character set enclosed within quotes. The quote itself is the only character excluded from the character string. By using an integer equivalent of the graphic character, even the quote may be included in a character string (see Strings, Section V).

Examples of Literal Strings:

```
"ABC"                   ""     (a null, empty, or zero length string)
"!!WHAT A DAY!!"        "  "   (a string with two blanks)
" X  Y   Z  "
```

Blank spaces are significant within a string.

VARIABLES

A variable is a name to which a value is assigned. This value may be changed during program execution. A reference to the variable acts as a reference to its current value. Variables are either numeric or string.

Numeric variables are a single letter (from A to Z) or a letter immediately followed by a digit (from 0 to 9):

A A0

P P5

X X9

A variable of this type always contains a numeric value that is represented in the computer by a real floating-point number. Other numeric representations can be specifically requested with the type statement (see Variable Types, Section IV). These types are integer, long floating-point, and complex.

A variable may also contain a string of characters. This type of variable is identified by a variable name consisting of a letter and $, or a letter, digit, and $:

A$ A0$

P$ P5$

The value of a string variable is always a string of characters, possibly null or zero length. String variables can be used without being declared with a DIM statement (see section V) only if the variable contains a single character.

If a variable names an array (see Arrays, Section III), it may be subscripted. When a variable is subscripted, the variable name is followed by one or two subscript values enclosed in parentheses. If there are two subscripts, they are separated by a comma. A subscript may be an integer constant or variable, or any expression that is evaluated to an integer value:

A(1)	A0(N,M)
P(1,1)	P5(Q5,N/2)
X(N+1)	X9(10,10)

A simple numeric variable and a subscripted numeric variable may have the same name with no implied relation between the two. The variable A is totally distinct from variable A(1,1).

Simple numeric variables can be used without being declared. Subscripted variables must be declared with a DIM statement (see Section III) if the array dimensions are greater than 10 rows, or 10 rows and 10 columns. The first subscript is always the row number, the second the column number. The subscript expressions must result in a value between 1 and the maximum number of rows and columns.

String arrays differ from numeric arrays in that they have only one dimension, and hence only one subscript. Also, the name of a string array and a simple string variable may not be the same (see String Arrays in Section V). Examples of subscripted string array names are:

A$(1)   A0$(N)

FUNCTIONS

A function names an operation that is performed using one or more parameter values to produce a single value result. A numeric function is identified by a three-letter name followed by one or more formal parameters enclosed in parentheses. If there is more than one, the parameters are separated by commas. The number and type of the parameters depends on the particular function. The formal parameters in the function definition are replaced by actual parameters when the function is used.

Since a function results in a single value, it can be used anywhere in an expression where a constant or variable can be used. To use a function, the function name followed by actual parameters in parentheses (known as a function call) is placed in an expression. The resulting value is used in the evaluation of the expression.

Examples of common functions:

SQR(x) where x is a numeric expression that results in a value $\geqslant 0$. When called, it returns the square root of x. For instance, if N = 2, SQR(N+2) = 2.

ABS(x) where x is any numeric expression. When called, it returns the absolute value of x. For instance, ABS(-33) = 33.

BASIC/3000 provides many built-in functions that perform common operations such as finding the sine, taking the square root, or finding the absolute value of a number. The available functions are listed in Appendix E. In addition, the user may define and name his own functions should he need to repeat a particular operation. How to write functions is described in Section VI, User-Defined Functions.

The functions described so far are numeric functions that result in a numeric value. Functions resulting in string values are also available. These are identified by a three-letter name followed by a $. String functions are described with user-defined functions in Section VI; available built-in string functions are listed in Appendix E.

OPERATORS

An operator performs a mathematical or logical operation on one or two values resulting in a single value. Generally, an operator is between two values, but there are unary operators that precede a single value. For instance, the minus sign in A - B is a binary operator that results in subtraction of the values; the minus sign in -A is a unary operator indicating that A is to be negated.

The combination of one or two operands with an operator forms an expression. The operands that appear in an expression can be constants, variables, functions, or other expressions.

Operators may be divided into types depending on the kind of operation performed. The main types are arithmetic, relational, and logical (or Boolean) operators.

The arithmetic operators are:

Operator	Operation	Example
+	Add (or if unary, no operation)	A + B or +A
-	Subtract (or if unary, negative)	A - B or -A
*	Multiply	A × B
/	Divide	A ÷ B
** or ^	Exponentiate (if ^ is used, it is changed internally to **)	A^B
MOD	Modulo; remainder from division	A - B × INT(A ÷ B) where INT(x) returns the largest integer ≤ x. If A and B are positive, A MOD B is the remainder from A ÷ B.

In an expression, the arithmetic operators cause an arithmetic operation resulting in a single numeric value.

The relational operators are:

=	Equal	A = B
<	Less than	A < B
>	Greater than	A > B
<=	Less than or equal to	A ≤ B
>=	Greater than or equal to	A ≥ B
<> or #	Not equal (if # is used, it is changed internally to <>)	A ≠ B

When relational operators are evaluated in an expression they return the value 1 if the relation is found to be true, or the value 0 if the relation is false. For instance, A = B is evaluated as 1 if A and B are equal in value, as 0 if they are unequal.

Maximum and minimum operators are:

MIN	Select the lesser of two values	A MIN B
MAX	Select the greater of two values	A MAX B

These operators are evaluated as follows:

A MIN B = A if A is less than or equal to B; = B if B is less than A

A MAX B = A if A is greater than or equal to B; = B if B is greater than A

Logical or Boolean operators are:

AND	Logical "and"	A AND B
OR	Logical "or"	A OR B
NOT	Logical complement	NOT A

Like the relational operators, the evaluation of an expression using logical operators results in the value 1 if the expression is true, the value 0 if the expression is false.

Logical operators are evaluated as follows:

A AND B	= 1 (true) if A and B are both ≠ 0; = 0 (false) if A = 0 or B = 0
A OR B	= 1 (true) if A ≠ 0 or B ≠ 0; = 0 (false) if both A and B = 0
NOT A	= 1 (true) if A = 0; = 0 (false) if A ≠ 0

A string operator is available for combining two string expressions into one:

+	Concatenation	A$ + B$

The values of A$ and B$ are joined to form a single string; the characters in B$ immediately follow the last character in A$. If A$ contains "ABC" and B$ contains "DEF", then A$ + B$ = "ABCDEF" (see Strings, Section V).

EVALUATING EXPRESSIONS

An expression is evaluated by replacing each variable with its value, evaluating any function calls, and performing the operations indicated by the operators. The order in which operations is performed is determined by the hierarchy of operators:

** (highest)

NOT

* / MOD

+ -

+ (string concatenate)

MIN MAX

Relational (=, <, >, <=, >=, <>)

AND

OR (lowest)

The operator at the highest level is performed first followed by any other operators in the hierarchy shown above. If operators are at the same level, the order is from left to right. Parentheses can be used to override this order. Operations enclosed in parentheses are performed before any operations outside the parentheses. When parentheses are nested, operations within the innermost pair are performed first.

For instance: 5 + 6*7 is evaluated as 5 + (6×7) = 47

7/14*2/5 is evaluated as ((7/14)×2)/5 = .2

If A=1, B=2, C=3, D=3.14, E=0

then:

A+B*C is evaluated as A +(B×C) = 7

A*B+C is evaluated as (A×B)+C = 5

A+B–C is evaluated as (A+B)–C = 0

(A+B)*C is evaluated as (A+B)×C = 9

A MIN B MAX C MIN D is evaluated as ((A MIN B)MAX C)MIN D = C = 3

When a unary operator immediately follows another operator of higher precedence, the unary operator assumes the same precedence as the preceding operator. For instance,

B**-B**C is evaluated as $(B^{-B})^C$ = 1/64 or .015625

In a relation, the relational operator determines whether the relation is equal to 1 (true) or 0 (false):

(A*B) < (A-C/3) is evaluated as 0 (false) since A*B=2 which is not less than A-C/3=0.

In a logical expression, other operators are evaluated first for values of zero (false) or non-zero (true). The logical operators determine whether the entire expression is equal to 0 (false) or 1 (true):

E AND A-C/3	is evaluated as 0 (false) since both terms in the expression are equal to zero (false).
A+B AND A*B	is evaluated as 1 (true) since both terms in the expression are different from zero (true).
A=B OR C=SIN(D)	is evaluated as 0 (false) since both expressions are false (0).
A OR E	is evaluated as 1 (true) since one term of the expression (A) is not equal to zero.
NOT E	is evaluated as 1 (true) since E=0.

If any ambiguity exists between the relational operator "=" and the assignment operator, the equal sign is treated as an assignment operator:

A=B=1	assigns 1 to both A and B.
A=1=B	assigns 1 to A if B equals 1, or 0 to A if B does not equal 1.

For rules governing the evaluation of relational expressions using strings, see Comparing Strings in Section V.

Statements

Statements essential to writing a program in BASIC are described here. Statements in general are described in Section I. It should be recalled that all statements must be preceded by a statement number and are terminated by pressing the return key. Statements are not executed until the program is executed with the RUN command.

Assignment Statement

This statement assigns a value to one or more variables. The value may be in the form of an expression, a constant, a string, or another variable of the same type.

Form

When the value of the expression is assigned to a single variable, the forms are:

variable = expression

LET variable = expression

When the same value is to be assigned to more than one variable, the forms are:

variable = variable = . . . = variable = expression

LET variable = variable = . . . = variable = expression

Several assignments can be made in one statement if they are separated by commas:

variable = expression, . . ., variable = expression

LET variable = expression, . . ., variable = expression

Note that the word *LET* is an optional part of the assignment statement.

Explanation

In this statement, the equal sign is an assignment operator. It does not indicate equality, but is a signal that the value on the right of assignment operator be assigned to the variable on the left. If any ambiguity exists between the relational operator "=" and the assignment operator, the equal sign is treated as an assignment operator.

When a variable to be assigned a value contains subscripts, these are evaluated first from left to right, then the expression is evaluated and the resulting value moved to the variable.

If a value is assigned to more than one variable, the assignment is made from right to left. For instance, in the statement A=B=C=2, first C is assigned the value 2, then B is assigned the current value of C, and finally A is assigned the value of B.

Examples

```
10 LET A=5.02
20 A=5.02
```

The variable A is assigned the value 5.02. Statements 10 and 20 have the same result.

```
30 X=Y7=Z=Z1=0
```

Each variable X, Y7, Z, and Z1 is set to zero. This is a simple method for initializing variables at the start of a program.

```
35 LET N=2
40 LET A[N]=N=9
```

First N is assigned the value 2 in line 35. In line 40 N is assigned the value 9, then the array element A(2) is assigned the value 9.

```
50 N=0
60 LET N=N+1
70 LET A[N]=N
```

Statements 50 through 70 set the array element A(1) to 1. By repeating statements 60 and 70, each array element can be set to the value of its subscript.

```
80 A=10.5,B=7.5
90 B$="ABC",C$=B$
```

Variable A is set to 10.5, then B is set to 7.5. The string variable B$ is assigned the value ABC, then C$ is assigned the value of B$ (or ABC).

```
100 C$=B$="ABC"
```

This statement has the same result as statement 90.

```
110 LET A=10.5,B=7.5,B$=C$="ABC"
```

Statement 110 has the same effect as the two statements 80 and 90.

REM Statement

This statement allows the insertion of a line of remarks in the listing of the program. The remarks do not affect program execution.

Form

REM any characters

Like other statements, REM must be preceded by a statement number. Unlike other statements, it cannot be continued on the next line.

Explanation

The remarks introduced by REM are saved as part of the BASIC program, and printed when the program is listed or punched. They are, however, ignored when the program is executed.

Remarks are easier to read if REM is followed by spaces, or a punctuation mark as in the examples.

Examples

```
10 REM: THIS IS AN EXAMPLE
20 REM  OF REM STATEMENTS.
30 REM -- ANY CHARACTERS MAY FOLLOW REM: "//**!!&&&,ETC.
40 REM...REM STATEMENTS ARE NOT EXECUTED
```

GOTO Statement

GOTO overrides the normal sequential order of statement execution by transferring control to a specified statement. The statement to which control transfers must be an existing statement in the current program.

Form

GOTO statement label

GOTO integer expression OF *statement label, statement label, . . .*

GOTO may have a single *statement label*, or may be multi-branched with more than one *statement label.*

If the multi-branch GOTO is used, the value of the *integer expression* determines the label in the list to which control transfers.

Explanation

If the GOTO transfers to a statement that cannot be executed (such as REM or DIM), control passes to the next sequential statement after that statement. GOTO cannot transfer into or out of a function definition (see Section VI). If it should transfer to the DEF statement, control passes to the line following the function definition.

The labels in a multi-branch GOTO are selected by numbering them sequentially starting with 1, such that the first label is selected if the value of the expression is 1, the second label if the expression equals 2, and so forth. If the value of the expression is less than 1 or greater than the number of labels in the list, then the GOTO is ignored and control transfers to the statement immediately following GOTO.

Examples

The example below shows a simple GOTO in line 200 and a multi-branch GOTO in line 600.

```
100 LET I=0
200 GOTO 600
300 PRINT I
400 REM   THE VALUE OF I IS ZERO
500 LET I=I+1
600 GOTO I+1 OF 300,500,800
700 REM   THE FINAL VALUE OF I IS 2
800 PRINT I

>RUN
 0
 2
```

When run, the program prints the initial value of I and the final value of I.

GOSUB/RETURN Statements

GOSUB transfers control to the beginning of a simple subroutine. A subroutine consists of a collection of statements that may be performed from more than one location in the program. In a simple subroutine, there is no explicit indication in the program as to which statements constitute the subroutine. A RETURN statement in the subroutine returns control to the statement following the GOSUB statement.

Form

GOSUB statement label

GOSUB integer expression OF *statement label, statement label, . . .*

RETURN

GOSUB may have a single *statement label,* or may be multi-branched with more than one *statement label.* In a multi-branch GOSUB, the particular label to which control transfers is determined by the value of the *integer expression.* The RETURN statement consists simply of the word RETURN.

Explanation

A single-branch GOSUB transfers control to the statement indicated by the label. A multi-branch GOSUB transfers to the statement label determined by the value of the integer expression. As in a multi-branch GOTO, if the value of the expression is less than 1 or greater than the length of the list, no transfer takes place. A GOSUB must not transfer into or out of a function definition (see Section VI).

When the sequence of control within the subroutine reaches a RETURN statement, control returns to the statement following the GOSUB statement.

Within a subroutine, another subroutine can be called. This is known as nesting. When a RETURN is executed, control transfers back to the statement following the last GOSUB executed. Up to ten GOSUB statements can occur without an intervening RETURN; more than this causes a terminating error.

Examples

In the first example, line 20 contains a simple GOSUB statement; the subroutine is in lines 50 through 70, with RETURN in line 70.

```
 10 LET B=90
 20 GOSUB 50
 30 PRINT "SINE OF B IS ";A
 40 GOTO 80
 50 REM: THIS IS THE START OF THE SUBROUTINE
 60 LET A=SIN(B)
 70 RETURN
 80 REM: PROGRAM CONTINUES WITH NEXT STATEMENT
>RUN
SINE OF B IS  .893992
```

The GOSUB statement can follow the subroutine to which it transfers as in the example below.

```
 10 LET B=90
 20 GOTO 100
 30 REM: THIS IS START OF SUBROUTINE
 40 LET A=SIN(B)
 50 RETURN
 60 REM: OTHER STATEMENTS CAN APPEAR HERE
 70 REM: THEY WILL NOT BE EXECUTED
 80 A=24,B=50
 90 PRINT A;B
100 GOSUB 30
110 PRINT A
120 REM: A SHOULD EQUAL .893992
130 PRINT B
140 REM: B SHOULD EQUAL 90
>RUN
 .893992
 90
```

This example shows a multi-branch GOSUB in line 20. The third subroutine executed has a nested subroutine. An IF. . .THEN statement is used in the example; should its function not be clear, see Conditional Statements below in this section.

```
  10 A=0
  20 GOSUB A+1 OF 100,150,200
  30 LET A=A+1
  40 IF A<3 THEN GOTO 20
  50 GOTO 300
  60 REM:  STATEMENT 50 BRANCHES AROUND ALL THE SUBROUTINES
 100 REM:  FIRST SUBROUTINE IN MULTIBRANCH GOSUB
 110 LET X=SQR(A+25)
 120 PRINT "X = ";X
 130 RETURN
 150 REM:  SECOND SUBROUTINE IN MULTIBRANCH GOSUB
 160 LET Y=COS(X)
 170 PRINT "Y = COSINE X = ";Y
 180 RETURN
 200 REM:  THIRD SUBROUTINE IN MULTIBRANCH GOSUB
 210 REM:  IT CONTAINS A NESTED SUBROUTINE
 220 LET Y=Y+X
 225 PRINT "Y + X = ";Y
 230 GOSUB 260
 240 RETURN
 250 REM:  STATEMENT 240 RETURNS CONTROL TO STATEMENT 30
 260 REM:  FIRST STATEMENT IN NESTED SUBROUTINE
 270 B=SIN(Y)
 280 PRINT "SINE Y = ";B
 290 RETURN
 295 REM:  STATEMENT 290 RETURNS CONTROL TO STATEMENT 240
 300 REM:  PROGRAM CONTINUES WITH NEXT STATEMENT
>RUN
X =  5
Y = COSINE X =  .283663
Y + X =  5.28366
SINE Y = -.841213
```

END/STOP Statements

The END and STOP statements are used to terminate execution of a program. Either may be used, neither is required. An END is assumed following the last line entered in the current program.

Form

END

STOP

The END statement consists of the word END; the STOP statement of the word STOP.

Explanation

Both END and STOP terminate the program run. END has a different function from STOP only when programs are segmented (see Section X, Segmentation). When END is executed in a program segment that has been called by another program with INVOKE, control returns to the statement after INVOKE.

Whenever STOP is used, the program terminates. STOP in a program called with INVOKE terminates all program execution, including any suspended programs.

Examples

These three programs are effectively the same:

```
 10 LET A=2,B=3
 20 C=A**-A**B
 30 PRINT C
>RUN
 .015625
```

```
 10 LET A=2,B=3
 20 C=A**-A**B
 30 PRINT C
 40 END
>RUN
 .015625
```

```
 10 LET A=2,B=3
 20 C=A**-A**B
 30 PRINT C
 40 STOP
>RUN
 .015625
```

When sequence is direct and the last statement in the current program is the last statement to be executed, END or STOP are optional. They have a use, however, when sequence is not direct and the last statement in the program is not the last statement to be executed:

```
100 LET A=2
120 GOSUB 140
130 END
140 LET B=A+1
150 X=A**(B**A)
160 PRINT X
170 RETURN
>RUN
 512
```

The subroutine at line 140 follows the END statement.

```
   10 LET A=2
   20 X=A**2+A
   30 PRINT X
   40 IF X<100 THEN GOTO 80
   50 PRINT "X= ";X
   60 PRINT "A= ";A
   70 STOP
   80 A=A+1
   90 GOTO 20
>RUN
  6
  12
 20
 30
 42
 56
 72
 90
 110
X=  110
A=  10
```

The STOP statement at line 70 is skipped until the value of X is equal to or exceeds 100.

Looping Statements

The looping statements FOR and NEXT allow repetition of a group of statements. The FOR statement precedes the statements to be repeated, and the NEXT statement directly follows them. The number of times the statements are repeated is determined by the value of a simple numeric variable specified in the FOR statement.

Form

FOR variable = expression TO expression

FOR variable = expression TO expression STEP expression

The *variable* is initially set to the value resulting from the *expression* after the equal sign. When the value of the *variable* passes the value of the *expression* following TO, the looping stops. If STEP is specified, the *variable* is incremented by the value resulting from the STEP *expression* each time the group of statements is repeated. This value can be positive or negative, but should not be zero. If a STEP *expression* is not specified, the variable is incremented by 1.

The NEXT statement terminates the loop:

NEXT variable

The *variable* following NEXT must be the same as the *variable* after the corresponding FOR.

Explanation

When FOR is executed, the variable is assigned an initial value resulting from the expression after the equal sign, and the final value and any step value are evaluated. Then the following steps occur:

1. The value of the FOR variable is compared to the final value; if it exceeds the final value (or is less when the STEP value is negative), control skips to the statement following NEXT.
2. All statements between the FOR statement and the NEXT statement are executed.
3. The FOR variable is incremented by 1, or if specified, by the STEP value.
4. Return to step 1.

The user should not execute the statements in a FOR loop except through a FOR statement. Transferring control into the middle of a loop can produce undesirable results.

FOR loops can be nested if one FOR loop is completely contained within another. They must not overlap.

Examples

Each time the FOR statement executes, the user inputs a value for R and the area of a circle with that radius is computed and printed:

```
  10 FOR A=1 TO 5
  20   INPUT R
  30   PRINT "AREA OF CIRCLE WITH RADIUS ";R;" IS ";3.14*R**2
  40 NEXT A
>RUN
?1
AREA OF CIRCLE WITH RADIUS  1     IS  3.14
?2
AREA OF CIRCLE WITH RADIUS  2     IS  12.56
?4
AREA OF CIRCLE WITH RADIUS  4     IS  50.24
?8
AREA OF CIRCLE WITH RADIUS  8     IS  200.96
?16
AREA OF CIRCLE WITH RADIUS  16    IS  803.84
```

The FOR loop executes six times, decreasing the value of X by 1 each time:

```
  10 FOR X=0 TO -5 STEP -1
  20   PRINT X-5
  30 NEXT X
>RUN
-5
-6
-7
-8
-9
-10
```

The first X elements of the array P(N) are assigned values. When N = X, the loop terminates. In this case, the value of X is input as 3:

```
  10 INPUT X
  20 FOR N=1 TO X
  30   LET P[N]=N+1
  40   PRINT P[N]
  50 NEXT N
>RUN
?3
 2
 3
 4
```

The examples below show legal and illegal nesting. A diagnostic is printed when an attempt is made to run the second example:

```
10 REM..THIS EXAMPLE IS LEGAL
20 FOR A=1 TO 10
30   FOR B=1 TO 5
40     LET X[A,B]=0
50   NEXT B
60 NEXT A
```

```
  10 REM.. THIS EXAMPLE IS ILLEGAL
  20 FOR A=1 TO 10
  30   FOR B=1 TO 5
  40     LET X[A,B]=0
  50   NEXT A
  60 NEXT B
>RUN
'FOR' - 'NEXT' VARIABLES DON'T MATCH IN LINE 50
```

Conditional Statements

Conditional statements are used to test for specific conditions and specify program action depending on the test result. The condition tested is a numeric expression that is considered true if the value is not zero, false if the value is zero. Conditional statements are always introduced by an IF statement; an ELSE statement may follow the IF statement. Both IF and ELSE statements may be followed by a series of statements enclosed by DO and DOEND.

Form

IF expression THEN statement label

IF expression THEN statement

IF expression THEN DO

statement

.

.

.

DOEND

An IF. . .THEN statement can be followed by an ELSE statement to specify action in case the value of the *expression* is false. Like IF, ELSE can be followed by a *statement*, a *statement label*, or a series of statements enclosed by DO. . .DOEND.

ELSE statement label

ELSE statement

ELSE DO

statement

.

.

.

DOEND

ELSE statements never appear in a program unless preceded by an IF. . .THEN statement. An ELSE statement must immediately follow an IF. . .THEN statement or the DOEND statement corresponding to an IF. . .THEN DO statement; no intervening statements (including REM) are permitted. DO. . .DOEND statements may follow only an IF. . .THEN or an ELSE statement.

The four diagrams below show all possible combinations of conditional statements. Items enclosed by [ ] are optional; one of the items enclosed by { } must be chosen. Statements immediately following THEN and ELSE are not labeled; all other statements must be labeled.

- *label IF expression THEN* { *label* | *statement* }

  [*label ELSE* { *label* | *statement* }]

- *label IF expression THEN DO*

  label statement

  ⋮

  label DOEND

  [*label ELSE* { *label* | *statement* }]

- *label IF expression THEN* { *label* | *statement* }

  label ELSE DO

  label statement

  ⋮

  label DOEND

- *label IF expression THEN DO*

  label statement

  ⋮

  label DOEND

  label ELSE DO

  label statement

  ⋮

  label DOEND

Explanation

The expression following IF is evaluated, and if true the program transfers control to the label following THEN or executes the statement following THEN. If DO follows THEN, the program executes the series of labeled statements terminated by DOEND. The program then continues. If the expression is false, control transfers immediately to the next statement or to the statement following DOEND if THEN DO was specified.

When an ELSE statement follows the IF. . .THEN statement, it determines the specific action should the IF expression be false. When the expression is true, the ELSE statement or the group of ELSE statements enclosed by DO. . .DOEND is skipped, and the program continues with the next statement after ELSE or DOEND.

A FOR statement can be specified in a DO. . .DOEND group; if so, the corresponding NEXT must be within the same DO. . .DOEND group. (See FOR. . .NEXT statement description in this section.)

IF statements are nested when an IF statement occurs within the DO. . .DOEND group of another IF statement. In such a case, each ELSE is matched with the closest preceding IF that is not itself part of another DO. . .DOEND group.

Examples

The various types of IF statement are illustrated with the following examples:

```
10 IF A=B THEN 30
20 LET A=B
30 PRINT A,B
```

If A equals B, the program skips to line 30, otherwise, it sets A equal B in line 20 and continues. In either case, line 30 is executed.

```
10 IF A=B THEN PRINT B
20 ELSE PRINT A,B
```

If A equals B, the value of B is printed, otherwise, both values are printed. The program then continues.

```
10 IF A=B THEN 100
20 ELSE 200
```

Program control transfers to line 100 if A equals B, to line 200 if not.

```
10 IF A=B THEN GOTO 100
20 ELSE GOTO 200
```

These two statements are identical in effect to the preceding two statements.

```
 10 IF A<100 THEN A=A+5
 20 ELSE DO
 30   LET X=A
 40   GOTO 100
 50 DOEND
 60 GOTO 10
100 PRINT X
```

If A is less than 100, it is increased by 5 and control skips to line 60 where control is returned to line 10. When A is equal to or greater than 100, X is set equal to A and control skips to line 100.

```
  5 INPUT A
 10 IF A<100 THEN DO
 20   A=A+1
 30   GOTO 200
 40 DOEND
 50 ELSE DO
 60   X=A
 70   A=0
 80   GOSUB 850
 90 DOEND
100 PRINT "A>=100"
120 END
200 PRINT "A=";A
210 END
850 PRINT "X=";X
851 PRINT "A=";A
852 RETURN
```

If A is less than 100, it is increased by 1 and control goes to line 200. If A is equal to or greater than 100, X is set equal to A, A is set to zero and the subroutine at line 850 is executed. The subroutine returns control to line 100.

If a value less than 100 is input for A, line 200 is executed and the program ends:

```
>RUN
?75
A= 76
```

If a value greater than 100 is input for A, the subroutine is executed, then line 100 is executed and the program terminates:

```
>RUN
?150
X= 150
A= 0
A>=100
```

The examples below illustrate nested IF. . .THEN statements.

```
  10 INPUT A,B,C
  20 IF (A+10)=(B+5) THEN DO
  30   A=B
  40   IF A>C THEN A=C
  50   ELSE C=B
  60 DOEND
  70 PRINT A,B,C
>RUN
?5,10,15
 10                 10                 10
```

With the particular values input, the first IF is true and the second IF is false. As a result both A and C are set equal to B.

```
 10 INPUT A,B,C
 20 IF A>B THEN DO
 30   IF B>C THEN DO
 40     IF C=10 THEN DO
 50       C=C+1
 60       GOTO 200
 70     DOEND
 80     ELSE GOTO 220
 90   DOEND
100   ELSE DO
110     IF C=10 THEN B=C+A
120     ELSE C=B-A
130     GOTO 180
140   DOEND
150 DOEND
160 PRINT "A<=B,A=";A
170 GOTO 230
180 PRINT "A>B,B<=C,B=";B
190 GOTO 230
200 PRINT "A>B>C,C=10"
210 GOTO 230
220 PRINT "A>B>C,C<>10,C=";C
230 END
>RUN
?10,15,20
A<B,A= 10

>RUN
?15,5,10
A>B,B<C,B= 25

>RUN
?20,15,5
A>B>C,C<>10,C= 5
```

So that nested IF statements may be easier to follow, the LIST command indents them as shown in these examples.

PRINT Statement

PRINT causes data to be output at the terminal. The data to be output is specified in a print list following PRINT.

Form

PRINT

PRINT print list

The *print list* consists of items separated by commas or semicolons. The list may be followed by a comma or a semicolon. If the list is omitted, PRINT causes a skip to the next line. Items in the list may be numeric or string expressions, special print functions for tabbing or spacing, or FOR loops to provide repeated output. The form of the FOR loop is:

(FOR statement, print list)

where the *print list* contains any items allowed in the PRINT statement list including other FOR loops. The FOR statement is described earlier in this section under the heading Looping Statements.

Explanation

The contents of the print list is printed. If there is more than one item in the print list, commas or semicolons must separate the items. The choice of a comma or semicolon affects the output format.

The output line is divided into five consecutive fields: four of 15 characters and one of 12 characters, for a total of 72 characters. When a comma separates items, each item is printed starting at the beginning of a field. When a semicolon separates items, each item is printed immediately following the preceding item. In either case, if there is not enough room left in the line to print the entire item, printing of the item begins on the next line.

The separator between items can be omitted if one or both of the items is a quoted string. In this case, a semicolon is inserted automatically.

A carriage return and linefeed are output after PRINT has executed, unless the output list is terminated by a comma or semicolon. In this case, the next PRINT statement begins on the same line.

If an expression appears in the print list, it is evaluated and the result is printed. Any variable must have been assigned a value before it is printed. Each character between quotes in a string constant is printed, excluding quotes.

If a FOR loop is included in the print list, each item in the print list associated with the FOR statement is printed once for each time the FOR loop is executed.

Numeric values are left justified in a field whose width is determined by the magnitude of the number. The smallest field is six characters. Numeric output format is discussed in detail below.

Examples

When items are separated by commas, they are printed in up to five fields per line; separated by semicolons, they directly follow one another. In the example below, the items are numeric, so each item is assigned a minimum of six characters.

```
 10 LET A=B=C=D=E=15
 20 LET A1=B1=C1=D1=E1=20
 30 PRINT A,B,C1,C
 40 PRINT A;B;C1;C;D;E;A1;D1;E1
 50 PRINT A,B;C,D
>RUN
15              15              20              15
15    15    20    15    15    15    20    20    20
15              15    15        15
```

In the example below, a DIM statement is used to specify the number of characters in each string; if omitted, the strings are assumed to have only one character.

```
 10 DIM B$[3],C$[3]
 20 C$=B$="ABC"
 30 PRINT B$,C$
>RUN
ABC             ABC
```

In the example below, the first PRINT statement evaluates and then prints three expressions. The second PRINT skips a line. The third and fourth PRINT statements combine a string constant with a numeric expression. No fields are used in the print line for string constants unless a comma appears as separator. The fourth PRINT statement prints output on the same line as the third because the third statement is terminated by a comma.

```
  10 LET A=B=C=D=E=15
  20 LET A1=B1=C1=D1=E1=20
  30 PRINT A*B,B/C/D1+30,A+B
  40 PRINT
  50 PRINT "A*B =";A*B,
  60 PRINT "THE SUM OF A AND B IS";A+B
>RUN
 225              30.05           30

A*B = 225       THE SUM OF A AND B IS 30
```

A FOR statement can be specified in a print list with its own print list, all included within parentheses:

```
  10 FOR I=1 TO 3
  20   INPUT R
  30   A[I]=3.14*R**2
  40 NEXT I
  50 PRINT (FOR I=1 TO 3,A[I])
>RUN
?2
?3
?4
 12.56            28.26           50.24
```

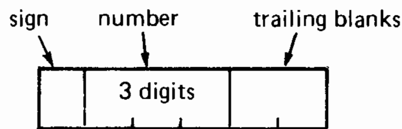
Note that NEXT is not needed when the FOR statement is included in a print list.

NUMERIC OUTPUT FORMATS

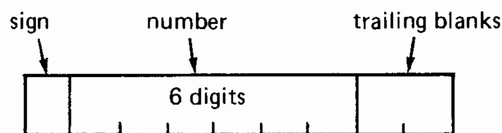
Numeric quantities are left justified in a field whose width is determined by the magnitude of the item. The width includes a position at the left of the number for a possible sign and at least one position to the right containing blanks. The width is always a multiple of three; the minimum width is six characters.

Integers

An integer with a magnitude less than 1000 requires a field width of six characters:



An integer with a magnitude between 1000 and 999999 inclusive requires a field width of nine characters:



Examples of integers:

The integers below are less than 1000 and greater than -1000:

```
 10 PRINT 1;999;30;-300;+295
>RUN
 1      999   30   -300   295
```

These integers are between 1000 and 999999 or between -1000 and -999999:

```
 10 PRINT 1000;+32751;-999999;45678
>RUN
 1000     32751   -999999   45678
```

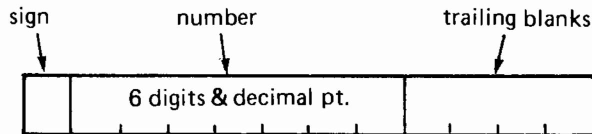
These integers are mixed in magnitude, but none are greater than 999999 or less than -999999:

```
 10 PRINT 1;1000;999;+32751;20;-999999;-300;45678;+296;5000
>RUN
 1      1000     999   32751    20   -999999  -300   45678  296
 5000
```

If an integer has a negative sign it is printed; a positive sign is not printed.

Fixed-Point Numbers

A fixed point number requires a field width of 12 positions. If the magnitude of the number is greater than or equal to .09999995 and less than 999999.5, or is less than .1 but can be printed with six significant digits, the number is printed as a fixed-point number with a sign. Trailing zeros are not printed, but a trailing decimal point is printed to show the number is not exact. The number is left-justified in the field with trailing blanks. The sign is printed only if it is negative.

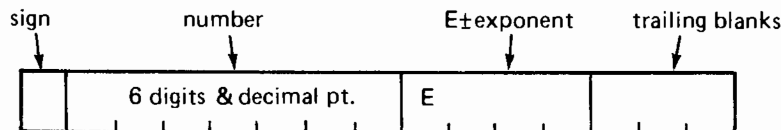


Examples of fixed-point numbers:

```
 10 PRINT 999999.4;.09999996;.000044
>RUN
 999999.      .1           .000044
```

Floating-Point Numbers

Any number, integer or fixed-point, with a magnitude greater than the magnitude of the numbers presented above, is printed as a floating-point number using a total field width of 15 positions:



Examples of floating-point numbers:

```
 10 PRINT 2345678;.0000044
>RUN
 2.34568E+06     4.40000E-06

 10 PRINT 23456789;.00000044
>RUN
 2.34568E+07     4.40000E-07

 10 PRINT .00003943;.0000257895
>RUN
 3.94300E-05     2.57895E-05
```

PRINT FUNCTIONS

These print functions may be included in a PRINT statement print list. A comma after any print function is treated as a semicolon.

TAB Function

The form of the tabulation function is:

TAB(integer expression)

The print position is moved to the column specified by the *integer expression*. Print positions are numbered from 0 to 71. If the print position must be moved to the left because the integer expression is less than the current position, nothing is done. If the expression is greater than 71, the print position is moved to the beginning of the next line.

SPA Function

The form of the spacing function is:

SPA(integer expression)

Blanks are printed for the number of spaces indicated by the *integer expression.* Nothing occurs when the expression is zero or negative. If the number of spaces will not fit on the current line, or the expression exceeds 71, a carriage return and line feed is generated.

The limit of 71 on TAB and SPA expressions does not apply to PRINT USING (see Section IX).

LIN Function

The form of the line skip function is:

LIN(integer expression)

The terminal performs a carriage return and as many line feeds as are specified in the expression. If the value is negative, the absolute value of the expression is used for the number of line feeds; no carriage return is generated. Normally, a carriage return and one line feed is performed at the end of a PRINT statement unless there is a trailing comma or semicolon.

CTL Function

The form of the carriage control function is:

CTL(integer expression)

All characters following the CTL in the PRINT statement are printed immediately using the integer expression as the carriage control. The CTL function is useful primarily when the output device is a line printer. The carriage control codes are listed below.

Carriage Control Codes

Decimal Code	Carriage Action
32	Single-space
43	Carriage return, no line feed
48	Double-space
49	Page eject (form feed)
64	Post-spacing
65	Pre-spacing
66	Single-space, with auto page eject (60 lines/pg)
67	Single-space, without auto page eject (66 lines/pg)
128+nn	Space nn lines (no automatic page eject). nn=1 thru 63 (i.e., codes 129 thru 191).
192	Page eject (*ftc #1)
193	Skip to bottom of form (*ftc #2)
194	Single-spacing, with auto page eject (*ftc #3)
195	Single-space on next odd-numbered line, with auto page eject (*ftc #4)
196	Triple-space, with auto page eject (*ftc #5)
197	Space 1/2 page, with auto page eject (*ftc #6)
198	Space 1/4 page, with auto page eject (*ftc #7)
199	Space 1/6 page, with auto page eject (*ftc #8)
208	No carriage return, no line feed.
256	Post-spacing
257	Pre-spacing
258	Single-space, with auto page eject (60 lines/pg)
259	Single-space, without auto page eject (66 lines/pg)
	*Format Tape Channel number

Examples of Print Functions

The TAB, SPA, and LIN functions are illustrated below:

```
  10 PRINT TAB(8);" TITLE:PRINT HEADING";SPA(10);"SUMMARY REPORT";
  20 PRINT LIN(3);" DETAIL LINES"
>RUN
         TITLE:PRINT HEADING          SUMMARY REPORT


 DETAIL LINES
```

The LIN function can generally be used to provide double or triple spacing, to suppress spacing, or to provide a line feed. For instance,

Double Space	LIN(2)
Suppress Spacing	LIN(0)
Line Feed only	LIN(*-integer expression*)

```
  10 PRINT "ABC";LIN(-1);"DEF";LIN(2);"GHI"
>RUN
ABC
   DEF

GHI
```

Some frequently used carriage control characters are:

Double Space	CTL(48)
Page Eject	CTL(49)
Suppress Spacing	CTL(43)

The decimal numbers associated with the carriage control characters are used as the integer expression in the CTL function. To illustrate:

```
  10 LET P=1,X=500
  20 PRINT CTL(49),"PAGE NO";P
  30 PRINT CTL(48),"DETAIL LINE"
  40 PRINT TAB(15),X,CTL(43);
  50 PRINT TAB(10),"X="
>RUN
```

After ejecting to the top of a new page, the print items are output as:

```
PAGE NO 1


DETAIL LINE
          X=    500
```

READ/DATA/RESTORE Statements

Together, the READ, DATA, and RESTORE statements provide a means to input data to a BASIC/3000 program. The READ statement reads data specified in DATA statements into variables specified in the READ statement. RESTORE allows the same data to be read again.

Form

READ item list

The items in the *item list* are either variables or FOR loops. Items are separated by commas. A FOR loop has the form:

(FOR statement, item list)

where the *item list* contains variables or FOR loops separated by commas.

DATA constant, constant,. . .

The *constants* are either numeric or string. Constants in the DATA statement are assigned to variables in the READ statement according to their order: the first constant to the first variable, the second to the second and so forth.

RESTORE

RESTORE label

The *label* identifies a DATA statement.

Explanation

When a READ statement is executed, each variable is assigned a new value from the constant list in a DATA statement. RESTORE allows the first constant to be assigned again when READ is next executed or, if a label is specified, the first constant in the specified DATA statement.

More than one DATA statement can be specified. All the constants in the combined DATA statements comprise a data list. The list starts with the DATA statement having the lowest statement label and continues to the statement with the highest label. DATA statements can be anywhere in the program; they need not precede the READ statement, nor need they be consecutive.

If a variable is numeric, the next item in the data list must be numeric; if a variable is a string, the next item in the data list must be a string constant. It is possible to determine the type of the next item with the TYP function (see Section VIII).

If the READ statement contains a FOR statement, the items following the FOR statement within parentheses are assigned values once for each time the FOR statement is executed. The FOR variable can be used in the item list, as can further FOR statements.

A pointer is kept in the data list showing which constant is the next to be assigned to a variable. This pointer starts at the first DATA statement and is advanced consecutively through the data list as constants are assigned. The RESTORE statement can be used to access data constants in a non-serial manner by specifying a particular DATA statement to which the pointer is to be moved.

When the RESTORE statement specifies a label, the pointer is moved to the first constant in the specified statement. If the statement is not a DATA statement, the pointer is moved to the first following DATA statement. When no label is specified, the pointer is restored to the first constant of the first DATA statement in the program.

Examples

The data in statement 10 is read in statement 20 and printed in statement 30:

```
  10 DATA 3,5,7
  20 READ A,B,C
  30 PRINT A,B,C
>RUN
 3                  5                   7
```

Note the use of RESTORE in this example. It permits the second READ to read the same data into a second set of variables:

```
   5 DIM A$[3],B$[3]
  10 DATA 3,5,7
  20 READ A,B,C
  30 READ A$,B$
  40 DATA "ABC","DEF"
  50 RESTORE
  60 READ D,E,F
  70 PRINT A$+B$,A;B;C;D;E;F
>RUN
ABCDEF              3       5       7       3       5       7
```

In the following examples, the data from three DATA statements is read into an 8-element array variable and a simple variable. The same data is then restored and read into three simple variables.

```
  10 DATA 3,5,7
  20 DATA 9,11,13
  30 DATA 15,17,19
  40 READ (FOR I=1 TO 8,C[I]),D
  50 PRINT (FOR I=1 TO 8, C[I]),D
>RUN
 3               5               7               9               11
 13              15              17              19
```

```
  10 DATA 3,5,7
  20 DATA 9,11,13
  30 DATA 15,17,19
  40 READ (FOR I=1 TO 8,C[I]),D
  50 PRINT (FOR I=1 TO 8,C[I]),D
  60 RESTORE
  70 READ A
  80 RESTORE 20
  90 READ B
 100 RESTORE 30
 110 READ C
 120 PRINT A,B,C
>RUN
 3               5               7               9               11
 13              15              17              19
 3               9               15
```

INPUT Statement

The INPUT statement allows the user to input data to his program from the terminal. INPUT has options that allow the user to save excess input and to print prompting strings before input. FOR loops may be included in the item list associated with INPUT.

Form

INPUT

INPUT item list

The items in the *item list* may be variables, string constants, or FOR loops. Items are separated by commas. FOR loops have the form:

(FOR statement, item list)

where the *item list* contains variables or FOR loops separated by commas.

A colon (:) may precede or follow the INPUT *item list*. When a colon follows the list, excess input is saved in a buffer; when a colon precedes the list, input is assigned from the buffer before it is requested from the user at the terminal.

An INPUT statement with no *item list* clears the input buffer; INPUT followed only by a colon fills the buffer.

Explanation

When an INPUT statement is executed, a question mark (?) is printed at the terminal and the program waits for the user to type his input. The input is in the form of constants separated by commas. If an insufficient number of constants is typed, the program responds with two question marks (??). This requests the user to input more constants. The type of data item, numeric or string, must match the type of variable it is destined for.

Like the READ and PRINT statements, the INPUT statement can include any number of FOR loops. Each time a FOR statement is executed, the user inputs a constant to match the variables in the item list associated with the FOR statement.

Numeric Constants. Numeric constants always begin with the first non-blank character preceding the comma or the end of the line.

String Constants. A string may be unquoted, in which case it begins with the first non-blank character and ends with the last non-blank character in the line. It may not contain quotation marks. A string may also be quoted, in which case it is delimited on each side by quotes and is followed either by a comma or the end of the line.

The INPUT statement can be requested to print a string constant instead of a question mark by placing the string constant immediately before a variable. When the value for the variable is needed, the string is printed instead of the usual question mark. Any number of these request strings can be included in the variable list.

Examples

```
  10 DIM C$[25]
  20 INPUT A,B,C$
  30 X=A*B**2
  40 PRINT C$;X
>RUN
?2,5,"X=A TIMES B SQUARED,  X="
X=A TIMES B SQUARED,  X= 50
```

```
  10 INPUT "INPUT VALUE OF RADIUS  ",R
  20 X=3.14*R**2
  30 PRINT "AREA OF X =",X
>RUN
INPUT VALUE OF RADIUS  25
AREA OF X =     1962.5
```

This example illustrates the various prompts for input:

```
  10 INPUT A,"NUMBER?",B,C
  20 PRINT A,B,C
>RUN
?15
NUMBER?63.5
??7
 15             63.5            7
```

If all input values are entered at one time, only the first prompt is used:

```
  10 INPUT A,"NUMBER?",B,C
  20 PRINT A,B,C
>RUN
?15,63.5,7
 15             63.5           7
```

The examples below illustrate FOR loops in the INPUT item list:

```
  10 INPUT (FOR I=1 TO 5 STEP 2,A[I])
  20 PRINT (FOR I=1 TO 5 STEP 2,A[I])
>RUN
?1,3,5
 1              3              5
```

```
  10 INPUT N,(FOR K=1 TO N,"WHAT'S NEXT?",B[K])
  20 PRINT (FOR K=1 TO N,B[K])
>RUN
?3
WHAT'S NEXT?1
WHAT'S NEXT?2
WHAT'S NEXT?3
 1              2              3
```

```
  10 INPUT N,(FOR S1=1 TO N,(FOR I=1 TO N,C[S1,I]))
  20 PRINT (FOR S1=1 TO N,(FOR I=1 TO N,C[S1,I]))
>RUN
?2
??1,2,3,4
 1              2              3              4
```

The example below illustrates the use of the colon (:) to save input in the buffer, and to assign input from the buffer. A colon following the input list saves the buffer; a colon preceding the input list assigns values from the buffer.

In this example, four input values are placed in the buffer. However, following line 20 the buffer is cleared because there is no colon after E. Another value must be input for F.

```
  10 INPUT A,B:
  20 INPUT :E
  30 INPUT :F
  40 PRINT A,B,E,F
>RUN
?1,2,3,4
??9
 1              2              3              9
```

By putting a colon after E as well as before it, the entire buffer is saved:

```
  10 INPUT A,B:
  20 INPUT :E:
  30 INPUT :F
  40 PRINT A,B,E,F
>RUN
?1,2,3,4
 1              2              3              4
```

BUF FUNCTION

The BUF function is used in conjunction with INPUT to determine the type of the next data item in the buffer. The form is:

BUF(X)

The parameter X has no meaning; any expression can replace X as the actual parameter. The results of executing BUF(X) are:

Value of BUF(X)	Next Item in Buffer
1	real
2	string
4	no data in buffer
5	integer
6	long
7	complex

BUF(X) will not return the value 3.

Example

```
 10 INPUT :
 20 IF BUF(0)=4 THEN GOTO 190
 30 IF BUF(0)=5 THEN DO
 40   INPUT :A:
 50   PRINT "INTEGER A=";A
 60   GOTO 20
 70 DOEND
 80 IF BUF(0)=1 THEN DO
 90   INPUT :B:
100   PRINT "REAL NO =";B
110   GOTO 20
120 DOEND
130 IF BUF(0)=2 THEN DO
140   INPUT :C$:
150   PRINT "STRING C=";C$
160   GOTO 20
170 DOEND
180 GOTO 20
190 PRINT "END OF BUFFER"
```

When run, the user can input any number of constants and they will be kept in the input buffer. This example assumes that no long or complex numbers will be input.

```
>RUN
?1.3,"X",576,35.2,66.6,75,"A","C"
REAL NO = 1.3
STRING C=X
INTEGER A= 576
REAL NO = 35.2
REAL NO = 66.6
INTEGER A= 75
STRING C=A
STRING C=C
END OF BUFFER
```

ENTER Statement

The ENTER statement provides the program with more control over the input operation. The statement can limit the amount of time allowed to input data from the input device (e.g., terminal), provide the program with the actual input time, indicate whether the data is of the correct type, and return logical device number of the user's terminal.

Form

There are three forms of the ENTER statement:

ENTER # terminal variable

ENTER time limit expression, actual time variable, input variable

ENTER # terminal variable, time limit expression, actual time variable, input variable

The *terminal variable* after # is used to return the logical device number of the terminal; the *time limit expression* specifies the time allowed for input; the *actual time variable* is assigned the actual time used; and the *input variable* is assigned the value typed in.

Explanation

The first form sets the terminal variable equal to the user's terminal logical device number.

The time limit expression specifies the length of time, in seconds, that the user is allowed to enter his input. The value must be in the range 1 to 255. If it is greater, 255 is used; if it is less, 1 is used.

The actual time variable is set to the approximate time, in seconds, that the user takes to respond. If an improper input is typed, this value is negated. If the user fails to respond within the allotted time, this variable is set to -256.

Only one value can be typed in for each ENTER statement and it is assigned to the input variable. A string should not be entered enclosed in quotes, but it may contain quotes. A string that is too long is truncated on the right.

The ENTER statement differs from the INPUT statement in that a "?" is not printed on the user terminal and the system returns to the program if the user does not respond within a specified time limit (there is no time limit on INPUT). Also, the program does not generate a linefeed after the user types in a carriage return.

Examples

```
  10 DIM C$[25]
  20 ENTER #A
  30 PRINT "TERMINAL NO.=";A
  40 PRINT "YOU HAVE 1 MINUTE TO TYPE 25 CHARACTERS FOR C$"
  50 ENTER 60,B,C$
  60 PRINT LIN(1);"ACTUAL TIME=";B
  70 PRINT C$
  80 PRINT LIN(1);"TYPE VALUE FOR C"
  90 ENTER #A,60,B,C
 100 PRINT LIN(1);"ACTUAL TIME=";B
 110 PRINT C
>RUN
TERMINAL NO.= 17
YOU HAVE 1 MINUTE TO TYPE 25 CHARACTERS FOR C$
EMBEDDED "QUOTES" O.K.
ACTUAL TIME= 13.41
EMBEDDED "QUOTES" O.K.

TYPE VALUE FOR C
25.7E-8
ACTUAL TIME= 6.62
 2.57000E-07
```

The system enters the logical terminal number in the variable A as a result of line 20; A can then be referenced as in line 30. Since ENTER does not provide a prompt character, it is useful to print some form of prompt particularly because there is a time limit on the input.

Note that the system does not provide a linefeed after input. It is therefore essential, if any output is to be printed after the input line, to provide a linefeed (use LIN function) within the PRINT statement. Without this linefeed, a subsequent output line overprints the input line.

A common use of ENTER is to test students:

```
  10 PRINT "WHAT IS .25 TIMES 75?"
  20 ENTER 30,T,X
  30 IF X=.25*75 THEN GOTO 70
  40 PRINT LIN(1),"SORRY,THE CORRECT ANSWER IS";.25*75
  50 PRINT "TRY THE NEXT PROBLEM"
  60 GOTO 80
  70 PRINT LIN(1);"CORRECT,YOU ANSWERED IN";T;"SECONDS"
  80 REM..THE NEXT PROBLEM COULD START HERE
>RUN
WHAT IS .25 TIMES 75?
18.75
CORRECT,YOU ANSWERED IN 3.35        SECONDS
```

> BASIC

When a BASIC/3000 program is waiting for input at the terminal as a result of an INPUT or ENTER statement, the user can interrupt input and request a new level of the BASIC/3000 Interpreter by typing > BASIC.

The computer returns a greater than sign (>) to prompt for other BASIC statements or commands.

The previous program is suspended until the user types EXIT. EXIT in this case returns control to the INPUT or ENTER statement in the previous program. The computer types two question marks (??) to signal that it is waiting for further input.

Example

```
 10 PRINT "WHAT IS THE SQUARE ROOT OF 94?"
 20 INPUT I
>RUN
WHAT IS THE SQUARE ROOT OF 94?
?>BASIC
BASIC 01.0
>10 PRINT SQR(94)
>RUN
 9.69536

>EXIT


??9.69536

>
```

The user responds to the INPUT prompt signal with > BASIC. He can then enter and run another program. EXIT returns control to the original program. He now enters the value he got as a result of the program run in > BASIC.

When BASIC/3000 is entered with > BASIC, it cannot be entered again in the same way. That is, there is no nesting of this feature.

Commands

So far we have used a set of commands (LIST, RUN, SCRATCH) for simple program manipulation. Both LIST and RUN have parameters and functions other than were illustrated. The full capability of commands used to run a program, to edit a program, and to save a program in the library are described here. The commands are:

RUN

The Editing Commands:

LIST

SCRATCH

DELETE

RENUMBER

LENGTH

Library Commands:

NAME

SAVE

GET

APPEND

PURGE

CATALOG

Commands in general are described in Section I. It should be recalled here that commands do not have labels; they are entered directly after the > prompt character and are executed immediately. Unlike statements, commands may not contain embedded blanks except between parameters. Some commands may be abbreviated.

Certain conventions are used in the command description:

UPPER-CASE	Key words that must be spelled correctly
lower-case	Words defined by the user
[ ]	Enclose optional items
{ }	Enclose required items
|	Separates alternatives, one of which must be chosen
. . .	Indicate the preceding item may be repeated

In the command descriptions, certain keywords are used:

programname	a BASIC/3000 program file
filename	a non-BASIC/3000 file
asciifile	an MPE/3000 ASCII file

Key word parameters may be in any order.

RUN

The RUN command executes a BASIC/3000 program; the form is

RUN [programname] [, label] [, OUT=asciifile] [, NOWARN] [, FREQ] [, NOECHO]

If *programname* is specified, the named program is retrieved from the user's library and made the current program. Any program previously in the user's work area is scratched. The current program then is executed. Any traces and breakpoints are deleted. (Traces and breakpoints are described in Section VII, Debugging.)

If *label* is specified, execution starts at the first executable statement at or after the label number. The starting statement must not be within a function definition. If the label specifies a DEF statement, execution begins at the first executable statement following the function definition.

OUT=*asciifile* diverts all printed output and trace information to the specified ASCII file.

NOWARN suppresses warning messages.

FREQ causes a table to be printed following program execution that summarizes the usage of all statements in all programs that are part of the run. There may be more than one program in a run when segmentation is used (see Section X, Segmentation).

NOECHO suppresses printing of program input when the input and list files are not on the same device.

Examples of RUN

The program below is the current program:

```
 10 DATA 3,5,7
 20 DATA 9,11,13
 30 DATA 15,17,19
 40 READ (FOR I=1 TO 8,C[I]),D
 50 PRINT (FOR I=1 TO 8,C[I]),D
 60 RESTORE
 70 READ A
 80 RESTORE 20
 90 READ B
100 RESTORE 30
110 READ C
120 PRINT A,B,C
```

First the entire program is run, then it is run starting at line 60:

```
>RUN
 3              5              7              9              11
 13             15             17             19
 3              9              15

>RUN,60
 3              9              15
```

Next the same program is run with a frequency table:

```
>RUN,FREQ
 3                   5                   7                   9                   11
 13                  15                  17                  19
 3                   9                   15


FREQUENCY TABLE

TOTAL STATEMENTS = 12
TOTAL TIME = .297 SECONDS

        FREQUENCY       EXECUTION TIME
LABEL COUNT PCT      AVE    TOTAL   PCT
  10      1    8     .001    .001     0
  20      1    8     .000    .000     0
  30      1    8     .001    .001     0
  40      1    8     .022    .022     7
  50      1    8     .160    .160    54
  60      1    8     .001    .001     0
  70      1    8     .002    .002     1
  80      1    8     .000    .000     0
  90      1    8     .002    .002     1
 100      1    8     .001    .001     0
 110      1    8     .002    .002     1
 120      1    8     .000    .000     0
SYSTEM OVERHEAD              .105    35
```

Editing Commands

The editing commands always affect the current program, that is, the program that is currently being entered at the terminal.

LIST

The LIST command lists all or part of the current program; the form is

LIST [first [- last]] [, OUT=asciifile] [, RECSIZE=number] [,NONAME]

where *first* and *last* specify the range of statements to be listed, and *asciifile* specifies the ASCII file to which the list is diverted. If RECSIZE is specified, *number* specifies the number of characters per record for the list file. If *NONAME* is specified, the program name is not listed; this is useful when listing programs to be read back with the XEQ command. The default parameters are the normal list file and a record size of 72 characters per record. If neither *first* nor *last* is specified, the entire program is listed. If only *first* is specified, just that statement is listed.

Examples

```
>LIST
```

The entire current program is listed at the terminal.

```
>LIST 1-100,OUT=FASTFILE,RECSIZE=130
```

Statements 1 through 100 of the current program are listed on the file FASTFILE with a record size of 130.

Note that a listing can be stopped by pressing the CTRL Y key. The user is returned to BASIC control.

SCRATCH

The SCRATCH command deletes the entire current program and its name; the form is

SCRATCH | *SCR*

SCRATCH also clears traces and breakpoints. (Traces and breakpoints are described in Section VII, Debugging).

Example

```
>SCR
```

The current program is deleted, and a new current program can be entered in the user's work area.

DELETE

The DELETE command deletes one or more specified statements; the form is

{*DELETE* | *DEL*} *first* [- *last*] [, *first* [- *last*]] . . .

where *first* and *last* are statement labels; the statements referenced by the parameters are deleted from the program. Each *first-last* pair specifies a range of statements which are to be deleted. If a *first* is given without a *last*, only the one statement is deleted.

Example

```
>DEL 45,75,400-700
```

Statements 45, 75, and all statements from 400 through 700 inclusive are deleted from the user's current program.

RENUMBER

The RENUMBER command allows the user to renumber any of the statements in the current program; the form is

{ *RENUM* | *RENUMBER* } [*newfirst* [, *delta* [, *oldfirst* [- *oldlast*]]]]

oldfirst and *oldlast* specify the range of original statements to be renumbered (defaults are 1 — 9999). If only *oldfirst* is specified, the default for *oldlast* is 9999. The first of these statements is assigned the number *newfirst* (default is 10) and each of the remainder is assigned a statement number *delta* greater than its predecessor (default for *delta* is 10). Any statement in the program which references a renumbered statement is changed as required for consistency.

Examples

```
>RENUMBER
```

The statements in the current program are renumbered in increments of 10 starting with statement number 10.

```
>RENUM 5,5,1-890
```

The old statement numbers 1 through 890 are renumbered starting with 5 and increasing by 5.

LENGTH

The LENGTH command reports the size of the current program; the form is

LENGTH | *LEN*

The length of the current program (in 16-bit words) is printed

Example

```
>LENGTH
```

The length of the current program is printed.

Examples Using Editing Commands

The user inputs a program; mistakes within lines are corrected with CTRL H.

```
>10 INPUG\T A,B,C,D,E
>20 REM..INPUT 5 VALUES
>30 LET S=(A@\+B+C+D+E)?\/5
>40 REM..S=AVERAGE OF 5 INPUT VALUES
>50 PRINT S
```

LIST correctly lists the program:

```
>LIST
  10 INPUT A,B,C,D,E
  20 REM..INPUT 5 VALUES
  30 LET S=(A+B+C+D+E)/5
  40 REM..S=AVERAGE OF 5 INPUT VALUES
  50 PRINT S
```

LENGTH gives the length in computer words:

```
>LENGTH
53 WORDS.
```

The remark lines are deleted and the program is listed:

```
>DELETE 20,40
>LIST
  10 INPUT A,B,C,D,E
  30 LET S=(A+B+C+D+E)/5
  50 PRINT S
```

Next, the program is renumbered and listed again:

```
>RENUMBER
>LIST
  10 INPUT A,B,C,D,E
  20 LET S=(A+B+C+D+E)/5
  30 PRINT S
```

The program is scratched. When LIST is now specified, there is no current program; the computer returns a ">" to prompt for further entries:

```
>SCRATCH
>LIST
>
```

Library Commands

When a current program is complete, and if it is to be used again, it should be saved in the user's library. A copy of the current program identified by a name is kept in the library when the program is saved. The current program is not affected; it remains the current program until log off, or until it is scratched with the SCRATCH command.

When a program is saved, it must be given a name either with the NAME or SAVE command. The program name is used to get, to append, or to purge a program in the user's group library. The name must be unique among names in a particular user's group library, but it may be duplicated in other groups. A catalog of the programs and files contained in the user's library may be requested with the CATALOG command.

NAME

The NAME command assigns a name to the current program; the form is

NAME programname

The *programname* specified is assigned to the current program. The *programname* can be any combination of eight alphabetic and numeric characters, beginning with an alphabetic character.

Example

```
>NAME PROGX
```

The current program is assigned the name PROGX.

SAVE

The SAVE command stores a copy of the current program in the user's library; the form is

SAVE [programname] [!] [, FAST] [, RUNONLY]

If *programname* is specified, that name is given to the saved copy, but not to the current program. If *programname* is omitted, the name of the current program is assumed; in this case, the program must have been named before it can be saved. If there is no file with the same name in the user's library, a new file is created and a copy of the current program is stored in it. If a file with the same name already exists in the library, the SAVE command is rejected unless the exclamation mark is specified, in which case the original file is purged and a new file created.

FAST causes the program to be saved in pseudo-compiled form so that it can be RUN more quickly. It also ensures that the program is valid (matching FOR-NEXT pairs, etc.).

A program saved for RUNONLY is assumed to be free of errors and ready for execution. When a RUNONLY program is brought into the user's work area with GET, certain commands are illegal until a SCRATCH or another GET. For instance, a RUNONLY program cannot be listed or modified. The only commands legal when a RUNONLY program is current are:

ABORT
CATALOG
CREATE
DUMP
EXIT
GET
KEY
PURGE
RESUME or GO
RUN
SCRATCH
SPOOL
SYSTEM
TAPE
XEQ

Examples

```
>SAVE PROGX
```

The name PROGX is assigned to the copy of the current program that is saved in the user's library.

```
>NAME PROGX
>SAVE
```

The current program is given the name PROGX, and then a copy is saved in the user's library.

```
>SAVE PROGX!,FAST,RUNONLY
```

A copy of the current program is assigned the name PROGX and stored in the user's library; any other program with the name PROGX is purged from the library. The program is saved in pseudo-compiled form and, if retrieved as the current program, only commands legal with RUNONLY can be used.

GET

The GET command loads a specified BASIC/3000 program into the user's working space; the form is

GET programname

where *programname* is the name of a program to replace the current program. GET deletes all traces and breakpoints.

Example

```
>GET SEARCH
```

SEARCH is a program saved in the user's library. It is now also available in the user's work area replacing any previous program in that area.

PURGE

The PURGE command removes a file or program from the user's library; the form is

PURGE {*basicfile* | *programname* | *filename*}

The file or program specified is deleted from the user's library; it is not recoverable once it has been purged.

Example

```
>PURGE PROGX
```

PROGX is a file or program in the user's library. It is no longer available to the user and its name may be assigned to another file or program.

APPEND

The APPEND command appends a specified program to the user's current program; the form is

APPEND programname

The program specified is appended to the end of the current program. The last sequence number of the current program must be smaller than the first sequence number of the appended program. Programs which have been saved in pseudo-compiled form (see SAVE command) and RUNONLY programs cannot be appended.

Example

```
>APPEND PROGX
```

PROGX is a program saved in the user's library. It is appended to the program currently in the user's work area.

CATALOG

The CATALOG command provides a list of programs or files specified by the user. The list includes the program or file name, the type, the number of logical records, and if desired, the record width.

The form is:

{*CAT* | *CATALOG*} [*fileset*] *[,ALL] [,RECSIZE] [,OUT=asciifile] [,START=filename]*

where:

fileset	one or more files or programs referenced by file name, group name, and/or account name. When *fileset* is omitted, all the files in the user's log-on group are listed. (See the next page for a full description of *fileset*.)
ALL	all ASCII and Binary files are included in the list; if ALL is omitted, only BASIC files and programs are listed.
RECSIZE	requests the record width for each file. If RECSIZE is omitted, record width is not listed.

OUT=*asciifile*	the file listing is diverted to the specified ASCII file; if OUT is omitted, the list is on the list device (e.g., the terminal).
START=*filename*	the listing starts with the specified file name.

For each file listed, the file name, the type (BF for BASIC file, SP for saved program, FP for fast saved program, A for ASCII, B for Binary) and the number of records in the file are listed. The record width is listed if RECSIZE is specified; the width is in bytes for ASCII files, in words otherwise. The listing is printed in as many columns as will fit across the width of the list device.

Output can be stopped with CTRL Y, as with the LIST command.

The fileset parameter has three fields that allow the user to request descriptions of one file alone, or various sets of files. The filename field indicates a specific file or all files within the units designated by the other fields. The group field denotes the group to which the files belong. The account field denotes the account to which the group belongs, or it may specify all accounts in the system. To specify all files, groups, or accounts, the user enters the character @ in the appropriate field. The three fields are separated by periods.

The table below shows the possible combination of entries in *fileset*:

File Field	Group Field	Account Field	Entry Example	Meaning
filename	groupname	accountname	FILE.GROUP.ACCT	The file named, in the group and account designated.
filename	groupname		FILE.GROUP	The file named, in the group designated under the log-on account.
filename			FILE	The file name, under the log-on group.
@	groupname	accountname	@.GROUP.ACCT	All files in the group named, under the designated account.
@	groupname		@.GROUP	All files in the group named, under the log-on account.
@			@	All files in the log-on group. This is the default case.
@	@	accountname	@.@.ACCT	All files in all groups under the account named.
@	@		@.@	All files in all groups under the log-on account.
@	@	@	@.@.@	All files in the system.

@ means all

Examples Using Library Commands

A program is input, named, and saved in the user's library. It is then scratched as the current program:

```
>100 INPUT A,B,C,D,E
>120 LET S=(A+B+C+D+E)/5
>130 PRINT S
>NAME AVERAGE
>SAVE
>SCRATCH
```

A second program is entered, named, and saved. The first program is then appended to this program to make a third program. It too is named and saved:

```
>10 INPUT R
>20 P=3.14
>30 A=P*R**2
>40 PRINT A
>NAME AREA
>SAVE
>APPEND AVERAGE
>SAVE CALC
```

Any of these programs may now be brought back as the current program with GET. To illustrate, each is retrieved and then listed:

```
>GET AVERAGE
>LIST
AVERAGE
 100 INPUT A,B,C,D,E
 120 LET S=(A+B+C+D+E)/5
 130 PRINT S
>GET AREA
>LIST
AREA
  10 INPUT R
  20 P=3.14
  30 A=P*R**2
  40 PRINT A
>GET CALC
>LIST
CALC
  10 INPUT R
  20 P=3.14
  30 A=P*R**2
  40 PRINT A
 100 INPUT A,B,C,D,E
 120 LET S=(A+B+C+D+E)/5
 130 PRINT S
```

To determine whether a particular program is in the user's library, he can type CATALOG followed by the program name. If there are not too many files in the current log-on group, he can simply type CATALOG to get a list of all the files currently saved.

In this example, the user requests a catalog of the program CALC. He then types RUN CALC and the program will be retrieved from the library and run:

```
>CATALOG CALC


ACCOUNT=LANG        GROUP=BASIC

  NAME     RECORDS    NAME     RECORDS    NAME     RECORDS
CALC      SP     2
>


>RUN CALC
CALC
?60
 11304
?34,56,43,61,54,73
 49.6
```

If there is no further need for the saved programs, each may be purged as follows:

```
>PURGE CALC
>PURGE AREA
>PURGE AVERAGE
```

The program CALC remains the current program as a result of the RUN CALC command until it is scratched or is replaced by another program in the user's library, or until the user exits from BASIC.

Saved programs remain in the library after log-off and can only be removed with the PURGE command.

SECTION III
Arrays

An array (or matrix) is a set of variables which is known by one name. The individual elements of an array are specified by the addition of a subscript to the array name: for example, M(7) is the seventh element of array M.

Arrays have either one or two dimensions. A one-dimensional array consists of a single column of many rows. The elements are specified by a single subscript, indicating the row desired. Rows and columns are numbered starting with 1. A two-dimensional array consists of a specified number of rows and a specified number of columns organized into a table. For example, an array M of five rows and three columns can be represented as follows:

	Columns		
Rows	1	2	3
1	M(1,1)	M(1,2)	M(1,3)
2	M(2,1)	M(2,2)	M(2,3)
3	M(3,1)	M(3,2)	M(3,3)
4	M(4,1)	M(4,2)	M(4,3)
5	M(5,1)	M(5,2)	M(5,3)

Each element of the array is specified by a pair of subscripts separated by commas; the first indicates the row and the second the column.

Every array in a BASIC/3000 program is defined in one of three ways:

- Through a DIM statement that specifies the array name, and the number of rows and columns.
- Through a type declaration that specifies the same information as DIM and also declares the array to contain a particular data type.
- Through usage—numeric arrays that are used but are not explicitly defined in a DIM or type statement have 10 rows if one-dimensional or 10 rows and 10 columns if two-dimensional.

The physical size of an array is the total number of elements originally allocated to it; the logical size is the current number of rows times the current number of columns. The physical size of an array cannot be changed during execution, but the logical size (that is, the number of rows and columns) can be changed with a REDIM statement so long as the physical size is not exceeded.

BASIC/3000 permits arrays of all numeric data types as well as one-dimensional string arrays. Remarks in this section refer to numeric arrays, unless otherwise noted. String arrays are described in section V.

This section describes DIM and REDIM as used for numeric arrays. In addition it describes special statements used for computation and manipulation of one- and two-dimensional arrays. All of these statements begin with the word MAT.

DIM Statement

The DIM statement is used to reserve storage for arrays and to set upper bounds on the number of elements in arrays. DIM statements may also be used with strings (see Section V).

Form

DIM variable(integer),variable(integer), . . .

where the *variable* is the array name, and the *integer* specifies the number of rows in a one-dimensional array.

DIM variable(integer,integer),variable(integer,integer), . . .

where the *variable* names a two-dimensional array, and the first *integer* specifies the number of rows in the array, the second *integer* the number of columns.

Rows and columns are numbered starting with 1. The overall array size is the number of elements. In a one-dimensional array it is identical to the number of rows; in a two-dimensional array it is the product of the rows and columns.

More than one array can be named in a DIM statement; they are separated by commas.

Explanation

The elements of an array are specified by subscripted variables. The values of the elements are undefined when the program begins. The number of elements in the array is defined by a DIM statement, a type statement, or by usage. The DIM statement can appear anywhere in a program and is not executed. If control transfers to a DIM statement, execution falls through to the next sequential statement.

Examples

```
10 DIM A[15],B[15,5],B1[2,10]
20 REM    A HAS 15 ROWS, ONE COLUMN
30 REM      B AND B1 ARE TWO-DIMENSIONAL ARRAYS
40 REM      B HAS 15 ROWS, 5 COLUMNS;B1 HAS 2 ROWS,10 COLUMNS
50 DIM C[5],C1[5,1],C2[1,5]
60 REM     C AND C1 HAVE THE SAME DIMENSIONS: 5 ROWS, 1 COLUMN
70 REM     C2 HAS 1 ROW, 5 COLUMNS
```

Note that the DIM statement for C1 in line 50 would be the same if it were C1(5).

REDIM Statement

The REDIM statement is used to vary the number of rows and columns in arrays. REDIM is also used with strings (see Section V).

Form

REDIM variable(integer expression),variable(integer expression), . . .

REDIM variable(integer expression,integer expression),
variable(integer expression,integer expression), . . .

REDIM is like DIM except that the rows and columns can be specified with *integer expressions.* The value of the expression must be positive.

When more than one array is specified in a REDIM expression, they are separated by commas.

Explanation

The variables in a REDIM statement must have been previously dimensioned either explicitly with a DIM or type statement, or implicitly through use. When using REDIM to redimension an array, the number of rows and columns can be changed as desired provided these two conditions are met:

- The number of dimensions must not be changed.
- The total number of elements (rows times columns) must not be increased beyond the physical size (original dimensions) of the array.

Any data elements whose subscripts are included in both the old and new dimensions retain their old values in the newly dimensioned array. New elements have undefined values.

Arrays may be implicitly redimensioned in MAT READ, MAT INPUT, and the MAT Initialization and MAT Operation statements.

Examples

```
100 DIM A[20],B[5,5]
120 FOR X=1 TO 20
130   A[X]=0
140 NEXT X
150 B[1,4]=100
160 PRINT (FOR A=1 TO 20,A[A]),B[1,4]
170 REDIM A[10],B[2,6]
180 PRINT (FOR A=1 TO 10,A[A]),B[1,4]
```

Each element in A is set to zero, then one element in B is set to 100, and the results are printed. After redimensioning, the results are again printed. Note that B(4,1) is not affected by REDIM since it is still within the bounds of the redimensioned array.

```
 0          0          0          0          0
 0          0          0          0          0
 0          0          0          0          0
 0          0          0          0          0
 100
 0          0          0          0          0
 0          0          0          0          0
 100
```

In the example below, array C is dimensioned by use to have 10 rows and 10 columns, and array C5 to have 10 rows. An element in each array is assigned a value which, when the arrays are redimensioned, are out of the bounds of the array. Other elements within the new bounds are then given values and printed.

```
 10 C[4,1]=99
 20 C5[10]=0
 30 REM   BOTH C AND C5 ARE DIMENSIONED BY USE
 40 REM   C WITH 10 ROWS AND 10 COLUMNS
 50 REM   C5 WITH 10 ROWS
 60 PRINT C[4,1]
 70 PRINT C5[10]
 80 REDIM C[2,12],C5[5]
 90 REM   C(4,1) AND C5(10) ARE NO LONGER DEFINED
100 C[2,1]=70
110 C5[5]=100
120 PRINT LIN(1),C[2,1],C5[5]
>RUN
 99
 0

 70            100
```

Storing Data in Arrays

There are several methods of assigning values to arrays. Individual elements can be assigned using the assignment statement:

```
10 LET A[5]=26
20 B[1,9]=N*4.5
```

In addition, individual elements can appear in INPUT and READ statements:

```
10 INPUT A[1],A[2],A[3]
20 READ B[12]
```

If embedded FOR loops are used, entire arrays can be filled element by element:

```
10 INPUT (FOR N=1 TO 5,A[N])
20 READ (FOR N=1 TO 5,(FOR M=1 TO 5,B[N,M]))
```

To simplify the use of arrays, the MAT INPUT and MAT READ statements are provided to fill entire arrays.

MAT READ/INPUT STATEMENTS

The MAT READ statement assigns values from DATA statements to entire arrays, row by row. If dimensions are specified, the array is given new logical dimensions. The MAT INPUT statement is identical to MAT READ except that the values are taken from the input device (e.g., terminal) as in an INPUT statement.

Form

MAT READ array, array, . . .

MAT INPUT array, array, . . .

each *array* is either an array name (A,B7, etc.) or an array name followed by new dimensions (A(5), B(5,J)). The dimensions can be expressions. The rules for assigning new dimensions are given in the description of REDIM.

Explanation

If an array is dimensioned in MAT INPUT or MAT READ, the new logical size (i.e., the total number of elements) must not be more than were originally allocated to the array, nor may the number of dimensions be altered.

If the array is a string array, only the number of elements can be changed by MAT INPUT or MAT READ. The size of the elements in the string cannot be changed. (See Section V for a description of strings and string arrays.)

None of the special extensions available with a simple INPUT, such as saving excess input, are allowed with MAT INPUT.

Examples

```
  10 DIM A[9],C[10,4]
  20 MAT READ A
  25 RESTORE
  30 MAT READ C[8,4]
  40 PRINT A[1],A[5],A[9]
  50 PRINT C[1,1],C[5,2],C[8,4]
1000 DATA 1,2,3,4,5,6,7,8,9,10
1010 DATA 10,9,8,7,6,5,4,3,2,1
1030 DATA 30,31,32,33,34,35,36,37,38,39
1040 DATA 40,41,42,43,44,45,46,47,48,49
>RUN
 1                   5                   9
 1                   3                   41
```

Three elements from each array are printed. Array C is redimensioned by MAT READ in line 30. Note that the RESTORE and DATA statements have the same functions with MAT READ as they do with READ.

In the next example, the MAT INPUT statement expects input from the user. Both arrays A and C are printed in their entirety using FOR loops as print items.

```
  10 DIM A[9],C[2,3]
  20 MAT INPUT A,C
  30 PRINT (FOR N=1 TO 9,A[N])
  40 PRINT (FOR N=1 TO 2,(FOR M=1 TO 3,C[N,M]))
>RUN
?9,8,7,6,5,4,3,2,1
??22,33,44,55,66,77
 9                   8                   7                   6                   5
 4                   3                   2                   1
 22                  33                  44                  55                  66
 77
```

Printing Data from Arrays

The mechanisms for printing data from arrays are parallel to those used for filling arrays. Individual elements can be printed using PRINT:

```
100 PRINT A[1],A[2],A[3]
```

If embedded FOR loops are used, entire arrays can be printed element by element:

```
100 PRINT (FOR N=1 TO 15,A[N])
200 PRINT (FOR N=1 TO 15,(FOR M=1 TO 5,B[N,M]))
```

To simplify the use of arrays, the MAT PRINT statement is provided to print entire arrays. MAT PRINT is also available for printing string arrays (see Section V) and for printing arrays to files (see Section VIII).

MAT PRINT STATEMENT

The MAT PRINT statement allows the printing of one or more complete arrays in a single statement. The elements are printed row by row and can be spaced out in fields or packed together, as in the PRINT statement (Section II).

Form

The form of a MAT PRINT statement is:

MAT PRINT mat print item,mat print item, . . .

A *mat print item* is either an array name or special function (TAB,LIN,CTL, and SPA); items are separated by a comma or semicolon and the list is optionally terminated by a comma or semicolon. FOR loops are not allowed in MAT PRINT.

Explanation

Each row of each array is printed separately, with double spacing between rows. If a comma follows the array, each element starts in one of the five divisions of the line (see "PRINT Statement," Section II). If a semicolon follows the array, the elements are printed packed together, as if each element were followed by a semicolon. If nothing follows the last array, a comma is assumed. All formatting is done according to the specifications under PRINT statement.

An undefined array element causes the program to terminate. A one-dimensional array is printed as a single row.

Examples

```
  10 DIM A[10],B[5,5],C[2,2]
  20 MAT READ A,B[3,5],C
  30 MAT PRINT A
  40 PRINT
  50 MAT PRINT B,LIN(1),C,LIN(1)
  60 MAT PRINT A;LIN(2),B;
1000 DATA 2.5,46.7,75,0,50.1,0,0,0,19.8,0
1010 DATA 1,2,3,4,5,6,7,8,9,10
1020 DATA 11,12,13,14,15,16,17,18,19,20
>RUN
 2.5            46.7            75              0               50.1
 0              0               0               19.8            0


 1              2               3               4               5

 6              7               8               9               10

 11             12              13              14              15


 16             17

 18             19



 2.5         46.7        75    0     50.1     0     0     0
 19.8        0


 1     2     3     4     5

 6     7     8     9     10

 11    12    13    14    15
```

Note the effect of the semicolons following A and B in the MAT PRINT statement, line 60, on the printed output. MAT READ in line 20 redimensions array B; redimensioning of arrays is not permitted in a MAT PRINT statement.

Initializing Arrays

Three special functions (ZER, CON, IDN) provide the means to initialize numeric arrays with certain values, and optionally to redimension the arrays.

Form

The forms of MAT initialize statements are:

MAT numeric array=function

MAT numeric array=function(*dimension*)

The allowable *functions* are ZER, CON, and IDN.

The (*dimension*) part is optional and consists of one or two integer expressions separated by a comma. It changes the logical size of the array.

Explanation

ZER sets all elements of the array to zero.

CON sets all elements of the array to one.

IDN assigns an identity array to the array specified. The identity array is all zeroes, except the major diagonal which is all ones. The major diagonal starts in the upper left corner. If the array is not square, ones are extended along the diagonal as far as possible.

If an array is redimensioned by ZER, CON or IDN, the new size cannot have more elements than the original size, nor can the number of dimensions be altered.

Examples

```
  10 DIM A[5,5]
  20 MAT A=ZER
  30 MAT PRINT A
>RUN
 0              0              0              0              0

 0              0              0              0              0

 0              0              0              0              0

 0              0              0              0              0

 0              0              0              0              0
```

Function ZER sets each element in array A to zero.

```
  10 DIM A[4,4]
  20 MAT A=CON(3,4)
  30 MAT PRINT A
>RUN
 1               1               1               1

 1               1               1               1

 1               1               1               1
```

MAT A=CON(3,4) redimensions array A to have 3 rows and 4 columns, and sets each element in the newly dimensioned array to 1.

```
  10 DIM A[5,5]
  20 MAT A=IDN(4,4)
  30 MAT PRINT A
>RUN
 1               0               0               0

 0               1               0               0

 0               0               1               0

 0               0               0               1
```

IDN(4,4) changes the dimensions of A to 4 rows by 4 columns and sets the major diagonal to 1, the remaining elements to zero. If the array is not square, the extra elements are set to zero:

```
  10 DIM A[5,5]
  20 MAT A=IDN(5,3)
  30 MAT PRINT A
>RUN
 1               0               0

 0               1               0

 0               0               1

 0               0               0

 0               0               0
```

Array Operations

This group of six statements provides functions which operate on one or more entire arrays:

MAT Copy statement

MAT Add/Subtract statement

MAT Multiply statement

MAT Inverse statement

MAT Transpose statement

MAT Scalar Multiply statement

The arrays named in each statement all must be the same numeric type (see Section IV, Variable Types).

ARRAY COPYING

The MAT Copy statement copies one array into another. The form is

MAT numeric array=numeric array

The array on the right is copied into the array on the left. The destination array must have at least as many elements as the source and the same number of dimensions. It is redimensioned to have the same number of rows and columns as the source.

Examples

```
  10  DIM A1[2,3],B2[3,2]
  20  MAT READ B2
  30  MAT A1=B2
  40  MAT PRINT A1
1000  DATA 2.5,46.7,75,0,50.1,0,0,0,19.8,0
>RUN
 2.5              46.7

 75               0

 50.1             0
```

ARRAY ADDITION/SUBTRACTION

The MAT Add/Subtract statement performs array addition or subtraction (element by element) upon arrays of identical logical size and assigns the result to another array. The form is

MAT numeric array=numeric array+numeric array

MAT numeric array=numeric array – numeric array

The resulting array is assigned to the array on the left, which is redimensioned as in MAT copy. Any or all of these arrays may be the same array.

Examples

```
  10 DIM B[2,2],A1[2,2],A2[2,2]
  20 MAT READ A1,A2
  30 MAT B=A1+A2
  40 MAT PRINT A1,LIN(2),A2,LIN(2),B
1010 DATA 1,2,3,4,5,6,7,8,9,10
>RUN
 1                 2

 3                 4



 5                 6

 7                 8



 6                 8

 10                12
```

The values in arrays A1 and A2 are added to produce the values printed for array B. Using the same data, A1 is subtracted from A2 to produce the following results in B:

```
  10 DIM B[2,2],A1[2,2],A2[2,2]
  20 MAT READ A1,A2
  30 MAT B=A1-A2
  40 MAT PRINT B
1010 DATA 1,2,3,4,5,6,7,8,9,10
>RUN
-4                -4

-4                -4
```

ARRAY MULTIPLICATION

The MAT Multiply statement performs an array multiplication on an array of dimension *m* by *n* and an array of dimension *n* by *p*; that is, the number of columns in the first array must equal the number of rows in the second. The result, a new array of dimension *m* by *p*, is assigned to a third array. The form is

*MAT numeric array = numeric array * numeric array*

Each row of the array to the left of * is multiplied by each column of the array on the right to produce the new element. The resulting array is assigned to the array to the left of the assignment operator. This array is redimensioned to dimension *m* by *p* as in the MAT Copy statement. Any or all of these arrays may be the same array.

Examples

```
 10 DIM A1[2,3],A2[3,2],B[2,2]
 20 MAT READ A1,A2
 30 MAT B=A1*A2
 40 MAT PRINT A1;LIN(1),A2;LIN(1),B;
100 DATA 1,2,3,4,5,6
200 DATA 4,5,6,7,8,9
>RUN
1      2      3   }
                  }  array A1
4      5      6   }

4      5          }
                  }
6      7          }  array A2
                  }
8      9          }

40     46         }
                  }  array B = A1*A2
94     109        }
```

The method for performing a matrix multiplication is to multiply each element of the first row of array A1 by the corresponding element of the first column of A2 and to add the products. The result is the element B(1,1). Then each element in the first row of A1 is multiplied by the corresponding element in the second column of A2 and these are added to produce B(1,2). B(2,1) is the sum of the products resulting from the multiplication of row 2 of A1 and column 1 of A2; B(2,2) is the sum of the products of row 2 of A1 and column 2 of A2. To illustrate:

1X4 (4) + 2X6 (12) + 3X8 (24) = 40

4X4 (16) + 5X6 (30) + 6X8 (48) = 94

1X5 (5) + 2X7 (14) + 3X9 (27) = 46

4X5 (20) + 5X7 (35) + 6X9 (54) = 109

A second example multiplies the square array C by itself. In this case, the number of columns always equals the number of rows.

```
  10 DIM C[3,3]
  20 MAT INPUT C
  30 MAT PRINT C;LIN(1)
  40 MAT C=C*C
  50 MAT PRINT C;
>RUN
?2,4,6,8,1,3,5,7,9
 2      4      6

 8      1      3

 5      7      9



 66     54     78

 39     54     78

 111    90     132
```

To achieve the result MAT C=C*C;

C(1,1) = 2X2 (4) + 4X8 (32) + 6X5 (30) = 66

C(1,2) = 2X4 (8) + 4X1 (4) + 6X7 (42) = 54

C(1,3) = 2X6 (12) + 4X3 (12) + 6X9 (54) = 78

C(2,1) = 8X2 (16) + 1X8 (8) + 3X5 (15) = 39

C(2,2) = 8X4 (32) + 1X1 (1) + 3X7 (21) = 54

C(2,3) = 8X6 (48) + 1X3 (3) + 3X9 (27) = 78

C(3,1) = 5X2 (10) + 7X8 (56) + 9X5 (45) = 111

C(3,2) 5X4 (20) + 7X1 (7) + 9X7 (63) = 90

C(3,3) 5X6 (30) + 7X3 (21) + 9X9 (81) = 132

This next example multiplies a two-dimensional array with three rows and two columns by a one dimensional array with two rows. The result is a one-dimensional array with three rows.

```
 10 DIM A[3,2],B[2],C[3]
 20 MAT READ A
 30 MAT READ B
 40 MAT C=A*B
 50 DATA 1,2,3,4,5,6,1,2
 60 MAT PRINT A;LIN(1),B;LIN(1),C;
>RUN
 1      2

 3      4

 5      6


 1      2


 5      11      17
```

To achieve the result MAT C=A*B:

C(1)= 1×1 (1) + 2×2 (4) = 5

C(2)= 3×1 (3) + 4×2 (8) = 11

C(3)= 5×1 (5) + 6×2 (12) = 17

ARRAY INVERSION

The MAT Inverse statement assigns the inverse of a square array (i.e., number of rows equals number of columns) to another array. The inverse of an array is the array which, when multiplied by the original array, results in the identity array. The form is

MAT numeric array = INV (*numeric array*)

The arrays must not have been declared type integer (see Section IV). The array to the left of the assignment operator is redimensioned as in MAT Copy. The two arrays may be the same.

Example

```
  10 DIM A[10,3],B[5,5]
  20 MAT INPUT B
  30 MAT A=INV(B)
  40 MAT PRINT B,LIN(2),A
>RUN
?1,0,0,0,0,2,1,0,0,0,3,2,1,0,0,4,3,2,1,0,5,4,3,2,1
 1              0              0              0              0

 2              1              0              0              0

 3              2              1              0              0

 4              3              2              1              0

 5              4              3              2              1



 1              0              0              0              0

-2              1              0              0              0

 1             -2              1              0              0

 0              1             -2              1              0

 0              0              1             -2              1
```

25 values are input to the square array B, then using INV, array A is set to the inverse of B. Array A is redimensioned to the same dimensions as B.

ARRAY TRANSPOSITION

The MAT Transpose statement assigns the transposition of an *n* by *m* array to an *m* by *n* array. Transposition switches rows and columns. The form is

MAT numeric array = TRN (*numeric array*)

The array to the left is redimensioned as in the MAT Copy statement. The two arrays may be the same.

Example

```
 10  DIM A[5,3],B[3,5]
 20  MAT INPUT B
 30  MAT A=TRN(B)
 40  MAT PRINT B,LIN(2),A
>RUN
?1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
 1              2              3              4              5

 6              7              8              9              10

 11             12             13             14             15



 1              6              11

 2              7              12

 3              8              13

 4              9              14

 5              10             15
```

Array A is the result of transposing array B with the TRN function. The columns in B are the rows in A; the rows in B are the columns in A.

ARRAY SCALAR MULTIPLICATION

The MAT Scalar Multiply statement multiplies all of the elements of an array by a specified value and assigns the result to another array. The form is

MAT *numeric array* = (*numeric expression*) * *numeric array*

The array to the left is redimensioned as in the MAT Copy statement. The two arrays may be the same.

Example

```
  10 N=5
  20 MAT INPUT B
  30 PRINT
  40 MAT A1=(N*2)*B
  50 MAT PRINT A1;
  60 DIM A1[3,4]
  70 DIM B[2,6]
>RUN
?1,2,3,4,5,6,7,8,9,10,11,12

 10     20     30     40     50     60

 70     80     90     100    110    120
```

Scalar multiplication simply multiplies each element of the array by the specified numeric expression, in this case N*2 or 10 since N=5. Each element of the resulting array A1 is 10 times the corresponding element in B. The dimensions of A1 are copied from B. The two arrays must be the same numeric type; the numeric expression may be a different type.

The numeric expression, if a different type, is converted to the type of the arrays before multiplication is performed. The conversion may affect the result if, for instance, the scalar expression is type real and the arrays are type integer. Consider the following:

MAT A = (2.5)*B, where A and B are integers, is equivalent to MAT A = (3)*B.

Array Functions

Two functions which can be used in expressions return information about arrays: ROW and COL.

ROW Function

The ROW function has the form

ROW (array)

and returns the number of rows in the array (a one-dimensional array consists of one column, many rows).

COL Function

The COL function has the form

COL (array)

and returns the number of columns in the array (returns 1 if the array is one-dimensional).

Examples

```
  10 READ M,N
  20 MAT READ A[M,N]
  30 PRINT (FOR I=1 TO ROW(A),(FOR J=1 TO COL(A),A[I,J]))
  40 DATA 3,5,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
>RUN
 1             2             3             4             5
 6             7             8             9             10
 11            12            13            14            15
```

The dimensions of array A are read into the variables M and N. The functions ROW(A) and COL(A) are used in the FOR loop to determine the print bounds for printing the array.

SECTION IV
Variable Types

In addition to the floating-point real numbers used so far in this manual, BASIC/3000 allows three additional representations of data: integer, long real, and complex. Including real, these four number types apply to variables, arrays, constants, expressions, assignments, functions, input and output.

Type Statements

The type statements allocate space for variables and arrays and assign them a specific data type. Any particular variable or array can appear only in one type statement or DIM statement.

Form

The form of the type statement is

type typespec list

where *type* is either INTEGER, REAL, LONG, or COMPLEX. The *typespec list* includes variables and arrays to be assigned the data type of *type*. Arrays are defined in the same form as in the DIM statement (Section III).

Explanation

A simple variable or array which does not appear in a type statement is automatically type REAL. The explicit typing of variables in a REAL statement is, therefore, redundant, except within a function body, where all local variables must be declared in order to distinguish them from variables of the same name outside the function. Real numbers are represented as 32-bit quantities consisting of a sign, exponent, and fraction. The range of real numbers is $\pm (10^{-77}, 10^{77})$ with approximately 6 to 7 digits of precision.

Variables which appear in an INTEGER statement hold integers. The range of integers is -32767 to 32767.

Variables which appear in a LONG statement hold long numbers. Long representation is a 64-bit quantity with sign, exponent, and fraction. The range is identical to real, but long has a precision of 16 to 17 digits.

Variables which appear in a COMPLEX statement hold numbers in complex form. Complex representation is a 64-bit quantity consisting of two real numbers, one for the real part of the complex number and one for the imaginary part.

NUMERIC CONSTANT FORMS

When constants are used in an expression, DATA statement, or during execution of an INPUT or ENTER statement, they are represented in one of five forms: integer, fixed-point, floating-point, complex, or long. Fixed and floating-point numbers are *type* REAL.

Integer Form

An integer is a series of digits without a decimal point. A number in integer form is represented externally (e.g., on the list device) as *type* INTEGER, but internally as *type* REAL. Examples of integer form:

```
  10 INTEGER A,B,C,D
  20 A=10,B=150,C=5903,D=5
  30 PRINT A,B,C,D
>RUN
 10                150               5903              5
```

When arithmetic operations are performed on expressions containing an integer constant, the results are real numbers. However, when both operands are *type* INTEGER, the result is truncated to the nearest integer. For instance,

```
  10 INTEGER I,J
  20 LET I=3,J=5
  30 PRINT 3/5,3/J,I/5,I/J
>RUN
 .6                .6                .6                0
```

Fixed-Point Form

A fixed-point number is a series of digits with a decimal point. A number in fixed-point form is represented internally as *type* REAL. For example:

```
  10 REAL A,B,C,D
  20 A=73,B=5.5,C=.000567,D=153.97
  30 PRINT A,B,C,D
>RUN
 73                5.5               .000567           153.97
```

Floating-Point Form

A floating-point number is a fixed or integer form number followed by the letter E and an optionally signed exponent. The exponent represents the power of 10 by which the number is multiplied. For example 3E-11 equals 3×10^{11}. Numbers in floating-point form are represented internally as *type* REAL. Examples of floating-point numbers:

```
  10 REAL A,B,C,D
  20 A=3E-11,B=.4723E-4,C=1.1E4,D=1.1E10
  30 PRINT A,B,C,D
>RUN
 3.00000E-11       4.72300E-05       11000             1.10000E+10
```

A fixed or floating point real number that has an integer value between -999999 and 999999 is printed as an integer.

Complex Form

A complex number consists of two numbers in integer, fixed-point, or floating-point form, separated by a comma and enclosed in parentheses. The first number is the real part, the second represents the imaginary part. Complex numbers are represented internally as *type* COMPLEX. Examples of complex numbers:

```
 10 COMPLEX A,B,C,D
 20 A=(3,5),B=(3.2E-9,0),C=(0,-47),D=(0,0)
 30 PRINT A,B,C,D
>RUN
( 3.00000E+00, 5.00000E+00)     ( 3.20000E-09, 0.00000E+00)
( 0.00000E+00,-4.70000E+01)     ( 0.00000E+00, 0.00000E+00)
```

Long Form

Numbers in long form are identical to numbers in real form, except that the letter E is replaced by the letter L. Long numbers have almost double the precision of real numbers. Long numbers are represented internally as *type* LONG. Examples of long numbers:

```
  10 LONG A,B,C,D
  20 A=3L-11,B=4.751259L-6,C=-1.1L5,D=1.1L-15
  30 PRINT A,B,C,D
>RUN
 3.000000000000000L-11           4.751259000000000L-06
-1.100000000000000L+05           1.100000000000000L-15
```

Examples of Type Statements

This example assigns values to and prints two integer variables and an integer array:

```
  10 INTEGER A,B1,N[5,5]
  20 LET A=5,B1=10
  30 MAT N=ZER
  40 PRINT A,B1
  50 MAT PRINT N;
>RUN
 5              10
 0     0     0     0     0

 0     0     0     0     0

 0     0     0     0     0

 0     0     0     0     0

 0     0     0     0     0
```

Note that the *type* statement is used instead of a DIM statement to define the dimensions of array N.

This example assigns values to and prints two real variables; one is printed as floating-point and the other as fixed-point:

```
 10 REAL I,J
 20 LET I=2795348.6,J=2.79E-3
 30 PRINT I,J
>RUN
 2.79535E+06    .00279
```

I is printed as a floating-point number because its magnitude is greater than 999999.5; J is printed as fixed-point because its magnitude is less than 999999.5 (see Numeric Output Formats in the PRINT statement description, Section II). Note that the printed value of I is rounded.

The following example inputs values to the *type* LONG variable P, then doubles each value and prints it:

```
  10 LONG P
  20 INPUT P
  30 LET P=P+P
  40 PRINT P
>RUN
?2.7L+10
 5.40000000000000000L+10

>RUN
?2.5L+12
 5.00000000000000000L+12

>RUN
?2.0L+11
 4.00000000000000000L+11
```

The example below reads data into two complex variables and one complex array, and then prints the variable and array values:

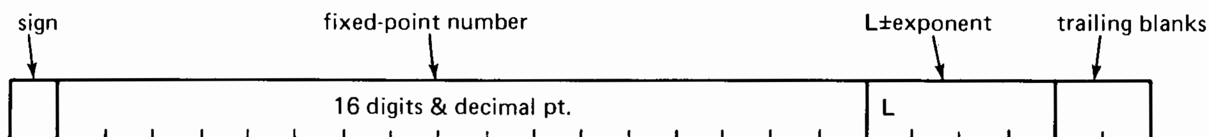
```
  10 COMPLEX C9,Q8,M[15]
  20 READ C9,Q8
  30 MAT READ M[5]
  40 PRINT C9,LIN(1),Q8,LIN(1)
  50 MAT PRINT M;
  90 DATA (4.5E-6,1.2E-9),4.23E6
 100 DATA (3,9),(4.5,-30),(4.5E-6,1.2E-9),(25.3,30.2),(0,0)
>RUN
( 4.50000E-06, 1.20000E-09)
( 4.23000E+06, 0.00000E+00)

( 3.00000E+00, 9.00000E+00)    ( 4.50000E+00,-3.00000E+01)
( 4.50000E-06, 1.20000E-09)    ( 2.53000E+01, 3.02000E+01)
( 0.00000E+00, 0.00000E+00)
```

PRINTING LONG AND COMPLEX DATA

Numbers of all data types can be output with the PRINT statement. All numeric quantities, regardless of type, are printed left-justified in a field whose width is always a multiple of 3. At least one blank is always printed on the right side of the field, unless it is the last item on the line.

The output form for values of type INTEGER and REAL is described under Numeric Output Formats in the PRINT statement description, Section II.

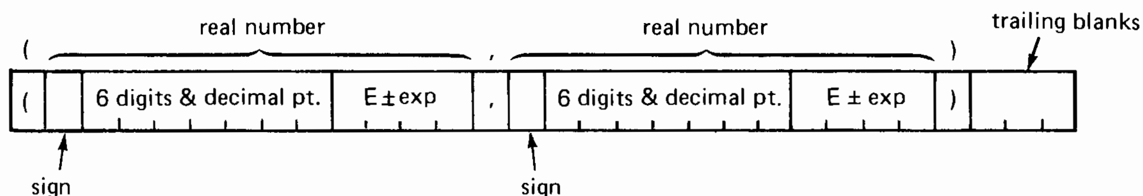
The output form for long quantities is an 16 digit fixed-point number followed by an exponent and two trailing blanks. The total required is 24 print positions.



For example:

```
  10  LONG A,B,C,D,E
  20  A=7.3215L9,B=4.32L-8
  30  C=4.3214978L-8,D=2.173L2
  40  E=2.173L6
  50  PRINT A;B;C;LIN(1),D;E
>RUN
 7.321500000000000L+09    4.320000000000000L-08    4.321497800000000L-08
 2.173000000000000L+02    2.173000000000000L+06
```

The output form for complex numbers is two real numbers separated by a comma and enclosed in parentheses (i.e., this is the same form as a complex constant). Each part of the number is printed as a separate 6-digit fixed-point number, followed by an exponent. The total required is 30 print positions including 3 trailing blanks:



For example:

```
   10 COMPLEX A,B
   20 LET A=(1.2E8,1.39E-6)
   30 LET B=(12.5,1.56E6)
   40 PRINT A;B
  >RUN
  ( 1.20000E+08, 1.39000E-06)   ( 1.25000E+01, 1.56000E+06)
```

NUMERIC EXPRESSIONS

Variables of all data types and numbers of all data forms can be used in numeric expressions. BASIC/3000 provides the arithmetic operations for all four data types as well as automatic conversion when two operands are not of the same type. The following table summarizes the results of combining arithmetic elements with any operator (except AND, OR, NOT, and relationals):

First Element Data Type \ Second Element Data Type	INTEGER	REAL	LONG	COMPLEX
INTEGER	INTEGER	REAL	LONG	COMPLEX
REAL	REAL	REAL	LONG	COMPLEX
LONG	LONG	LONG	LONG	COMPLEX
COMPLEX	COMPLEX	COMPLEX	COMPLEX	COMPLEX

When the operators AND, OR, NOT, =, <, >, <=, >=, and <> are used, the result is always type REAL (0 for false, 1 for true). When relations are performed on complex numbers, the real parts are compared first; the imaginary parts are compared only if the real parts are equal.

Examples

An integer combined with a real type in an expression results in a real number; two integers result in an integer:

```
  10 INTEGER I,I1
  20 REAL R
  30 LET I=25,I1=50,R=2.75
  40 PRINT I+I1
  50 PRINT I+R
>RUN
 75
 27.75
```

A real type combined with a long results in a long type number; a long type combined with a complex results in a complex type number:

```
  10 REAL R
  20 LONG L
  30 LET L=-5.25L2,R=2.75
  40 PRINT L+R
  50 COMPLEX C
  60 C=(2.75,-1.25)
  70 PRINT L+C
>RUN
-5.222500000000000L+02
(-5.22250E+02,-1.25000E+00)
```

CONDITIONAL STATEMENT

The numeric expression used to make a branching decision in a conditional statement (Section II) can contain, or result in, any numeric data type. The expression is considered false if equal to 0, true otherwise.

NUMERIC ASSIGNMENT

When the result of a numeric expression is assigned to a variable, it is converted to the type of that variable. In a LET statement, the same result can be assigned to several variables in turn, from right to left (A=B=C=5+D7). These variables need not be of the same type. If they are not, a conversion is performed at each step in the assignment.

The method of conversion used in assigning values to variables of differing data types is summarized in this table:

Variable Type	Value Type	Conversion Method
INTEGER	REAL	Round.
INTEGER	LONG	Round.
INTEGER	COMPLEX	Round real part; drop imaginary part.
REAL	INTEGER	Float.
REAL	LONG	Truncate to real precision.
REAL	COMPLEX	Drop imaginary part.
LONG	INTEGER	Float to long precision.
LONG	REAL	Extend mantissa with zeroes.
LONG	COMPLEX	Extend mantissa of real part with zeroes; drop imaginary part.
COMPLEX	INTEGER	Float for real part; imaginary part equals zero.
COMPLEX	REAL	Imaginary part equals zero.
COMPLEX	LONG	Truncate to real precision for real part; imaginary part equals zero.

Note that this table applies wherever values are assigned to variables (INPUT, READ, etc.).

An example of multiple assignment with type conversion is:

```
 10  INTEGER I
 20  REAL R
 30  COMPLEX C
 40  LONG L
 50  LET R=C=I=L=1.5L0
 60  PRINT R,C,I,L
>RUN
 2              ( 2.00000E+00,  0.00000E+00)     2
 1.50000000000000L+00
```

Note that the long number is rounded up to 2 when it is converted to the integer variable I. R and C also equal 2 since they are assigned after I. If line 50 is changed so that C is assigned before I, the rounding does not affect C:

```
 10  INTEGER I
 20  REAL R
 30  COMPLEX C
 40  LONG L
 50  LET R=I=C=L=1.5L0
 60  PRINT R,C,I,L
>RUN
 2              ( 1.50000E+00,  0.00000E+00)     2
 1.50000000000000L+00
```

If the constant 1.4 is assigned instead of 1.5, the number is rounded to 1 when it is converted to an integer and assigned to I. As a result of this integer conversion, R is also set equal to 1:

```
 10  INTEGER I
 20  REAL R
 30  COMPLEX C
 40  LONG L
 50  LET R=I=C=L=1.4L0
 60  PRINT R,C,I,L
>RUN
 1              ( 1.40000E+00,  0.00000E+00)     1
 1.40000000000000L+00
```

ENTERING NUMERIC DATA

Constants of all data forms can be entered using READ, INPUT, and ENTER statements. Once entered they are converted to the type of the receiving variable according to the table under "Numeric Assignment."

OTHER USES OF DATA TYPES

Numbers of all data types can be output with controlled format with the PRINT USING statement (see Section IX). Numbers of all data types can also be written onto and read from mass storage data files. This process is described fully in Section VIII.

NUMERIC ARRAYS

Arrays can be of all data types. Each element of the array is a variable of the specified type. The type statement effectively provides the dimensions of an array. All of the MAT statements dealing with arrays (see Section III) apply equally to integer, real, long, and complex arrays, except that integer arrays cannot be inverted with the MAT Inverse statement. Arrays of different types cannot be mixed in a MAT statement.

Examples

```
 10 INTEGER I[3,5]
 20 LONG L[2,2]
 30 REAL R[2,2]
 40 COMPLEX C[2,2]
 50 MAT I=ZER
 60 MAT R=CON
 70 MAT C=IDN
 80 MAT L=IDN
 90 MAT L=(25**2)*L
100 MAT PRINT I;LIN(1),R;LIN(1),C;LIN(1),L;
>RUN
 0     0     0     0     0

 0     0     0     0     0

 0     0     0     0     0


 1     1

 1     1


( 1.00000E+00, 0.00000E+00)   ( 0.00000E+00, 0.00000E+00)

( 0.00000E+00, 0.00000E+00)   ( 1.00000E+00, 0.00000E+00)


 6.25000000000000L+02   0.00000000000000L+00

 0.00000000000000L+00   6.25000000000000L+02
```

FUNCTION CLASS

Numeric built-in functions are divided into four classes according to the nature of their result.

The following table defines the four classes of function. The type of result they return is shown for the different argument types:

Type of Argument

Class	INTEGER/REAL	LONG	COMPLEX
1	REAL	LONG	REAL
2	REAL	LONG	COMPLEX
3	COMPLEX	COMPLEX	COMPLEX
4	REAL	REAL	REAL

The numeric functions are listed below according to their class. A complete list of these functions with their meaning is contained in Appendix E.

Class 1 Functions

ABS(x)	Absolute value of x.
ATN(x)	Arctangent x.
INT(x)	Largest integer less than or equal to x.
CEI(x)	Smallest integer greater than or equal to x.

Class 2 Functions

EXP(x)	e^{x}
LOG(x)	$\log_{e}x$
SQR(x)	Square root of x.
SIN(x)	Sine x.
COS(x)	Cosine x.

TAN(x)	Tangent x.
SNH(x)	Hyperbolic sine x.
CSH(x)	Hyperbolic cosine x.
TNH(x)	Hyperbolic tangent x.
PIX(x)	$\pi * x$

Class 3 Functions

CNJ(x)	Complex conjugate of x.
CPX(x,y)	Complex number $x+yi$

Class 4 Functions

RND(x)	Random number.
REA(x)	Real part of x.
IMG(x)	Imaginary part of x.

SECTION V
Strings

BASIC/3000 allows the programmer to manipulate character strings through the use of string literals, variables, arrays, functions, operators, assignment statements, and input/output statements. Many of the uses of strings are enhancements to statements that have already been described, such as READ and PRINT.

LITERAL STRINGS

A literal string is a sequence of up to 255 characters. Each character is represented internally by a number between zero and 255 as defined in the standard ASCII character set (see Appendix A). Some of these characters have graphic representations (they can be printed—A, B, d, %), while others do not (they are nonprinting—return, linefeed). Both types of characters can be included in a literal string, but each is handled differently.

Form

A literal string consists of a series of graphic characters surrounded by quote marks:

> "*character string*"

The quote mark cannot be included as a character in the *character string.*

The quote mark and nonprinting characters can, however, be included in a literal string by using the integer numeric equivalent of the character preceded by an apostrophe:

> '*integer*

The *integer* may be in the range 0-255, but it is good practice to restrict this form to nonprinting characters and the quote mark (34). Nonprinting characters can be combined with quoted strings in a literal string.

Explanation

Literal strings can include both upper case and lower case letters. When a literal string is printed, each character value is printed literally on the output device. However, when a program is listed, literal strings are listed with all graphic characters except the quote mark in quotes and non-graphic characters represented in the apostrophe form.

Examples

`""`	*A null string (a string of zero length)*
`"BASIC"`	
`"B     "`	
`'13'10`	*Carriage return, line feed*
`'13'10"TRIPLE STRING"'7`	*The literal ends with a bell*
`"A"'124"B"`	*The literal is A vertical line B*
`'34`	*The quote mark*

The apostrophe literal form can be juxtaposed with another apostrophe literal; quoted strings cannot be juxtaposed with one another.

DIM Statement with Strings

Literal strings can be contained in string variables, simple or subscripted. Simple string variables greater than one character in length and every array string variable must be dimensioned in a DIM statement. The purpose of the DIM statement is to reserve storage for strings and arrays and to establish their names and maximum size.

Form

The DIM statement consists of the word DIM followed by a list of variable and array definitions separated by commas.

DIM variable(string size),variable(string size), . . .

where *variable* is the name of a simple string variable specified as a letter followed by a $ or a letter and a digit followed by a $. The *string size* is an integer constant that specifies the maximum number of characters the string can contain.

DIM variable(array size,string size),variable(string size), . . .

The *array size* specifies the total number of elements in the array; the *string size* specifies the maximum number of characters in each element. Only one-dimensional string arrays are allowed. Both array size and string size are integers.

If more than one *variable* is included in a single DIM statement, they must be separated by commas. Simple string and string array variables may be dimensioned in the same DIM statement.

Explanation

If a string variable does not appear in a DIM statement then it is implicitly defined as a one-character simple variable. String arrays must be declared in DIM; there is no implicit size for string arrays as there is for numeric arrays. String variables and elements of string arrays are initialized to the null string.

The DIM statement can also be used to declare numeric arrays in the same or a different statement (see Section III).

Example

```
 10 DIM A$[19],B$[5,12],C5$[10,12]
 20 A$="TITLE OF SECTION IS"
 30 MAT READ B$
 40 MAT READ C5$
 50 DATA "ARRAYS","STRINGS","MESSAGES","FILES","INPUT/OUTPUT"
 60 DATA "A1","A2","A3","A4","A5","B1","B2","B3","B4","B5"
 70 PRINT A$,C5$[5],B$[5]
>RUN
TITLE OF SECTION IS            A5                 INPUT/OUTPUT
```

The entire simple string variable A$ is printed, followed by the 5th element of the string array C5$ and the 5th element of the string array B$.

REDIM Statement with Strings

The purpose of the REDIM statement is to dynamically vary the number of elements in a string array, but not the string size itself.

Form

The form of the REDIM statement is the word REDIM followed by a list of previously dimensioned string array variables, each one followed by a new size specification in parentheses.

REDIM variable(new array size)

The *new array size* must be an integer or an integer expression that results in a value between 1 and the previously defined maximum size of the array. Only the array size may be redimensioned; the string size may not be changed. Simple string variables may not be redimensioned. The *variable* must be a string array that has been dimensioned, either implicitly or in a DIM statement.

Explanation

REDIM changes the size of a one-dimensional string array, but cannot increase the original size. The array must have been previously dimensioned with DIM. Unlike DIM, REDIM is an executable statement and its position in a program has meaning. The REDIM statement can also be used to change the dimensions of numeric arrays in the same or a different statement (see Section III).

Example

The number of elements in the string arrays C5$ and B$ is reduced in a REDIM statement in line 50. Then the values read into the 3rd element of C5$ and the 2nd element of B$ are printed following the value of the simple string A$. The maximum length of the individual elements in C5$ and B$ is not affected by REDIM; these lengths remain as specified in the original DIM statement.

```
 10 DIM A$[20],B$[5,12],C5$[10,2]
 20 A$="TITLE OF SECTION IS"
 30 DATA "ARRAYS","STRINGS","MESSAGES","FILES","INPUT/OUTPUT"
 40 DATA "A1","A2","A3","A4","A5","B1","B2","B3","B4","B5"
 50 REDIM B$[3],C5$[5]
 60 MAT READ B$
 70 RESTORE 40
 80 MAT READ C5$
 90 PRINT A$,C5$[3],B$[2]
>RUN
TITLE OF SECTION IS           A3              STRINGS
```

String Variable

A string variable (simple or subscripted) is used to hold a string literal. The declared size of a string variable is called its physical length. The maximum length of any string variable is 255 characters. A string variable not mentioned in a DIM statement is a simple variable one character in length.

During execution, each string variable contains strings whose length cannot exceed the variable's physical size. This dynamic length is called the logical length of the variable and is initialized to zero (i.e., the null string) at the beginning of program execution.

Form

A simple string variable is referenced by its name and an optional *substring designator* in parentheses.

string name

string name(*first character*)

string name(*first character*,*last character*)

string name(*first character*;*number of characters*)

The *string name* is a letter followed by a $ or a letter and a digit followed by a $. The string name may be followed by a *substring designator* in parentheses.

The *substring designator* consists of one or two numeric expressions, separated by a comma or semi-colon. The first expression always specifies the *first character* position of the substring. The ending character is determined by the second expression.

If the two expressions are separated by a comma, the second expression specifies the *last character position*; if they are separated by a semicolon it specifies the *number of characters.* If there is only one expression, the ending character position is the last character of the string.

A string array variable is referenced by the *string name* followed, in parentheses, by a *subscript* and an optional *substring designator* separated by a comma.

string name(*subscript*)

string name(*subscript*,*first character*)

string name(*subscript*,*first character*,*last character*)

string name(*subscript*,*first character*;*number of characters*)

The *subscript* is an integer expression that specifies the element of the array to be selected. Since a string array may have only one dimension, there may be only one subscript value.

The *substring designator* and the *string name* are specified in the same way for string array variables as for simple string variables.

Unlike numeric array variables, a string array variable must not have the same name as a simple string variable.

```
  10 DEF FNR$(A$)
  20   IF LEN(A$)<=1 THEN RETURN A$
  30   RETURN FNR$(A$[2])+A$[1,1]
  40 FNEND
  50 DIM X$[5],B[2,5]
  60 X$="12345"
  70 IF FNR$(X$)="54321" THEN PRINT "YES"
  80 ELSE PRINT X$
  90 MAT READ B
 100 DATA 10,20,30,40,50,60,70,80,90,100
 110 END
>BREAK 20,70-90
>RUN
*BREAK  70
>SHOW X$
X$="12345"
>GO
*BREAK  20
>SHOW FNR$(A$)
FNR$:A$="12345"
>UNBREAK 20
>GO
YES
*BREAK  90
>SHOW A(*)
A DOES NOT EXIST
>SHOW B(*,*)
B[*]
UNDEFINED    UNDEFINED    UNDEFINED    UNDEFINED    UNDEFINED

UNDEFINED    UNDEFINED    UNDEFINED    UNDEFINED    UNDEFINED

>BREAK 100
>GO
*BREAK  100
>SHOW B(*,*)
B[*]
 10     20     30     40     50

 60     70     80     90     100

>GO
```

SET Command

The SET command allows the user to set any variable to a constant value; this command is legal only during a break period.

Form

The form of the SET command is

SET item = constant

The items to be set can include variables and array elements and local variables, specified as in the SHOW command, except that the form using asterisks may not be used.

Examples

```
 10 DIM X[5]
 20 MAT READ X
 30 DATA 273.1,765.3,795.1,654.9,195.7
 40 PRINT FND(X[*],5)
 50 END
210 DEF FND(A[*],INTEGER N)
220   REAL I,J
230   J=1
240   FOR I=2 TO N
250     IF A[I]>A[J] THEN J=I
260   NEXT I
270   RETURN J
280 FNEND
>BREAK 30
>RUN
*BREAK  30
>SHOW X(1)
X[1]=273.1
>SET X(1)=950.2
>SHOW X(1)
X[1]=950.2
>GO
 1
```

The result of the program is changed by setting the first element in the array X to a higher value than the other elements.

When the break points are removed, the program runs with the data read from the DATA statement in line 30:

```
>UNBREAK
>RUN
 3
```

FILES Command

The FILES command is legal only during a break period. It prints a list of all the files that are currently open in the executing program. The list is by name and internal file number (see Section VIII, Files).

Form

FILES

Explanation

When FILES is typed during a break, a list of the file numbers specified by the FILES statement in the executing program is printed. The numbers are in ascending order and each is followed by a file name if the file is open, by an asterisk if the file number is reserved but not yet open, or by *#n* where *n* is the file number of a file opened in another program that called the current program with INVOKE. The file name of an open file is qualified by the group name and account name.

Examples

In the first example, FILES specified in the break at line 30 shows four open files. The break at line 40, after the ASSIGN statement closed file number 5, shows only three files currently open:

```
  10 REM    PROGRAM ONE
  20 FILES A,B,*,C,D
  30 ASSIGN *,5
  40 END
>BREAK 30
>RUN
*BREAK  30
>FILES
1  A.BASIC.LANG
2  B.BASIC.LANG
3  *
4  C.BASIC.LANG
5  D.BASIC.LANG
>BREAK 40
>GO
*BREAK  40
>FILES
1  A.BASIC.LANG
2  B.BASIC.LANG
3  *
4  C.BASIC.LANG
5  *
```

In this example, program FIRST calls program SECOND with INVOKE. BREAK PROG is used to specify a breakpoint when control goes to SECOND and again when control reverts to FIRST. In SECOND, three local files are open, one of which is internal file #2 or the file B. It also shows the internal files A and B that were opened in FIRST and remain open following the INVOKE. The FILES command at the break upon return to FIRST shows that only the two files local to FIRST are open and that file #3 has been reserved:

```
SECOND
  10 REM     PROGRAM SECOND
  20 FILES C,#2,D
  30 END

FIRST
  10 REM     PROGRAM FIRST
  20 FILES A,B,*
  30 INVOKE "SECOND"
>BREAK PROG
>RUN
FIRST
*BREAK, INVOKE: SECOND
>FILES
1  A.BASIC.LANG
2  B.BASIC.LANG
3  *
4  C.BASIC.LANG
5  #2
6  D.BASIC.LANG
LOCAL FILES START AT 4
>GO
*BREAK, REVERT: FIRST
>FILES
1  A.BASIC.LANG
2  B.BASIC.LANG
3  *
>GO

>
```

CALLS Command

The CALLS command is legal only during a break period. It prints a list of all functions that have not been completed, and of all programs that have been called with INVOKE but have not been completed by END. This list is in reverse chronological order starting with the most recent.

Forms

CALLS

Examples

At the breakpoint for statement 40, the CALLS command shows that function FNN called FNM. Note that functions are listed in reverse chronological order:

```
PROG1
  10 DEF FNM(A,B)=SGN(A)*FNN(ABS(A),ABS(B))
  20 DEF FNN(A,B)
  30   X=A-INT(A/B)*B
  40   RETURN X
  50 FNEND
  60 PRINT FNM(-4,3)
>BREAK 40
>RUN
PROG1
*BREAK PROG1 40
>CALLS
FNN
FNM
```

In the following example, ALPHA2 uses INVOKE to call BETA2; BETA2 uses CHAIN to call GAMMA2. Because GAMMA2 returns to ALPHA2, not BETA2, a CALLS command entered during the break in GAMMA2 shows that ALPHA2 invoked GAMMA2:

```
GAMMA2
  10 REM    PROGRAM GAMMA2
  20 PRINT "IN GAMMA2 -- RETURN TO ALPHA2"

BETA2
  10 REM     PROGRAM BETA2
  20 CHAIN "GAMMA2"



ALPHA2
  10 REM     PROGRAM ALPHA2
  20 INVOKE "BETA2"
  25 PRINT "BACK IN ALPHA2 -- TERMINATE"
>BREAK PROG
>RUN
ALPHA2
*BREAK, INVOKE: BETA2
>GO
*BREAK, CHAIN: GAMMA2
>CALLS
INVOKED BY ALPHA2
```

Each of the following three programs contains at least one function definition and function call. Function FNA in program ALEF1 calls FNB wherein ALEF1 calls BET1 with an INVOKE statement. At the breakpoint in line 40 of GIMEL, function FNE calls FNF. The CALLS command entered during the breakpoint in GIMEL shows a complete history of all nested function calls and INVOKE statements in reverse chronological order:

```
GIMEL
  10 REM      PROGRAM GIMEL
  20 DEF FNE(X)=FNF(X)
  30 DEF FNF(X)
  40   PRINT "IN GIMEL"
  50   RETURN 0
  60 FNEND
  70 X=FNE(4)

BET1
  10 REM      PROGRAM BET1
  20 DEF FNC(X)
  30   INVOKE "GIMEL"
  40   RETURN 0
  60 FNEND
  70 X=FNC(3)



ALEF1
  10 REM      PROGRAM ALEF1
  20 DEF FNA(X)=FNB(X)
  30 DEF FNB(X)
  40   INVOKE "BET1"
  50   RETURN 0
  60 FNEND
  70 X=FNA(2)
>BREAK PROG
>RUN
ALEF1
*BREAK, INVOKE: BET1
>GO
*BREAK, INVOKE: GIMEL
>BREAK 40
>GO
*BREAK GIMEL 40
>CALLS
FNF
FNE
INVOKED BY BET1
FNC
INVOKED BY ALEF1
FNB
FNA
```

SECTION VIII
Files

For problems that require permanent data storage external to a particular program, BASIC/3000 provides a data file capability. This capability allows flexible, direct manipulation of large volumes of data stored on files.

There are three types of files used in BASIC/3000: formatted files, binary files, and ASCII files. Formatted files are created and accessed through the BASIC/3000 Interpreter. Binary and ASCII files are created in MPE/3000 but can be accessed with BASIC/3000.

A catalog of ASCII and binary files, as well as of formatted files in the user's group library, can be requested with the BASIC/3000 CATALOG command (see Commands, Section II).

BASIC FORMATTED FILES

A formatted BASIC/3000 file is created under control of the BASIC/3000 Interpreter. It contains format words to indicate the type of the data items in the file. These format words are placed in each record automatically by the Interpreter (see Appendix H for formatted file structure). Formatted files allow run-time checking of the type of each data item.

BASIC formatted files are created with the CREATE command or statement. They may be accessed by any file statements or commands including UPDATE and ADVANCE, but not File LINPUT which is reserved for ASCII files. The user's ability to access a BASIC file depends on the MPE file security restrictions.

ASCII FILES

ASCII files are created through the MPE/3000 Operating system and treated by BASIC/3000 as terminal-like devices. They can be actual terminals. Output to them is formatted according to the rules for the PRINT statement (see Section II). Input from ASCII files is analyzed according to the rules of the INPUT statement, except that reading starts with the next item, not the next record.

All file statements except ADVANCE, UPDATE, and CREATE may be used with ASCII files. See ASCII File Access in this section for restrictions on accessing ASCII files.

BINARY FILES

Binary files are unformatted files created through the MPE/3000 Operating System. Data items are stored in binary files as binary words without type information. When data is read from a binary file, it is assumed to be the type of the variable into which it is being read. Items in binary files can cross record boundaries and new records are used only upon overflow.

All file statements except ADVANCE, UPDATE, File LINPUT, and CREATE can be used with binary files. Access to binary files is discussed in detail under Binary File Access in this section.

FILE NAME

When any file is created, whether it is ASCII, binary, or BASIC formatted, it is assigned a file name by the user who creates the file. The file name may contain up to eight alphanumeric characters, the first of which must be a letter. The file name may be fully qualified as follows:

file name [/lockword] [.group name [.account name]]

If a *lockword* was specified for the file at its creation, this same *lockword* must follow the *file name* when the file is accessed.

The *group name* and *account name* specify a group and account other than those under which the user logged on. If the file is part of the user's group and account, then these qualifiers may be omitted.

The unqualified file name specifies a file in the user's log-on group and account that is not restricted by a lockword.

Refer to the *HP 3000 Multiprogramming Executive Operating System Reference Manual* (03000-90005A) for details of the file name specification.

Creating a Formatted File

A formatted file can be created by the CREATE command or through the CREATE statement. CREATE allocates a file of a specified size, assigns a name, and initializes each record with an end-of-file mark. Both the record size and the number of records can be specified. Neither of these sizes can be changed later.

Form

CREATE command:

CREATE file name, file length

CREATE file name, file length, record size

file name is a simple string without quotes. The *file length* is an integer constant. The optional *record size* is an integer constant.

CREATE statement:

CREATE numeric variable, file name, file length

CREATE numeric variable, file name, file length, record size

The *numeric variable* is used to return a result about the status of the file. *File name* is a string expression, *file length* is an integer expression, and the optional *record size* is also an integer expression.

Explanation

The file name may be fully qualified by a lockword, group name, and/or account name.

The file length specifies the number of records to be allocated to the file.

The record size specifies the number of data words per record. Record size may be between 4 and 319 words; the default size is 106. The most efficient record sizes are 106, 212, and 319. Record size is the number of words needed by the user to contain his data; the BASIC/3000 Interpreter adds format words to each record (see Appendix H).

The numeric variable contains one of the following results when the CREATE statement is executed:

0	successful file creation
1	a file already exists with the same name
2	the file was not created for some reason other than a duplicate name

An error message will be printed if the CREATE command cannot create the specified file name because of a duplicate name or some other reason.

CREATE does not open the file for access. Files are opened with a FILES or an ASSIGN statement (see Opening Files, this section).

EXAMPLES:

```
>CREATE AFILE,30

  10 DIM B$[4]
  20 LET B$="BB"
  30 CREATE N2,"BFILE",15,212
  40 CREATE N3,"XFILE",20
  50 CREATE N4,"AA",15
  60 CREATE N5,B$,15
  70 CREATE N6,"DFILE",20
  80 CREATE N7,"FF2",20
  90 CREATE N8,"XX",10
 100 PRINT "N2=";N2,"N3=";N3,"N4=";N4,"N5=";N5
 110 PRINT "N6=";N6,"N7=";N7,"N8=";N8
>RUN
N2= 0              N3= 0              N4= 1              N5= 0
N6= 0              N7= 0              N8= 0
```

Eight files are created, the file AFILE with a CREATE command, the remainder with CREATE statements. All but BFILE have the default record size 106; records in BFILE have 212 data words.

In this run, the value of N4 is 1 indicating that a file name AA already exists.

Note that the file name in the CREATE command is an unquoted string; in the CREATE statements, the file name may be a string expression such as a quoted string or a string variable.

Purging a File

An ASCII, binary, or BASIC formatted file can be deleted from the system with a PURGE command or PURGE statement.

Form

PURGE command:

PURGE file name

where *file name* is a simple string without quotes.

PURGE statement:

PURGE numeric variable, file name

The *numeric variable* will contain a result following execution of the PURGE statement. The *file name* is a string expression.

Explanation

The file specified in the statement or command is purged and is not recoverable.

The numeric variable in the statement returns a result on the status of the purge operation:

0	successful purge
1	file is being accessed and cannot be purged
2	user is not permitted to purge this file
3	there is no such file

EXAMPLES:

```
>PURGE AFILE
  10 PURGE N,"BFILE"
  20 PRINT N
>RUN
 0
```

A PURGE command is used to purge AFILE, a PURGE statement to purge BFILE. The result of purging BFILE is printed. Since it was a successful purge, the result is zero. If the PURGE command had been unsuccessful, a message would have been printed.

Opening Files

In order for a program to access a file, the file must be open. For every file that is to be opened, an association is established between the file number used in access statements and the file name. The file number is an integer between 1 and 16. The file name of a BASIC/3000 formatted file is the name assigned with a CREATE command or statement. The file name of an ASCII or binary file is the name assigned when the file was created under MPE/3000 control (see *MPE/3000 Operating System Manual* for instructions).

The linkage between file name and file number is accomplished by one of two statements: the FILES statement or the ASSIGN statement. FILES causes file numbers to be assigned to the files and, if a file name is specified, the file is opened. ASSIGN associates a file name with a file number reserved by FILES but not named. It opens a file not previously opened by FILES.

FILES is a declarative statement, not a dynamic statement. This means that it is not executed but is processed before the run begins. It may appear anywhere in the program.

ASSIGN, on the other hand, is a dynamic statement. It is executed during the program run and its position affects program execution. If ASSIGN is used to open a file, it must be executed before any statements used to access that file.

> *Note: Readers with experience using HP 2000 Time Shared Basic may be confused by the use of the term open as used for the BASIC/3000 Interpreter. OPEN for 2000 Time Shared Basic is equivalent to CREATE for BASIC/3000.*

CLOSING FILES

All files are closed automatically upon program termination. A file may be dynamically closed during program execution with the ASSIGN statement. This should be done wherever practical to release buffer space for other files.

FILES Statement

Every file that is to be accessed must have a file number designated in a FILES statement. Each file designator reserves a file number starting with number 1. Up to 16 file numbers may be reserved for any run. If the file designator names a file, the file will be opened.

Form

FILES file designator list

One or more *file designators* may be specified, separated by commas if more than one. A *file designator* may be one of the following:

file name

*

#integer

The *file name* identifies an existing file created with the CREATE command or statement, or an ASCII or binary file created in MPE/3000. It may be fully qualified.

The * reserves a file number for a file that will be named and opened with an ASSIGN statement.

#integer specifies the internal file number equivalent to an existing file number.

Explanation

File designators are associated with file numbers in the order in which they appear in the FILES statement. The first is assigned file number 1, the second file number 2, and so forth. If there is more than one FILES statement, file numbers are reserved starting with the first FILES statement.

When a file name is specified, the file is opened and the program is given both read and write access to the file unless the file is already open in some other program. In this case, only read access is allowed and a warning message will be printed at execution time. If the file cannot be opened, the program terminates.

When the program that opened a file terminates, the file is closed. If the program had read and write access to the file, the write restriction is removed. This enables the next program opening the file to have both read and write access.

When an asterisk (*) is used instead of a file name, the file number is reserved but the file to be associated with that number is not specified. The ASSIGN statement must be used to associate a file name with the file number. ASSIGN must be specified before any reference is made to the file number.

If #*integer* is used instead of a file name, it specifies an internal file number. This number identifies a file declared with a FILES statement in another program when programs are segmented. (See Section X, Segmentation.)

The FILES statement is declarative, not dynamic; it may appear anywhere in a program and is not executed.

EXAMPLES:

```
10 FILES AFILE,BFILE,*
20 FILES AA,BB,*,*,*
30 FILES #6
```

Five files are specified in the three FILES statement. The formatted files AFILE and BFILE have file numbers 1 and 2 respectively. Files AA and BB have file numbers 4 and 5 respectively.

The files reserved for numbers 3, 6, 7, and 8 must be assigned names in an ASSIGN statement before they can be accessed.

The file associated with local file number 9 in statement 30 has been previously associated with internal file number 6. For the relation between internal and local file numbers, see Files and Segmentation in Section X, "Segmentation".

ASSIGN Statement

The ASSIGN statement is used to assign a file name to a file number reserved by FILES and open the file. If another file was associated with the file number, that file is closed. The result of the open operation is returned following execution of ASSIGN. Unlike FILES, the ASSIGN statement is executed.

Form

The forms of ASSIGN are

ASSIGN file name, file number, numeric variable

ASSIGN file name, file number, numeric variable, mask

ASSIGN file name, file number, numeric variable, restriction

ASSIGN file name, file number, numeric variable, mask, restriction

*ASSIGN *, file number*

The *file name* is a string expression; the *file number* is an integer expression with a value between 1 and 16. The *numeric variable* returns the result of the ASSIGN execution. The optional *mask* is a string expression used to encode or decode file data. The optional *restriction* is a two-letter code to specify any access restrictions on the file.

Explanation

In the first four forms, the file name is associated with the file number and the file is opened.

If an * is used instead of the file name, any file previously associated with the file number is closed. If the file is already closed, the statement is ignored. Since closing a file releases the buffer space that was allocated to it, it is good practice to close unneeded files.

An error results if the specified file number exceeds the number of positions in the program's FILES statements.

After ASSIGN is executed, a value is returned to the numeric variable:

0	file is available for read and write
1	file is available for read only
2	(unused)
3	the file does not exist or is not accessible
4	(unused)
5	no buffer space is available for the file
6	file is not available for read or write because of another user's current access
7	specified restrictions not possible
8	file is available for write only

If the value returned is 3, 5, 6, or 7 the file is not opened and any access to the file number causes a fatal error. If the returned value is 1 any attempt to print onto the file causes a terminal error. If the returned value is 8, any attempt to read the file causes a terminal error. Other references to the file assigned that file number are legal.

A *mask*, if specified, protects the data in a file. Whenever the file is assigned the same mask should be used, or the data will not be intelligible when read. The actual data is scrambled or unscrambled using the mask; the data types and the end-of-file or end-of-record marks are not affected.

The *restriction* may be one of the following:

Code	Meaning
RR	Read and Write Restriction - no other user can access the file
WR	Write Restriction - other users may read, but not write on, the file
WL	Write Restriction with Dynamic Locking - current user has option to lock file; other users may read only
NR	No Restriction - current user has option to lock file; other users can read from and print on the file
NL	No Restriction with Dynamic Locking - no restriction but user has option to lock the file

If the *restriction* is omitted, the file is opened with WR restriction. If this fails, then NR restriction is used.

The specified restriction is placed on the file and remains in effect as long as the file is open. If another restriction is in effect due to a concurrent access, the ASSIGN statement will return the result 6, and the file is not opened.

Examples

```
 10 FILES AFILE,BFILE,*
 20 FILES AA,BB,*,*,*
 30 ASSIGN "XFILE",3,X1,WR
 40 ASSIGN "DFILE",6,D1,RR
 50 ASSIGN "FF2",7,N,NL
 60 LET X$="X"
 70 ASSIGN X$,8,X,"ABZ1"
 80 PRINT X1,D1,N,X
 90 ASSIGN *,8
100 ASSIGN "CC",1,C1
110 PRINT C1
>RUN
 0                  0                  0                  0
 0
```

Files are assigned for each file number associated with an * in the FILES statements. A write restriction on XFILE prevents other users from writing on that file. A read and write restriction on DFILE prevents other users from having any access to that file. File FF2 can be locked and unlocked with the dynamic locking statements LOCK and UNLOCK; there are no access restrictions on FF2.

The Mask "ABZ1" is used to encode the data in file X. File X is closed in line 90.

In line 100, AFILE is closed and file CC is assigned file number 1 and opened.

The zeros in the numeric variables indicate that each file was available for reading and writing in the current run when it was opened.

The following example shows the values returned from an ASSIGN on the same file during the *same* process versus an ASSIGN on the same file from two *different* processes.

Same Process

1st FOPEN \ 2nd FOPEN	RR	WR	WL	NR	NL
RR	6	6	6	6	6
WR	7	7	7	0	3
WL	7	7	7	3	0
NR	7	7	7	0	3
NL	7	7	7	3	0

Different Processes

1st FOPEN \ 2nd FOPEN	RR	WR	WL	NR	NL
RR	6	6	6	6	6
WR	7	7	7	1	3
WL	7	7	7	3	1
NR	7	7	7	0	3
NL	7	7	7	3	0

File Access

There are two types of access to a file: serial and direct. For serial access, the items read or written immediately follow the previous access without concern for the underlying record structure. A pointer associated with each open file always points to the next item in the file to be accessed.

For direct access, a particular record is specified at which the access begins. In this case, the pointer is moved to the beginning of this record.

In BASIC/3000 formatted files, direct and serial access can be combined in the same file. It is possible, for instance, to position the pointer to the beginning of a record with a direct file statement, and then to access the file serially from that point.

Binary files may be accessed directly only if they are disc files with fixed length records. Otherwise, serial access must be used for binary files (Binary File Access, this section).

ASCII files may be accessed by serial or direct access. For a complete description of the restrictions on ASCII file access, see ASCII File Access in this section.

Serial File PRINT

The Serial File PRINT statement writes data items on a file, starting at the current position of the pointer. The items may be numeric or string expressions.

Form

The forms of a Serial File PRINT statement are:

PRINT #file number; print list

PRINT #file number; print list, END

PRINT #file number

PRINT #file number; END

The *print list* is a series of numeric and/or string expressions. The rules for specifying the list are the same as those described for the PRINT statement in Section II.

If the *print list* is omitted, the statement is ignored unless the file is an ASCII file in which case, a line is skipped as in a PRINT statement.

Optionally, END can be the last (or only) item in the *print list*; it writes an end-of-file mark.

Explanation

Each item in the print list is written on the file in the order it appears in the Serial File PRINT statement. The items are written starting at the position where the pointer currently appears overlaying whatever data may be in that position in the file. Record boundaries are ignored; a serial PRINT can start in the middle of one record and end in the middle of another. Each data item must, however, fit into a single record.

If END is the last item in the print list, an end-of-file mark is written after the last data item. When an attempt is made to read the end-of-file, an end-of-file condition occurs. If data is written immediately following the END, it overlays the end-of-file mark. If END is not specified, an end-of-record mark is written after the last data item. This is also overlaid by a subsequent PRINT.

If printing is attempted beyond the physical end of the file, an end-of-file condition occurs. The ON END statement, described in this section, specifies action to be taken when an end-of-file condition occurs. If it is not specified, the program terminates.

Examples

```
 10 FILES AFILE,BFILE,*
 20 FILES AA,BB,*,*,*
 25 DIM A$[5]
 30 DIM B[2,5]
 40 MAT READ B
 50 DATA 100,200,300,400,500,600,700,800,900,1000
 60 LET A$="ABCDE"
 70 PRINT #1;"ARRAY B",(FOR I=1 TO 2,(FOR J=1 TO 5,B[I,J])),END
 80 PRINT #2;A$,B[2,1],B[1,5]
 90 PRINT #2;END
100 PRINT #1;"END OF ARRAY"
>RUN

>DUMP AFILE
ARRAY B
 100
 200
 300
 400
 500
 600
 700
 800
 900
 1000
END OF ARRAY
>DUMP BFILE
ABCDE
 600
 500
```

The string expression "ARRAY B" followed by the entire contents of array B are written onto file number 1 (AFILE). An end-of-file mark is written following the last data item.

In line 80, the contents of the string variable A$ followed by the contents of two elements of array B are written on file number 2 (BFILE). Line 90 writes an end-of-file mark on BFILE.

Line 100 overlays the end-of-file mark previously written on AFILE with the string expression "END OF ARRAY". Since the END is omitted, an end-of-record mark is automatically written after the string expression.

Although record boundaries are ignored in a serial print, no item can be longer than a single record. The record size of AFILE was created as 106 words and BFILE as 212. Each number requires one to four words depending on type, and a string requires a word for approximately every two characters (see Appendix H, "File Structure" for exact requirements).

Serial File READ

The Serial File READ statement reads items from a file specified by file number into numeric or string variables. The first item read is the item following the current position of the pointer, that is, immediately following the last item accessed. As with serial print, record boundaries are ignored and the list of read items can start in the middle of one record and end in the middle of another.

Form

The form of a Serial File READ is:

READ # file number; read item list

The read item list is a series of variables and/or FOR loops separated by commas. The rules governing this list are the same as those described for the READ statement in Section II.

Explanation

For a formatted file each item in the specified file is read into a variable in the read item list, the first item into the first variable, the second into the second, and so forth.

The destination for a string value must be a string variable; the destination for a numeric value must be a numeric variable. Otherwise, a terminal error occurs. If the numeric value is not the same data type as the variable, conversion is performed as described in Section IV.

It is possible to check the type of the next data item with the TYP function, described later in this section.

When an attempt is made to read beyond a logical or physical end-of-file, an end-of-file condition occurs. Unless an ON END statement transfers control to another statement in the program, the program terminates.

Examples

```
 10 FILES AFILE,BFILE,*
 20 DIM A$[5],X$[10],Y$[10],C$[15]
 30 DIM B[2,5]
 40 MAT READ B
 50 DATA 100,200,300,400,500,600,700,800,900,1000
 60 LET A$="ABCDE"
 70 PRINT #1;"ARRAY B",(FOR I=1 TO 2,(FOR J=1 TO 5,B[I,J])),END
 80 PRINT #2;A$,B[2,1],B[1,5]
 90 PRINT #2;END
100 PRINT #1;"END OF ARRAY"
110 RESTORE #2
120 RESTORE #1
130 READ #1;X$,A1,B1,C1,D1,E1
140 PRINT X$,LIN(1),A1,B1,C1,D1,E1
150 READ #1;A2,B2,C2,D2,E2
160 PRINT A2,B2,C2,D2,E2
170 READ #2;Y$,A,B
180 PRINT Y$,A,B
190 READ #1;C$
195 PRINT C$
200 READ #1;X
210 REM..ATTEMPT TO READ END-OF-FILE CAUSES TERMINATION
>RUN

ARRAY B
 100             200             300             400             500
 600             700             800             900             1000
ABCDE            600             500
END OF ARRAY
END OF FILE IN LINE 200
```

After data is written on files 1 and 2 with the print statements in lines 70-100, and the pointer is restored to the start of each file in lines 110 and 120, the data that was written can be read.

The first six items in file 1 are read in line 130. The next five items are read in line 150. The three items written on file 2 are read in line 170. PRINT statements are inserted to test the accuracy of the reads and the previous writes.

A string item remains in file number 1; this is read in line 190. Line 200 attempts to read an end-of-file causing the message: END OF FILE IN LINE 200 to be printed.

File RESTORE Statement

The File RESTORE statement repositions the file pointer to the start of the file. The statement can be used for any file, but is particularly useful for serial files such as magnetic tape.

Form

RESTORE # file number

The *file number* identifies a file that is currently open.

Explanation

When File RESTORE is executed, the file pointer is set to point to the beginning of the first record in the file. A serial read or print will begin at that position.

Example

```
 10 FILES AFILE,BFILE
 20 PRINT #1,1;123.4
 30 PRINT #2,1;567.8
 40 RESTORE #2
 50 RESTORE #1
 60 READ #1;C
 70 READ #2;D
 80 PRINT C,D
>RUN
123.4          567.8
```

When the File RESTORE statements are executed, the pointer in file number 2 is moved back to the start of that file. Then the pointer in file number 1 is moved to the start of that file. If the files are magnetic tape, they are rewound.

Direct File PRINT

The Direct File PRINT statement writes a list of data items onto the specified file as a single record. Printing begins at a particular record specified in the PRINT statement. After printing, an end-of-record mark is written and any data previously contained in the record is lost. Data in records preceding and following the specified record is not changed.

Form

The forms of a Direct File PRINT are:

PRINT # file number, record number; print list

PRINT # file number, record number; print list, END

PRINT # file number, record number

PRINT # file number, record number; END

Both the *file number* and *record number* are integer expressions. The *print list* is optional and has the same format as a Serial File PRINT. If it is missing, the statement erases the contents of the specified record.

Explanation

The Direct File PRINT positions the pointer at the beginning of the specified record and then writes the contents of the print list. An end-of-record mark is written following the items in the print list. Any previous end-of-record marks are ignored.

The first record of the file is record number 1.

END writes an end-of-file mark. If no print list is specified, any data in the specified record is replaced by the end-of-file mark.

Serial and Direct PRINT statements can be used to write on the same file. A serial print following a direct print will write its data items immediately following the previous items.

Examples

```
 10 FILES AFILE,BFILE,*
 20 FILES AA,BB,*,*,*
 30 ASSIGN "XFILE",3,X1,NL
 40 ASSIGN "DFILE",6,D1,RR
 50 LET A1=1,B1=2,C1=3,D1=4,E1=5
 60 LET A$="A"
 70 FOR N=1 TO 10
 80   LET B[N]=N+1
 90 NEXT N
100 DIM B[10]
110 PRINT #3,1;"START OF XFILE"
120 PRINT #3,2;10,A1,(FOR N=1 TO 10,B[N])
130 REM..TWO RECORDS HAVE BEEN WRITTEN ON XFILE
140 PRINT #6,2;B1,C1,D1,E1
150 PRINT #6,1;A$
160 PRINT #6,3;END
170 REM..THE THIRD RECORD OF DFILE IS AN END-OF-FILE
>RUN
```

The first record of file number 3 contains a string value. The second record has two numeric items and the contents of a 10-element numeric array. No end of file is written on #3.

File number 6 also has a string item in its first record; it has four numeric items in the second record. The third record is an end-of-file.

Note that the records do not have to be written in the order they appear in the file.

Direct File READ

The Direct File READ statement reads data values starting at a specified record of a specified file and assigns them to variables. Numeric values can be assigned only to numeric variables, and string values to string variables, as in Serial File READ.

Form

The forms of the Direct File READ statement are:

READ # file number, record number; read item list

READ # file number, record number

The *file number* and *record number* are integer expressions. The optional *read item list* is of the same form as in a READ statement.

Explanation

Data values are read from the record and assigned to the variables in the item list. If a record number is specified outside the range of the file, an end-of-file condition occurs.

If the *read item list* is omitted, the statement moves the file pointer to the beginning of the specified record, but does not read any data.

Example

```
  10 DIM C[2,5]
  20 DIM A$[20],X$[20]
  30 READ #3,1;A$
  40 READ #3,2;X,Y,(FOR N=1 TO 2,(FOR P=1 TO 5,C[N,P]))
  50 READ #6,1;X$
  60 PRINT A$,X,Y
  70 MAT PRINT C
  80 PRINT X$
  90 FILES *,*,XFILE,*,*,DFILE
>RUN

START OF XFILE   10               1
 2               3                4               5               6

 7               8                9               10              11
A
```

In this example, the data previously written on records 1 and 2 of XFILE and on record 1 of DFILE are read. (See examples with Direct File PRINT.)

In the example below, ten records are written on file AA and then these records are copied to file BB and AA is closed.

```
 10 DIM X[10]
 20 MAT READ X
 30 ASSIGN "AA",1,A
 40 FOR R=1 TO 10
 50   PRINT #1,R;X[R]
 60 NEXT R
 70 PRINT #1;END
 80 ASSIGN "BB",2,B
 90 FOR R=1 TO 10
100   READ #1,R;X
110   PRINT #2,R;X
120 NEXT R
130 PRINT #2;END
140 PRINT "FILE AA COPIED TO FILE BB"
150 ASSIGN *,1
160 DATA 10,20,30,40,50,60,70,80,90,100
170 FILES *,*
>RUN
FILE AA COPIED TO FILE BB
```

ASCII File Access

ASCII files may be accessed with any statements except the ADVANCE, UPDATE or CREATE statements. In addition, the File LINPUT statement can be used to read the contents of an ASCII record.

Numeric values written on an ASCII file with a File PRINT statement cannot be read easily since the File READ statement expects commas between items, and the File PRINT does not write the commas on an ASCII file. To avoid this problem print "," between items if the file is to be read with a File READ statement, or use the File LINPUT statement and the CONVERT statement.

String values are printed exactly as they are printed by the PRINT statement (Section II).

File LINPUT Statement

The File LINPUT statement reads the entire contents of a record in an ASCII file into a string variable. File LINPUT is used to read ASCII files only.

Form

LINPUT # file number; string variable
LINPUT # file number, record number; string variable

Explanation

File LINPUT reads the contents of the record at which the pointer is currently positioned or at the specified record. This is like LINPUT (see Section II) except that input is from a file, not a terminal, and a record, not a line, is read.

If the string variable is not large enough to contain the entire record, the extra characters are discarded.

Example

```
 10 DIM A$[72],B$[20]
 20 READ A1,B1,C1,D1,E1
 30 RESTORE #10
 40 PRINT #10;A1,B1,C1,D1,E1
 50 PRINT #10;"1, JANUARY,1973"
 60 RESTORE #10
 70 LINPUT #10;A$
 80 LINPUT #10;B$
 90 PRINT A$,B$
100 DATA 10,20,30,40,50
110 FILES AFILE,BFILE,*,AA,BB,DFILE,FF2,X,*,ASCI
>RUN
 10              20              30              40              50
1, JANUARY,1973
```

The first two records of the ASCII file ASCI are input into the string variables A$ and B$, the first record in A$ and the second in B$.

Binary File Access

Access to binary files may be serial or direct as with BASIC formatted files. Direct access is allowed only for files on disc with fixed length records. Serial access is allowed for all files.

Data items are stored in binary files as binary words without type flags. When data is read from a binary file, it is assumed to be the same type as the variable into which it is being read. Items in binary files can cross record boundaries and new records are used only upon overflow.

Printing Strings

When a string item is printed, the item starts at a word boundary and is as long as the actual string. If the string is of odd length, an additional character of undefined value is written to complete the last word. Each word contains two characters.

Reading Strings

When a string is read from a binary file, the number of characters read depends on the form of the variable. For instance, if A$ is a simple string variable:

READ #1;A$	reads the physical length of A$
READ #1;A$(I)	reads the physical length of the substring starting at I
READ #1;A$(I,J)	reads J-I+1 characters into the substring starting at I
READ #1;A$(I;J)	reads J characters into the substring starting at I

Dynamic Locking

If a file is opened with an ASSIGN statement and either WL or NL is specified as a restriction, access to the file can be dynamically controlled with the LOCK and UNLOCK statements.

Form

LOCK # file number

UNLOCK # file number

The file identified by *file number* must have been opened with an ASSIGN statement specifying one of the restrictions WL or NL.

Explanation

LOCK gives a program exclusive control of a file until it is unlocked by the UNLOCK statement. During control by LOCK, no other program can lock the file until UNLOCK is executed. An attempt to lock a file that has been locked by another program will cause the program to be suspended until the file has been unlocked in the other program. Only one file at a time can be locked, although WL or NL may be specified for more than one file. Any write operations on a locked file are guaranteed to be physically completed before the UNLOCK is executed. Each access to a file during dynamic locking should be made between a LOCK and UNLOCK statement.

Note that the LOCK statement does not actually restrict other programs from accessing the file. Therefore, all programs must cooperate by first locking, then accessing, and then unlocking the file. Dynamic locking is not necessary if it is unlikely that more than one user will access the same file, or if none of the users are writing on the file.

Examples

```
  10 FILES AFILE,BFILE,*
  20 FILES AA,BB,*,*,*
  30 ASSIGN "X",8,T,NL
  40 PRINT T
  50 LET A=2.57325E13
  60 LOCK #8
  70 PRINT #8;A
  80 UNLOCK #8
>RUN
LOCK
 0
```

The file is locked in line 60. This assures that no other program can lock the same file, and that the write operation in line 70 will be completed and the pointer moved to the beginning of the next record. Line 70 contains a Serial File PRINT statement that writes the contents of the variable A onto file number 8. Line 40 prints the contents of the numeric variable T containing the result of the ASSIGN. This shows that the file is open and can be written on before LOCK is specified. Following execution of the UNLOCK statement, in line 80, any other user may lock the file.

ON END Statement

The ON END statement sets a flag for a specified file so that if and when an end-of-file condition occurs in reading and writing that file, control is transferred to a specified statement. If the flag is not set, an end-of-file condition causes program termination.

Form

The forms of ON END are

> *ON END # file number THEN label*
> *IF END # file number THEN label*

ON END and IF END are accepted interchangeably but the statement is always stored internally and listed as ON END.

Explanation

When an end-of-file condition occurs during execution of a Direct or Serial File READ statement or a File LINPUT statement, the ON END statement transfers control to the statement identified by *label*. When writing on a file with a Direct or Serial File PRINT statement, ON END transfers control to *label* when an attempt is made to write past the physical end-of-file.

ON END is an executable statement and the transfer label can be altered by a subsequent ON END statement. The label must not lie within the range of a function unless the ON END is also within that function. An ON END within a function must refer to a label within that function; ON END has no effect outside the function.

ON END is not executed if an end-of-file is encountered during execution of the ADVANCE statement (see ADVANCE description, this section).

Examples

In the example below, the Direct File READ in line 70 attempts to read an end-of-file written in line 50. The ON END statement transfers control to line 100.

```
  10 FILES AFILE,BFILE
  20 READ A1,B1,C1,D1,E1
  30 DATA 100,200,300,400,500
  35 ON END #2 THEN 100
  40 PRINT #2,1;A1,B1,C1
  50 PRINT #2,2;D1,E1,END
  60 READ #2,1;A,B,C
  70 READ #2,2;D,E,F
  80 END
 100 PRINT "END OF FILE 2"
 120 PRINT A,B,C,D,E
>RUN
END OF FILE 2
 100            200            300            400            500
```

In the next example, file RR is created with 5 records. The File PRINT statement in line 100 attempts to write past the end-of-file and the ON END statement causes a transfer to line 130.

```
>CREATE RR,5
  10 FILES RR
  20 DIM X[6]
  30 MAT READ X
  40 ON END #1 THEN 130
  50 PRINT #1,1;X[1]
  60 PRINT #1,2;X[2]
  70 PRINT #1,3;X[3]
  80 PRINT #1,4;X[4]
  90 PRINT #1,5;X[5]
 100 PRINT #1,6;X[6]
 110 DATA 10,20,30,40,50,60
 120 END
 130 PRINT "END OF FILE"
>RUN
END OF FILE

>DUMP RR
 10
 20
 30
 40
 50
```

ADVANCE Statement

The ADVANCE statement allows for skipping past items in a BASIC/3000 formatted file without reading them.

Form

The form of ADVANCE is

ADVANCE # file number; integer expression, numeric variable

The *integer expression* specifies the number of data items to be skipped and the *numeric variable* is used to return a result value.

Explanation

If the integer expression is negative, items are skipped in a reverse direction.

After execution of ADVANCE, the numeric variable equals zero if the ADVANCE was successful. If the ADVANCE statement encountered either an end-of-file or a start-of-file, the numeric variable equals the difference between the number of items requested and the number actually skipped. This value is negative if ADVANCE was in the reverse direction.

Example

```
  10 FILES *,*,AA,BB
  20 LET A=1,B=2,C=3,D=4,E=5,X$="X"
  30 PRINT #3;A,B,C,D,E,X$,END
  40 RESTORE #3
  50 ADVANCE #3;3,X1
  60 IF X1<>0 THEN GOTO 100
  70 READ #3;L,M
  80 READ #3;A$
  90 PRINT L,M,A$
 100 PRINT X1
>RUN
 4                  5                  X
 0
```

The first three items in file AA are skipped, then the next three are read and printed.

UPDATE Statement

The UPDATE statement allows an item in a BASIC/3000 formatted file to be modified without affecting any of the items that precede or follow. UPDATE overwrites the next item in the file and positions the pointer to the item that follows.

Form

The form of UPDATE is

UPDATE # file number; expression

Explanation

If the existing data item is numeric, the *expression* to be written must be numeric also. If their types do not match, the value of the expression is converted to the type of the existing data item.

If the existing item is a string, the expression must be a string. The string expression is truncated or blank-filled on the right to fit the size of the existing item exactly.

Examples

```
  10 FILES *,*,AA
  20 LET A=1,B=2,C=3,D=4,E=5
  30 LET A$="A",B$="B"
  40 PRINT #3,1;A,B,C,D,E,A$,B$
  50 RESTORE #3
  60 ADVANCE #3;3,X1
  70 UPDATE #3;4.57
  80 ADVANCE #3;2,X2
  90 UPDATE #3;"XYZ"
>RUN

>DUMP AA
 1
 2
 3
 4.57
 5
A
X
```

The fourth and seventh items in file AA are given new values with the UPDATE statement. The file is then dumped to illustrate the successful update.

Listing File Contents

DUMP COMMAND

The DUMP command displays the contents of a BASIC/3000 formatted file on another file: either the normal output file or a specified ASCII file. DUMP provides a simple way to print file contents at the terminal. Normally the contents of a file to be dumped are string data.

Form

DUMP file name

DUMP file name, OUT=asciifile

file name is a simple string without quotes. It must name a BASIC/3000 formatted file. *asciifile* is the name of an existing ASCII file.

Explanation

The contents of the named file are dumped on the normal output file (the terminal) unless *OUT=asciifile* is specified, in which case, the file is dumped on the specified ASCII file.

DUMP prints each item on a separate line; record boundaries are not indicated. DUMP terminates at the first end-of-file.

Example

```
  10 FILES AFILE
  20 LET X=10.5,Y=75,Z=150
  30 PRINT #1,1;10,20,30
  40 PRINT #1,2;"HELLO"
  50 PRINT #1,3;X,Y,Z
>RUN

>DUMP AFILE
 10
 20
 30
HELLO
 10.5
 75
 150
```

File Functions

Two functions are available in BASIC/3000 that assist in file access. They are TYP and REC.

TYP FUNCTION

The TYP function returns the type of the next data item for a particular file. This function is used in conjunction with File READ statements since the variables into which the data is read must be string if the item is string, numeric if the item is numeric.

Form

TYP (integer expression)

The absolute value of the *integer expression* must be an existing file number.

Explanation

The value returned by TYP depends on the type of the next data item in the file.

TYP(x)	Meaning
1	real
2	string
3	end-of-file
4	end-of-record
5	integer
6	long
7	complex

If the *integer expression* is greater than zero, the file is treated as a serial file. This means that the value 4 is never returned as end-of-record marks are skipped in a serial file.

If the *integer expression* is less than zero, the file is treated as a direct file and any of the values of TYP may be returned. Since the file number is based on the absolute value of the expression, an expression equal to -1 means that file number 1 is examined and treated as a direct file.

If the *integer expression* equals zero, TYP returns a result based on the current position of the pointer to the DATA statements (see READ/DATA/RESTORE description, Section II.) The value 4 is never returned, and 3 means end-of-data.

If the file is binary, TYP returns only the values 1, 3, or 4.

If the file is ASCII, TYP returns the same results with the same meaning as the BUF function. (See BUF Function description under INPUT Statement, Section II.)

Examples

File AFILE is written with integer, real, long, and complex numbers. A later program reads only the long and complex items using the TYP function to distinguish them:

```
  10 FILES AFILE
  20 INTEGER A,B
  30 LONG L1
  40 COMPLEX C1
  50 LET A=25,B=100,X=2.357E12,Y=7.649E-5
  60 LET L1=9.99999L-5
  70 LET C1=(.002359,.002175)
  80 PRINT #1;A,X,L1,B,Y,C1,"STRING RECORD",END
>RUN

>SCRATCH

  10 FILES AFILE
  20 LONG L3
  30 COMPLEX F1
  40 GOTO TYP(1) OF 50,50,130,50,50,70,100
  50 ADVANCE #1;1,A1
  60 GOTO 40
  70 READ #1;L3
  80 PRINT L3;
  90 GOTO 40
 100 READ #1;F1
 110 PRINT F1;
 120 GOTO 40
 130 PRINT LIN(1);"END OF FILE"
>RUN
 9.99999000000000L-05   ( 2.35900E-03,  2.17500E-03)
END OF FILE
```

REC FUNCTION

The REC function returns the record number at which the file pointer is currently positioned for a specified file.

Form

REC(integer expression)

The *integer expression* evaluates to a file number. REC returns the record number of the record in that file currently being accessed.

Example

```
  10 FILES AFILE,BFILE,XFILE,AA,BB
  20 LET A=1,B=2,C=3,D=4,E=5
  30 PRINT #5;A,B,C,D,E
  40 LET R=REC(1)
  50 READ #5,R;V,W,X,Y,Z
  60 PRINT V,W,X,Y,Z
>RUN
 1                  2                  3                  4                  5
```

The variable R is set to the value of REC(1); this value, the current record in file BB, is then used in the Direct File READ in line 50.

File Array Operations

There are four statements for accessing files with arrays:

Serial File MAT PRINT
Serial File MAT READ
Direct File MAT PRINT
Direct File MAT READ

SERIAL FILE MAT PRINT STATEMENT

This statement prints entire arrays on a file starting at the current position.

Form

MAT PRINT #file number; array list

MAT PRINT #file number; array list, END

MAT PRINT #file number; END

The *array list* contains a list of array names separated by commas. END writes an end-of-file mark.

Explanation

The arrays specified in the array list are written on the specified file row by row. END, whether alone or after the array list, writes an end-of-file.

SERIAL FILE MAT READ STATEMENT

This statement reads data items, starting with the current position of a specified file, to fill entire arrays row by row.

Form

MAT READ # file number; array list

Explanation

The array names in the array list will contain data read from the specified file.

Example

The example below writes the values of two arrays onto the file BB; it then reads the array value into two different arrays with different dimensions:

```
  10 FILES *,*,*,BB
  20 DIM A[2,5],B[3,2]
  30 MAT READ A
  40 MAT READ B
  50 DATA 10,20,30,40,50,60,70,80,90,100,1,2,3,4,5,6
  60 MAT PRINT #4;A,B
  70 RESTORE #4
  80 DIM C[5,2],D[2,3]
  90 MAT READ #4;C,D
 100 MAT PRINT C,D
>RUN
 10               20

 30               40

 50               60

 70               80

 90               100

 1                2                3

 4                5                6
```

DIRECT FILE MAT PRINT STATEMENT

This statement prints arrays starting at the beginning of a specified record within a specified file.

Form

MAT PRINT # file number, record number; array list

MAT PRINT # file number, record number; array list, END

MAT PRINT # file number, record number; END

Explanation

The array names in the list are written on the file starting at the beginning of the record identified by record number. An END in the array list causes an end-of-file mark to be written. The contents of the array list may cross record boundaries.

DIRECT FILE MAT READ STATEMENT

This statement reads entire arrays starting from a specified record in a specified file. The read may cross record boundaries.

Form

MAT READ # file number, record number; array list

The *array list* contains the names of entire arrays.

Explanation

The contents of the specified record number are read into the array variables specified in the array list.

Examples

Two arrays A and B are written on MAT1 starting at record number 5. The arrays are read from record number 5 into arrays C and D respectively.

```
 1Ø  FILES *,*,*,*
 2Ø  DIM A[1Ø],B[3,2],C[5,2],D[2,3]
 3Ø  MAT READ A,B
 4Ø  DATA 1,2,3,4,5,6,7,8,9,Ø
 5Ø  DATA 6.5,7.4,8.3,9.2,.1,5.5
 6Ø  ASSIGN "MAT1",4,M1
 7Ø  MAT PRINT #4,5;A,B
 8Ø  MAT READ #4,5;C,D
 9Ø  MAT PRINT C,D
>RUN
 1              2

 3              4

 5              6

 7              8

 9              Ø

 6.5            7.4            8.3

 9.2            .1             5.5
```

SECTION IX
Formatted Output

The PRINT USING and IMAGE statements of BASIC/3000 give the user explicit and exact control over the format of his program output. All types of numbers can be printed: integer, fixed-point, floating-point, and complex. The exact position of plus and minus signs can be specified. String values can be printed in specified fields; literal strings and blanks can be inserted wherever needed. Carriage return and line feed are under explicit control and lines longer than 72 characters can be printed.

Format strings are used to specify the output format. These strings are explicitly included in the PRINT USING statement, or they may be specified in IMAGE statements whose labels are referenced in the PRINT USING statements.

PRINT USING Statement

The PRINT USING statement allows the user to output a list of items according to a customized format.

Form

The forms of PRINT USING are:

PRINT USING label;print using list

PRINT USING label

PRINT USING string expression;print using list

PRINT USING string expression

The *print using list* is an optional list of expressions and functions from which items are printed. It is like a *print list* (see PRINT Statement, Section II) except that semicolons and trailing punctuation are not allowed.

The *label* is the line number of an IMAGE statement that contains a format string for the PRINT USING. The *string expression* evaluates to a format string for the PRINT USING.

The *print using list* can be omitted when the format consists entirely of literal specifications.

Explanation

A format string describes the form in which items in the *print using list* are to be printed. The full description of format strings is contained under Format Strings, this section.

Any commas in the print using list are separators only; they have no formatting function as they do in PRINT.

When PRINT USING is executed, each specification in the associated format string is extracted and examined. If the specification calls for a string or numeric value, the print using list is examined for a corresponding expression. Each expression is output according to its corresponding specification in the format string. Any print functions in the list are executed as they are encountered.

If the expression and the specification do not match because one is a string and the other is numeric, the program terminates with an error message. If the value of a numeric expression is greater than can be printed with the format, the number is printed without format control, and preceded by two asterisks, on a separate line. The program continues. An integer specification prints a real number with any fraction rounded to the nearest integer.

If the format specification is a literal specification, it is printed without examining the print using list. When the format string contains only literal specifications, the print using list, if present, is ignored.

If the end of the format string is reached before the end of the print using list, processing returns to the beginning of the string. The next expression in the list is matched to the first specification in the format string. When all expressions in the list have been printed, a carriage return and line-feed are generated unless a carriage control character specified in the format string suppresses the final carriage return and/or linefeed.

Examples

Numeric variables A and B are printed according to a format specified as a string expression in PRINT USING (line 30) and then according to a format specified in the IMAGE statement (line 50). PRINT USING in line 40 references the format string in line 50:

```
  10 DIM A$[10]
  20 LET A=100000,B=999999
  30 PRINT USING "10D";A,B
  40 PRINT USING 50;A,TAB(23),B
  50 IMAGE $DXDDDXDD
>RUN
    100000     999999
$1 000 00                  $9 999 99
```

The exact meaning of the format strings used in these examples is described below under Format Strings. Note here that:

integers are replicators causing repetition of the specification that follows

X causes a blank to be printed

D prints a decimal digit from the print using list

$ causes a dollar sign to be printed

MAT PRINT USING Statement

The MAT PRINT USING statement allows the printing of one or more complete arrays according to a specified format.

Form

The form of MAT PRINT USING is:

MAT PRINT USING label

MAT PRINT USING label;mat print using list

MAT PRINT USING string expression

MAT PRINT USING string expression;mat print using list

The *mat print using list* is a list of arrays and print functions.

The format string is in an IMAGE statement identified by *label*, or it is contained in the *string expression.*

Explanation

The format specifications for printing the arrays in the list are provided and evaluated in the same manner as for PRINT USING. The arrays are printed row by row as in the MAT PRINT statement, but all spacing is provided by the format string associated with the mat print using list.

Examples

```
  10 DIM A[3,7],B[3,5]
  20 MAT READ A,B
  30 MAT PRINT USING 100;B
  40 MAT PRINT USING "7(3D2X)/";A
  50 DATA 100,110,120,130,140,150,160,170,180,190,200
  60 DATA 210,220,230,240,250,260,270,280,290,300
  70 DATA 400,410,420,430,440,450,460,470,480,490
  80 DATA 500,510,520,530,540
 100 IMAGE 5(DXDDXX)/
>RUN
4 00  4 10  4 20  4 30  4 40
4 50  4 60  4 70  4 80  4 90
5 00  5 10  5 20  5 30  5 40

100  110  120  130  140  150  160
170  180  190  200  210  220  230
240  250  260  270  280  290  300
```

Array A is printed according to the string expression in line 40; array B is printed according to the format in the IMAGE statement in line 100, which inserts a blank after the first digit of each number.

IMAGE Statement

The IMAGE statement specifies a format string to be used in a PRINT USING statement. It is a declarative statement and is not executed.

Form

The form of the IMAGE statement is

IMAGE format string

The *format string* is not quoted. It may be used by PRINT USING or MAT PRINT USING statements.

Format string specification is described in this section under Format Strings.

Explanation

The format string specified in an IMAGE statement is associated through the label of the IMAGE statement with one or more formatted print statements. By specifying a format in an IMAGE statement, many PRINT USING and MAT PRINT USING statements can use the same format.

The legitimacy of the format string in the IMAGE statement is not checked until execution of the PRINT USING statement referencing it.

Examples

```
   10 DIM A$[20]
   20 A$="1,2,3,4,5,6,7,8,9,0,"
   30 A=123456,B=-789100,C=12
   40 X=1234.56
   50 IMAGE 6D2X,"LITERAL",S2DXX,$4D.2D
   60 IMAGE S4DXX.DD/20AX,S7D
   70 PRINT USING 50;A,C,X
   80 PRINT USING 60;X,A$,B
>RUN
123456  LITERAL+12  $1234.56
+1234  .56
1,2,3,4,5,6,7,8,9,0,  -789100
```

The decimal point in the format string indicates that a decimal point should be printed and, by its position, specifies the number of digits to the left and right of the decimal. The slash causes a carriage return and linefeed.

Format Strings

The format string consists of an optional carriage control character and comma followed by one or more format specifications separated by one or more commas and/or slashes. Each format specification consists of orderly combinations of format symbols.

Format Symbols

Format specifications are composed of permutations of the following symbols (the replicator *n* is an integer between 1 and 255 inclusive):

Symbol	Meaning
A	Causes the printing of a single character. *n*A causes *n* characters to be printed. A is legal only for string data items.
D	Causes a decimal digit to be printed. *n*D causes *n* decimal digits to be printed.
X	Causes a blank to be printed. *n*X causes *n* blanks to be printed.
.	Indicates placement of decimal point.
S	Indicates placement of "+" and "–" signs.
M	Indicates placement of "–" only.
E	Causes numbers to be printed in E-type float form.
C	Indicates a complex format follows.
K	Indicates compressed numeric formatting.
I	Causes the character "I" to be printed; *n*I causes *n* I's to be printed.
$	Causes the character "$" to be printed; *n*$ causes *n* dollar signs to be printed.
+ or –	Separate the real from the imaginary part in a complex number and indicate placement of the sign for the imaginary part.
/	Separates specifications and generates a carriage return and linefeed.
( )	Enclose a group field specification so that a replicator can be used to signify the multiple occurrence of this group of specifications. Also enclose complex specification preceded by C.
,	Separates specifications
" "	Enclose literal specifications.
'	Apostrophe is used for special string characters (e.g., '40)

The format symbols are combined according to certain rules to form the following types of specifications:

- literal specification
- string specification
- numeric specification

The symbols X, I, $, and literal strings are considered editing symbols and may be interspersed freely in any specification.

Literal Specification

This specification contains only the editing symbols X, I, $, and literal strings. When a literal string is encountered in a format string, the characters between quotes are printed. For instance,

"THIS IS A LITERAL STRING"

When the characters I and $ are encountered in a format string, the literal characters "I" and "$" respectively are printed.

Each X in a format string causes a single blank character to be printed.

Replicators may precede an X, I, or $ in a format string:

2I	equivalent to II
3X	equivalent to XXX
2$5X	equivalent to $$XXXXX

String Specification

In a string specification, any combination of editing symbols and A's (with or without replicators) is permitted. At least one A must be present to signify a string specification. The matching item in the print using list must be a string value; it is printed left-justified with blank fill on the right. If necessary, the string value is truncated on the right to fit in the field. For example,

3A	(prints 3 characters adjacent)
AXAXA	(prints 3 characters separated by blanks)

Examples

```
  10 DIM A$[10],B$[3]
  20 LET A$="ABCDEFGHIJ",B$="ABC"
  30 PRINT USING "3X3A3X,AXAXAX";B$,"UVWX"
  40 PRINT USING 50;A$
  50 IMAGE "ALPHA STRING",3X10A
>RUN
   ABC   U V W
ALPHA STRING   ABCDEFGHIJ
```

Numeric Specification

The specification for a numeric field other than complex consists of the symbols S, M, D, ".", and E. S and M may be used to position the sign, D's are used to indicate the positions of the digits, the "." is used to position the decimal point, and E may be used to specify that an exponent field is to be printed. The editing symbols may be freely inserted into a numeric field. The specification for a complex field usually consists of two numeric specifications combined in a special way that is described in detail later.

Generally, numbers are rounded when being converted for formatted output.

In all numeric specifications (except the imaginary part of a complex specification) printing of signs is handled by the characters S and M, according to the following rules:

1. If an S appears *before* all D's and any decimal point, the sign (whether + or -) is printed immediately preceding either the most significant digit or the decimal point, whichever is further to the left.

2. If an S appears after any D's or to the right of the decimal point, the sign is always printed in that particular position (i.e., a fixed sign).

3. If an M appears anywhere in a specification, either a blank is printed in that position (if the number is positive or zero) or a "-" is printed in that position (if the number is negative).

4. If no S or M appears anywhere in the specification, the number is printed as specified (if the number is zero or positive). If the number is negative, the specification is treated exactly as if the first D were replaced by an S, provided at least one D precedes the decimal point.

5. Only one S or M is allowed per specification.

If the field specified by any numeric specification is not large enough to hold the most significant digits of a number to be printed, the number is printed alone on the next line preceded by two asterisks. It is printed in a standard format according to type as if generated by a PRINT statement.

INTEGER SPECIFICATION. Any combination of editing symbols and D's (with or without replicators) is allowed in an integer specification, but at least one D must be present. One S or M can appear. The number is printed right-justified, one digit per D with leading zeros suppressed and the fractional part (if any) rounded to the nearest integer.

Examples

```
  1Ø PRINT USING 2Ø;-1,1,54ØØ,-19.3Ø1,-74,1Ø3.65
  2Ø IMAGE XDXDXD,DM2DX,S4D,3X5D,DDDDM,X$4D
>RUN
   - 1   1 +54ØØ      -19   74- $ 1Ø4
```

Note that the fractional part of decimal numbers are dropped after rounding.

FIXED SPECIFICATION. Any combination of editing symbols and D'S (with or without replicators) is allowed in a fixed specification, but at least one D and only one period must be present. One S or M can appear. The number is printed anchored around the position of the decimal point (which is printed) with leading zeros suppressed. If necessary, the fractional part of the number is rounded or filled with zeros before being printed to the right of the decimal point. One digit or blank is printed per D. Trailing zeros are printed.

Examples

```
 1Ø PRINT USING 2Ø;7Ø32,-4.29374,21,-.ØØ1,47Ø.32,1.9
 2Ø IMAGE 3XDS3D.5D,MDD.D,3D.X1ØD,DDD.DD,XMDDD.4D,DD

>RUN
   7+Ø32.ØØØØØ- 4.3 21. ØØØØØØØØØØ  -.ØØ  47Ø.32ØØ 2
```

FLOATING SPECIFICATION. A floating specification is an integer or fixed specification followed by E. Editing symbols may be interspersed freely. The most significant digits are printed from left to right, and an exponent is printed as E±dd. If a negative number is to be printed in floating point format, there must be an S or M in the specification.

Examples

```
  10 PRINT USING 20;74.92,-400000,(32.1,4),1E12,1.5L21
  20 IMAGE DDEX,M.DDEXX,4DEX,2(5D.EX)
>RUN
75E+00 -.40E+06  3210E-02 10000.E+08 15000.E+17
```

Note that the imaginary part of the complex number is not printed.

COMPLEX SPECIFICATION. The real part of a complex number can be printed using an integer, fixed, or floating specification. The imaginary specification is preceded by a + or - and consists of an integer, fixed, or floating specification *in which there is no S or M.* The only difference is in the treatment of the sign. The + is treated exactly like an S; the - is treated as an X if the number is not negative and as an S if the number is negative.

To print both the real and imaginary parts of a complex number, the real specification precedes the imaginary specification in the same format with no intervening comma.

An I is not printed automatically to indicate imaginary, but if desired, can be included in the format string as a literal or by using the I symbol.

Examples

```
  10 PRINT USING 30;(3,2),(0,1),(.003,-4),(.003,4),21
  30 IMAGE 3D+DDX,3D.D-.DE,2(X.3D-D.D,I),XXXDD
>RUN
  3 +2     .0 .1E+01 .003-4.0I .003 4.0I   21

  10 PRINT USING "XXXDD,+DD";(21,0),(3,-10)
>RUN
   21-10
```

The I in the format specification is printed as "I"; it may be used to indicate that the imaginary part of the number is being printed. The second example prints the real part of the first number, and the imaginary part of the second.

The C specification is a second method of formatting complex numbers for output; this results in formatted output in a form similar to the standard PRINT format (i.e., parentheses enclosing two numbers separated by a comma). The C specification is the letter C followed by two noncomplex numeric fields (integer, fixed, or floating) which are separated by a comma and enclosed in parentheses. The first field defines the format of the real part of the number and the second defines the format of the imaginary part. Both fields must be present. If a noncomplex number is printed using this field, the imaginary part is assumed to be zero.

Examples

```
  10 PRINT USING 20;(3,2),(0,1),(.003,-4),(3,-4),21
  20 IMAGE C(3D,SDD),C(3D.D,M.DE),C(DD,MD.D),C(D.E,MDI),C(DD,SXXDD)
>RUN
(  3, +2)(    .0,  .1E+01)( 0,-4.0)(3.E+00,-4I)(21,    +0)
```

In these examples, the same expressions printed in the first example are printed with the C specification. With this specification, the imaginary part may have an M or an S.

COMPRESSED FORMAT SPECIFICATION. The compressed specification consists of one K and any editing symbols. It prints a number using only as many characters as are required. The resulting format is identical to that used in the simple PRINT statement (Section II), except that no blanks appear and trailing zeroes and decimal points are deleted. This specification is useful when numeric data is to appear within text.

Examples

```
  10 PRINT USING 20;25
  20 IMAGE "I HAVE",XKX,"BANANAS"
>RUN
I HAVE 25 BANANAS
```

Separators

BASIC/3000 format strings have two separators: the comma (,) and the slash (/). A separator is required between two adjoining specifications. The slash, in addition to separating specifications, also generates a carriage return and linefeed. Two separators can be adjacent. The meaning of adjacent commas is equivalent to a single comma.

Examples

```
  10 PRINT USING 20;25,10,-5,(1,250),2575.5,2.5
  20 IMAGE DDX.E/4D,/,SDDX,D+3D.E//"HIWAY"/4DX,,SD.E
  30 PRINT USING "5A";"ABCDE"
>RUN
25  .E+00
  10
 -5  1+250.E+00

HIWAY
2575  +3.E+00
ABCDE
```

Each slash in the format string causes a carriage return and linefeed.

Grouping

BASIC/3000 format strings allow one or more field specifications to be enclosed by parentheses and preceded by a replicator to indicate the repetition of the entire enclosed set of specifications. This newly grouped field must be separated from other specifications of the format string by a separator. Grouped specifications can be nested indefinitely.

Examples

```
   5 LET A$="A",B$="B",C$="C"
  10 PRINT USING 20;3000,300,A$,400,B$,500,C$,1.5,2.4,3.3,6.6
  20 IMAGE 4D4X,3(3D.D/5X,3A)/2(2(D.DX)/)
>RUN
3000     300.0
     A   400.0
     B   500.0
     C
1.5 2.4
3.3 6.6
```

The IMAGE statement in the example above is exactly equivalent to the following IMAGE statement:

```
  5 LET A$="A",B$="B",C$="C"
 10 PRINT USING 20;3000,300,A$,400,B$,500,C$,1.5,2.4,3.3,6.6
 20 IMAGE 4D4X,3D.D/5X,3A,3D.D/5X,3A,3D.D/5X,3A/D.DX,D.DX/D.DX,D.DX/
>RUN
3000     300.0
    A   400.0
    B   500.0
    C
1.5 2.4
3.3 6.6
```

Carriage Control Characters

The carriage control characters are

- \+ suppress linefeed
- \- suppress carriage return
- # suppress linefeed and carriage return

These characters specify action to be taken *following* execution of the PRINT USING statement. They specify whether a carriage return and linefeed are generated after the last item is printed. If no carriage control characters are present, a carriage return and linefeed will be provided automatically following each line. This is the only automatic line control in PRINT USING; within a format specification, the user is responsible for supplying carriage returns and linefeeds with slashes.

Carriage control characters are executed after all other formatted output, but they must be specified *first* in the format string.

Example

```
  10 PRINT USING "#,S3D.2D,3XS6D";125,625
  20 PRINT USING "X3D.DEX,10A";125,"E STRING"
 >RUN
 +125.00      +625 125.0E+00 E STRING
```

The carriage control character "#" in line 10 suppresses the automatic carriage return and linefeed following that PRINT USING statement, and causes the next PRINT USING statement to print on the same line.

SECTION X
Segmentation

Because the maximum size of a BASIC/3000 program is necessarily limited by memory resources, BASIC/3000 provides language facilities for segmenting programs into units that can call each other. Each unit or subprogram must be saved in the user's library; from there it may be called by the currently executing program into the user's work area.

Two statements are used for interprogram transfer: INVOKE and CHAIN; and one statement, COM, allows variables to be used in common by several programs.

CHAIN Statement

The CHAIN statement terminates the current program and begins execution of another program, optionally starting at a specified statement number.

Form

The forms of CHAIN are:

> *CHAIN string expression*
>
> *CHAIN string expression,integer expression*

The *string expression*, when evaluated, is the name of a BASIC/3000 program that is in the user's library. This may be a fully qualified file name (see Section VIII, Files). When evaluated, the optional *integer expression* is a label in the called program. If present, execution begins at the first executable statement at or after the label; the exact label need not be present in the called program. If omitted, execution begins at the first executable statement in the called program.

Explanation

CHAIN calls the program identified by the string expression, and it replaces the current program. When the program called by CHAIN finishes execution, it terminates and does not automatically return to the calling program. The called program may call another program, including the original calling program, with another CHAIN statement or an INVOKE statement.

Only variables declared in a COM statement are saved during a CHAIN operation. All variables and arrays of the current program that were not declared in COM are lost when the new program begins execution, and all files opened in the current program are closed.

Examples

```
MAIN
  10 REM..PROGRAM MAIN
  20 LET X=200,A=X**3
  30 PRINT "A=";A
  40 PRINT "LEAVE MAIN AND ENTER SUBA AT LINE 30"
  50 CHAIN "SUBA",30
  60 REM..THIS STATEMENT IS NEVER EXECUTED


SUBA
  10 REM..PROGRAM SUBA
  20 PRINT "THIS STATEMENT IS NOT EXECUTED"
  30 PRINT "ENTER SUBA - LINE 30"
  40 LET B=125,C=B**2
  50 PRINT "C=";C
  60 PRINT "END OF SUBA - TERMINATE HERE"
  70 END
>RUN MAIN
MAIN
A= 8.00000E+06
LEAVE MAIN AND ENTER SUBA AT LINE 30
ENTER SUBA - LINE 30
C= 15625
END OF SUBA - TERMINATE HERE
```

The main program, MAIN, calls program SUBA with a CHAIN command in line 50. Execution of SUBA begins in line 30, and execution terminates with the last line of SUBA. None of the variable values from MAIN are saved following execution of CHAIN.

INVOKE Statement

The INVOKE statement is similar to the CHAIN statement, except that the calling program is suspended rather than terminated and resumes execution when the program called by INVOKE terminates. The called program can be explicitly terminated with an END statement, otherwise it is implicitly terminated by the end of the program. In both cases, control returns to the suspended program that performed the INVOKE. If the called program is terminated by a STOP statement or a terminal error, the entire run is terminated including any suspended programs.

Form

The forms of INVOKE are:

INVOKE string expression

INVOKE string expression,integer expression

The *string expression* evaluates to the name of a BASIC/3000 program in the user's library. It may be a fully qualified file name. The *integer expression* evaluates to the label of a starting statement.

Explanation

The statements and variables of the current program are saved in a temporary file created by BASIC/3000 so that execution of the program can be continued at a later time. Variables declared in COM statements are passed to the program called by INVOKE. Files opened in the current program are not closed by INVOKE.

Execution of the new program begins at the first executable statement in the program or, if specified, at or after the label. The exact label need not be present in the called program.

When an implicit or explicit END statement in the called program is executed, control returns to the suspended program at the point immediately following the INVOKE statement. A STOP statement in the called program will, however, terminate the entire run including any suspended programs.

INVOKE operations can be nested; that is, a program that has been called by INVOKE can itself invoke another program including a recursive INVOKE of itself.

Examples

```
MAINX
  10 REM..PROGRAM MAINX
  20 LET A=25
  30 INVOKE "SUBY"
  40 PRINT "CONTROL RETURNS TO MAINX"
  50 PRINT "C=A*10=";A*10
SUBY
  10 REM..PROGRAM SUBY
  20 PRINT "EXECUTION OF SUBY BEGINS"
  30 PRINT "B=";8**-3
  40 PRINT "END OF SUBY - RETURN TO MAINX"
  50 END
>RUN MAINX
MAINX
EXECUTION OF SUBY BEGINS
B= 1.95313E-03
END OF SUBY - RETURN TO MAINX
CONTROL RETURNS TO MAINX
C=A*10= 250
```

MAINX uses an INVOKE command in line 30 to call for execution of SUBY. Execution begins with the first executable statement in SUBY. When SUBY terminates, control returns to line 40 of MAINX. The value of the variable A has been saved during execution of SUBY.

EXAMPLE USING CHAIN AND INVOKE

The example below has four programs: A, B, C, and D. All four programs have been saved in the user's library. The command RUN A brings A into the user's work area as the current program. With each successive CHAIN or INVOKE, a new program replaces the previous program in the work area; in this case A is the final as well as the first program in the work area.

When D terminates it returns to C, and when C terminates it returns to A because C was chained to by B which was invoked by A.

```
A
  10 REM..PROGRAM A
  20 PRINT "ENTER PROGRAM A"
  30 INVOKE "B"
  40 PRINT "BACK IN PROGRAM A - TERMINATE"



B
  10 REM..PROGRAM B
  20 PRINT "ENTER PROGRAM B"
  30 CHAIN "C",30
  40 REM..CONTROL SHOULD NEVER RETURN TO THIS POINT
  41 REM..DUE TO CHAIN IN LINE 30
  50 PRINT "NOT TO BE EXECUTED - B"



C
  10 REM..PROGRAM C - EXECUTION STARTS IN LINE 30
  20 PRINT "NOT TO BE EXECUTED - C"
  30 PRINT "ENTER PROGRAM C - LINE 30"
  40 INVOKE "D",25
  50 PRINT "BACK IN C - RETURN TO A"
  60 REM..END IN LINE 70 WILL CAUSE RETURN TO PROGRAM A
  61 REM..BECAUSE PROGRAM B WAS INVOKED BY PROGRAM A
  62 REM..NOTE THAT THE "END" IS UNNECESSARY
  70 END



D
   5 REM..PROGRAM D - EXECUTION STARTS IN LINE 25
  15 PRINT "NOT TO BE EXECUTED - D"
  25 PRINT "ENTER PROGRAM D - LINE 25"
  35 PRINT "RETURN TO PROGRAM C"
  45 REM..IMPLICIT END WILL CAUSE RETURN TO PROGRAM C
  47 REM  NOTE THAT EXPLICIT "END" WOULD DO THE SAME
>RUN A
A
ENTER PROGRAM A
ENTER PROGRAM B
ENTER PROGRAM C - LINE 30
ENTER PROGRAM D - LINE 25
RETURN TO PROGRAM C
BACK IN C - RETURN TO A
BACK IN PROGRAM A - TERMINATE
```

Files and Segmentation

Within a program, a file is referenced by its file number as determined by its position in the FILES statement (see Section VIII). When programs call one another with CHAIN or INVOKE, it is possible to reference files that were declared with FILES statements in other programs.

BASIC/3000 maintains an internal file numbering scheme that assigns an internal file number to every file declared in a FILES statement. For main programs, these numbers are the same as the file numbers to which the user refers. Files declared in a program called by INVOKE are assigned internal file numbers beginning with a value one greater than the last internal file number. Files declared in a program called by CHAIN are assigned internal file numbers beginning with the same number as the files in the calling program. Whether the program is a main program, is called by INVOKE, or is called by CHAIN, the local file numbers used to refer to files within each program will begin with 1.

When a program calls another program with INVOKE, all the files in the calling program remain open; when a program calls with CHAIN, the files in the calling program are closed.

To illustrate:

```
M1
  10 REM  MAIN PROGRAM M1
  20 FILES A,B,C
  30 INVOKE "M2"

M2
  10 REM  SUBPROGRAM M2
  20 FILES D,E,F
  30 CHAIN "M3"

M3
  10 REM  SUBPROGRAM M3
  20 FILES G,H
```

The internal and local file numbers for this group of programs is:

Internal File Numbers	Local File Numbers
	M1
A = 1	A = 1
B = 2	B = 2
C = 3	C = 3
	M2
D = 4	D = 1
E = 5	E = 2
F = 6	F = 3
	M3
G = 4	G = 1
H = 5	H = 2
(6 is unassigned)	

By using the *#integer* file designator in a FILES statement, a program may reference files declared in another program that invoked it. The value of the integer is the internal file number of a previously declared file; the position of the designator in the FILES statement is used to assign the local file number.

Examples

```
M1
  10 FILES A,B,C
  20 PRINT #1,1;"FILE A"
  30 PRINT #2,1;"FILE B"
  40 PRINT #3,1;"FILE C"
  50 INVOKE "M2"

M2
  10 FILES D,E,F,#3
  20 DIM A$[6]
  30 PRINT #1,1;"FILE D"
  40 PRINT #2,1;"FILE E"
  50 PRINT #3,1;"FILE F"
  60 READ #4,1;A$
  70 PRINT A$
  80 CHAIN "M3"

M3
  10 FILES G,H,#1,#2
  20 DIM A$[6],B$[6]
  30 PRINT #1,1;"FILE G"
  40 PRINT #2,1;"FILE H"
  50 READ #3,1;A$
  60 READ #4,1;B$
  70 PRINT A$,B$

>RUN M1
M1
FILE C
FILE A          FILE B
```

Within M2, reference to local file #4 is the same as a reference to internal file number 3 (file C). However, the statement FILES D,E,F,#3 is not the same as the statement FILES D,E,F,C. This latter statement, if specified, would have treated file C as a logically different file from file C in M1; the file would have been reopened with new buffers and access restrictions. By using #3, there is only one logical file C and any accessing affects that file, and the file is not reopened. In the same way, the statement FILES G,H,#1,#2 in M3 differs from a possible statement FILES G,H,A,B.

COM Statement

The COM statement is used to pass data values between segmented programs. Variables specified in a COM statement are placed in a common area so that values assigned to these variables in one program will be retained when transferring to another program with CHAIN or INVOKE. This area is known as a COM block. There may be more than one COM block, and it may or may not have an identifying label.

COM statements must precede all DIM, Type, or DEF statements in a program. All typing and dimensioning of variables is done within the COM statement, and any variables that appear in a COM statement must not also appear in a type or DIM statement in the same program.

BASIC/3000 permits ten COM blocks for each run, one unlabeled and nine labeled uniquely with the digits 1 through 9.

Form

The forms of the COM statement are:

COM com item list

COM(nonzero digit)com item list

The *com item list* consists of a list of variable declarations. Simple variables are indicated by the variable name; arrays are indicated by the array name and a bounds indicator. The bounds indicator is equivalent to the dimension specification used in a DIM statement if the block is being created; it indicates only the number of dimensions if the block has been created and is currently active. The number of dimensions are specified with (*) or (*,*).

The type of items in the *com item list* is assumed to be real unless the variable name contains a $ to indicate a string variable, or the variable is preceded by a type specifier (INTEGER, LONG, COMPLEX, or REAL). The type specifier assigns that type to all succeeding variables until the end of the list or the next type specifier or string variable.

The optional block indicator is specified as a *nonzero digit* between 1 and 9. This assigns a label to a block being created, or specifies an existing labeled block.

Explanation

Programs execute in BASIC/3000 on a dynamic level basis. The original program run with the RUN command starts at level 1. When a CHAIN is executed, the new program executes also at level 1 since the old program terminates. However, when an INVOKE is executed, the new program executes at level n+1 where n is the level of the invoking program. When control returns to the invoking program, the level is reduced to n.

COM blocks become active whenever a program declares a COM block that is not currently existing or active. COM blocks created at level n are active until the dynamic level drops to n-1 or the run terminates.

If a COM statement references an inactive COM block, then numeric bounds for the arrays and strings are specified as in a DIM statement. If, however, a COM statement references an active block, then it need only indicate the number of dimensions. For a one-dimensional array or a string variable, the variable name is followed by (*); for a two-dimensional array or a string array, the variable name is followed by (*,*). The (*) can be omitted for simple string variables and will be assumed. Numeric bounds may be specified for an active COM block, but in this case they must be identical to the original COM statement bounds.

CORRESPONDENCE RULES. All variables in COM statements that reference an active COM block must match exactly in type, number of dimensions, and order within the COM statement. Also, if dimension size is specified instead of the *, these must match exactly. The names of corresponding variables need not be the same since equivalence is based on the order of appearance of the variables in the COM statements. A COM statement defining an active block must contain the same number of elements as the COM statement that created the block. More than one COM statement in a program can define the same block if the statements are contiguous.

The order of elements in a COM statement, or in more than one contiguous COM statement, implies the order of the variables in the COM block.

The rules governing correspondence between COM statements are checked when a CHAIN or INVOKE statement is executed. If any of these conditions is not met, a terminal error occurs.

Examples

Program A1 chains to program B1 which in turn chains to program C1:

```
A1
  10 COM B,B$[5],C,INTEGER D,E[5],LONG F[5,2]
  20 COM(5) Q,COMPLEX A,A$[2,2],P
  30 LET B=10,B$="ABCDE",C=20.5,D=1
  40 MAT READ E,F
  50 DATA 1,2,3,4,5,1.01L11,5.2L10,1.57L11,1.76L11,1.76L-9,1.53L11
  60 DATA 1.575L-4,1.57L6,1.57L-5,1.752L10
  70 LET Q=1954.75,A=(12.3,4),P=3.14
  80 MAT READ A$
  90 DATA "AB","CD"
 100 CHAIN "B1"
```

```
B1
  10 COM T,C$[*],Q,INTEGER F,D[*],LONG F1[*,*]
  20 PRINT "START B1"
  30 PRINT T,C$,Q,F
  40 MAT PRINT D;F1
  50 CHAIN "C1"

C1
  10 COM(5) A,COMPLEX B,C$[*,*],D
  20 PRINT "START C1"
  30 PRINT A,B,D
  40 MAT PRINT C$

>RUN A1
A1
START B1
 10              ABCDE               20.5            1
 1       2       3       4       5

 1.01000000000000L+11           5.20000000000000L+10

 1.57000000000000L+11           1.76000000000000L+11

 1.76000000000000L-09           1.53000000000000L+11

 1.57500000000000L-04           1.57000000000000L+06

 1.57000000000000L-05           1.75200000000000L+10

START C1
 1954.75         ( 1.23000E+01, 4.00000E+00)     3.14
AB

CD
```

In this example, program A1 creates the unlabeled COM block and also COM block 5. Program B1 references the unlabeled COM block and prints the data assigned to that block in A1. Program C1 references COM block 5 and prints the data assigned to that block in A1. In both B1 and C1, the number of dimensions of subscripted COM variables is indicated by (*) or (*,*).

Assume that program D1 chains to program E1:

```
D1
  10 COM(9) A[3,5],INTEGER D[6,6]
  20 MAT A=ZER
  30 MAT D=IDN
  40 CHAIN "E1"


E1
  10 COM(9) B[3,5],INTEGER P[6,6]
  20 MAT PRINT B;LIN(1),P;
>RUN D1
D1
 0      0      0      0      0

 0      0      0      0      0

 0      0      0      0      0


 1      0      0      0      0      0

 0      1      0      0      0      0

 0      0      1      0      0      0

 0      0      0      1      0      0

 0      0      0      0      1      0

 0      0      0      0      0      1
```

In this example, program D1 creates COM block 9, and program E1 references the data in block 9. Actual numeric bounds are specified in the COM statement in E1. This is legal only if the bounds are identical to the original bounds specification.

In the following example, execution of the four programs starts in W:

W

```
 10 REM PROGRAM W
 20 COM(3) B[5]
 30 FOR I=1 TO 5
 40   B[I]=I
 50 NEXT I
 60 INVOKE "X"
 70 PRINT LIN(5);"BACK IN W - B="
 80 MAT PRINT B
 90 CHAIN "Z"
```

X

```
 10 REM.. PROGRAM X
 20 COM(3) A[*]
 30 COM(4) LONG C[3,2]
 40 PRINT LIN(5);"IN X -- A=";LIN(1)
 50 MAT PRINT A
 60 FOR I=1 TO 5
 70   A[I]=10*I
 80 NEXT I
 90 FOR I=1 TO 3
100   FOR J=1 TO 2
110     C[I,J]=10*I+J
120   NEXT J
130 NEXT I
140 INVOKE "Y"
150 PRINT LIN(5);"BACK IN X -- C=";LIN(1)
160 MAT PRINT C
170 PRINT LIN(1);"RETURN TO W"
```

Y

```
 10 REM.. PROGRAM Y
 20 COM(4) LONG F[*,*]
 30 PRINT LIN(5);"IN Y -- F=";LIN(1)
 40 MAT PRINT F
 50 FOR I=1 TO 3
 60   FOR J=1 TO 2
 70     F[I,J]=100*I+10*J
 80   NEXT J
 90 NEXT I
100 PRINT LIN(1);"RETURN TO X"
```

Z

```
 10 REM.. PROGRAM Z
 20 COM(4) INTEGER I,J,K
 30 PRINT LIN(5);"IN Z-- COM(4)=";LIN(1)
 40 IF UND(I) THEN PRINT "  I UNDEFINED"
 50 ELSE PRINT "  I=";I
 60 IF UND(J) THEN PRINT "  J UNDEFINED"
 70 ELSE PRINT "  J=";J
 80 IF UND(K) THEN PRINT "  K UNDEFINED"
 90 ELSE PRINT "  K=";K
100 PRINT "TERMINATE IN Z"
```

```
>RUN W
W


IN X -- A=

 1                  2                   3                   4                   5


IN Y -- F=

 1.10000000000000L+01                   1.20000000000000L+01

 2.10000000000000L+01                   2.20000000000000L+01

 3.10000000000000L+01                   3.20000000000000L+01

RETURN TO X


BACK IN X -- C=

 1.10000000000000L+02                   1.20000000000000L+02

 2.10000000000000L+02                   2.20000000000000L+02

 3.10000000000000L+02                   3.20000000000000L+02

RETURN TO W


BACK IN W - B=
 10                 20                  30                  40                  50



IN Z-- COM(4)=

  I UNDEFINED
  J UNDEFINED
  K UNDEFINED
TERMINATE IN Z
```

Note that Z can create a new COM block 4 since execution has dropped below the dynamic level 2 at which X created the first COM block 4.

The following three examples illustrate illegal COM usage:

Assume that program F1 invokes program G1:

```
F1
  10 COM(1) A[10,10],LONG B[10,10]
  20 MAT A=CON
  30 MAT B=ZER
  40 INVOKE "G1"


G1
  10 COM(1) LONG D[*,*],REAL E[*,*]
  20 MAT PRINT D,E
>RUN F1
F1
COM NOT SAME AS FIRST OCCURENCE IN G1
```

This is illegal because the corresponding variables in the two COM statements do not agree in type and order. If the type references were interchanged in program G1, this would be a legal example.

Assume that program HH chains to program JJ:

```
HH
  10 COM(7) A$[3,5],B[2,5]
  20 MAT B=IDN
  30 MAT READ A$
  40 DATA "ABCDE","FGHIJ","LMNOP"
  50 CHAIN "JJ"


JJ
  10 COM(7) Q$[*],D[*,*]
  20 PRINT Q$
  30 MAT PRINT D
>RUN HH
HH
COM NOT SAME AS FIRST OCCURENCE IN JJ
```

This is illegal because Q$ is a simple string variable, whereas A$ is a string array.

Assume that program K1 invokes program P1:

```
K1
  10 COM(8) X[*,*]
  20 INVOKE "P1"
  30 MAT PRINT X


P1
  10 COM(8) Y[10,3]
  20 MAT Y=ZER
>RUN K1
MISSING SUBSCRIPT SPECIFICATION IN FIRST OCCURENCE OF COM IN K1
```

This is illegal because the creator of a COM block must specify the actual physical size of arrays and strings. If, however, P1 invoked K1 this would be a legal example.

SECTION XI

Communication with Non-BASIC Programs

A BASIC/3000 user can access an SPL/3000 procedure or a FORTRAN/3000 subprogram from a BASIC program with the CALL statement. As mentioned in Section I, he can enter MPE/3000 with the SYSTEM command or by pressing the break key. Another method for using the facilities of the MPE/3000 Operating System is with the SYSTEM statement. The SYSTEM statement enters an MPE system command dynamically into a BASIC/3000 program.

CALL Statement

The CALL statement calls for execution of an SPL/3000 procedure or a FORTRAN/3000 subprogram that exists in an MPE/3000 segmented procedure library (SL). Parameters may be passed to the called procedure or subprogram. The search for an external parameter begins in the group library. (See MPE/3000 Operating System, Reference Manual.

Care should be taken when using SPL or FORTRAN since these procedures are not controlled by the BASIC/3000 Interpreter and errors in them can propagate into the Interpreter causing indeterminate results.

Form

The forms of CALL are

CALL procedure name

CALL procedure name(actual parameter list)

** procedure name*

** procedure name(actual parameter list)*

The *procedure name* identifies the procedure or subprogram being called. The optional actual parameter list may contain numeric or string expressions, variables and array names followed by bound indicators (*) or (*,*). Anything that can be passed to a function (see Section VI) can be passed in a CALL statement.

The asterisk may be used interchangeably with CALL.

Explanation

CALL transfers control to the beginning of the specified procedure. If parameters are specified, they are passed by reference as address pointers rather than by value. If the parameter is an expression, a temporary variable is created by the Interpreter to contain the value of the expression.

In addition to the parameter addresses, BASIC/3000 passes the number of parameters and also code words describing the parameters (one code word for every three parameters). The code words may be used by the procedure to insure that the calling sequence is correct.

The parameter information used by the program or procedure being called is set up by the BASIC/3000 Interpreter when the CALL statement is executed. Depending on the called program, values may be returned to the BASIC program through the specified parameters. The formats of the parameter address table and of the parameter values is contained in Appendix F, Parameter Format. This information is useful primarily if the BASIC user is writing or modifying the called program. Otherwise, he only needs to know the type and the order of the parameters used in the called program in order to specify them in the CALL statement.

Examples

The first example calls the FORTRAN segment BOOLEAN containing three subroutines that perform Boolean operations. The first subroutine LAND performs a logical AND, the second LOR performs a logical OR, and the third LNOT performs a logical NOT. Each subroutine is called with a different CALL statement and the result is printed upon return to BASIC.

The second example calls the SPL procedure WHOM from a BASIC program. WHOM returns the user's name, group, accout, and home group from the identification codes entered by the user when he logs on. This information is derived by WHOM using the MPE/3000 system intrinsic WHO. WHO is an option variable procedure and, as such, cannot be called directly from BASIC/3000. This example illustrates, among other things, the interface with an intrinsic that cannot be referenced directly.

Each subroutine and procedure is listed following the BASIC program that calls it, and this list is followed by an SL (segmented procedure library) list requested after the segment is added to the SL.

1. Calling FORTRAN Subroutines

```
  10 INTEGER X,Y,Z
  20 LET X=1,Y=0
  30 *LOR(X,Y,Z)
  40 PRINT Z
>RUN
 1

  10 INTEGER X,Y,Z
  20 LET X=1,Y=0
  30 CALL LAND(X,Y,Z)
  40 PRINT Z
>RUN
 0

  10 INTEGER X,Y,Z
  20 LET X=1,Y=0
  30 CALL LNOT(Y,Z)
  40 PRINT Z
>RUN
-1
```

The FORTRAN segment containing subroutines LAND, LOR, and LNOT is listed below. The comment lines describe the parameter restrictions. Note that all three parameters should be integers.

The user should refer to the FORTRAN/3000 Reference Manual for instructions on writing a FORTRAN subprogram.

```
$CONTROL  USLINIT, NOLIST, SEGMENT=BOOLEAN
      SUBROUTINE  LAND(A,B,RESULT)
C         PERFORMS BOOLEAN-AND ON "A" AND "B", RETURNING RESULT IN
C         "RESULT".  "A" AND "B" MUST BE TYPE INTEGER SIMP. VAR. OR
C         EXPRESSION; "RESULT" MUST BE INTEGER SIMP. VAR.(SUBSCRIPTED
C         OR NOT).
        LOGICAL A,B,RESULT
        RESULT = A .AND. B
        RETURN
       END
$CONTROL  SEGMENT=BOOLEAN
      SUBROUTINE  LOR(A,B,RESULT)
C         PERFORMS BOOLEAN-OR ON "A" AND "B", WITH RESULT RETURNED IN
C         "RESULT".  PARAMETERS ARE SAME AS FOR "LAND".
        LOGICAL A,B,RESULT
        RESULT = A .OR. B
        RETURN
       END
$CONTROL  SEGMENT=BOOLEAN
      SUBROUTINE  LNOT(A,RESULT)
C         PERFORMS BOOLEAN-NOT ON "A", RETURNING RESULT IN "RESULT".
C         "A" AND "RESULT" ARE SAME AS FOR "LAND".
        LOGICAL A,RESULT
        RESULT = .NOT. A
        RETURN
       END
```

Explanation

Any string variable, simple or subscripted, can be qualified by a substring designator, which is used to select a part of the string to be extracted.

If the substring is specified by a single expression, the substring equals the rest of the string taken from the position indicated by the expression.

If two expressions are separated by a comma, the substring consists of the characters from the position specified by the first expression to the position specified by the second expression. (Note: the second expression can be one less than the first; this specifies the null string).

If two expressions are separated by a semicolon, the substring consists of the characters in the string variable starting at the position indicated by the first expression and taking the number of consecutive characters specified by the second expression. In this case, the expression may evaluate to zero, giving a null string.

If A$ is a simple variable:

A$(3,5)	is the 3rd through the 5th character of the string.
A$(3;3)	is also the 3rd through the 5th character of the string since, in this case, the second expression follows a semicolon indicating the number of characters rather than the last character.
A$(3;0)	is the null string.
A$(3,2)	is also the null string.
A$	every character in the string is selected.

If B$ is an array variable:

B$(3)	is the entire 3rd string in the string array.
B$(2,3,5)	is the 3rd through 5th characters in the second string of the string array.
B$(2,3;3)	is also the 3rd through 5th characters in the second string; the substring starts at the 3rd character and contains 3 characters.

A string array variable must always be subscripted except in MAT statements (see String MAT Operations, this section).

The subscript and substring designator expressions may be any integer expressions. Suppose the variables I and J are used, with I equal to 5 and J equal to 10:

C$(I)	is the 5th character to the end of the string if C$ is a simple string variable; it is the entire 5th string or element if C$ is a string array variable.
C$(I,J)	is the 5th through 10th character if C$ is a simple string variable; it is the 10th character to the end of the string of the 5th string if C$ is a string array variable.

If a substring extends beyond the logical length of a string variable, it is filled out to the specified size with blanks.

Examples

```
  10 DIM A$[10]
  20 A$="ABCDEFGHIJ"
  30 PRINT "STRING A$=";A$
  40 PRINT "SUBSTRING A$(5)=";A$[5]
  50 PRINT "SUBSTRING A$(2;5)=";A$[2;5]
  60 PRINT "SUBSTRING A$(2,5)=";A$[2,5]
  70 PRINT A$[2,1];"=NULL STRING"
>RUN
STRING A$=ABCDEFGHIJ
SUBSTRING A$(5)=EFGHIJ
SUBSTRING A$(2;5)=BCDEF
SUBSTRING A$(2,5)=BCDE
=NULL STRING
```

In the example above, note the difference between A$(2;5) and A$(2,5). In the example below, each array element is a two character string.

```
  10 DIM B$[10,2]
  20 REM   B$ IS A STRING ARRAY
  30 MAT READ B$
  40 DATA "A1","B2","C3","D4","E5","F6","G7","H8","I9","J0"
  50 PRINT "ARRAY ELEMENT B$(2)=";B$[2]
  60 PRINT "ARRAY ELEMENT B$(2,2)=";B$[2,2]
  70 PRINT "ARRAY ELEMENT B$(2,1;1)=";B$[2,1;1]
>RUN
ARRAY ELEMENT B$(2)=B2
ARRAY ELEMENT B$(2,2)=2
ARRAY ELEMENT B$(2,1;1)=B
```

String Expressions

String expressions consist of one or more source strings (literal strings, string variables, string valued functions) combined from left to right with the concatenate operator (+) to form a single new string value. String expressions can be assigned to string variables or compared with other string expressions to form a numeric expression.

Form

The form is a list of source strings separated by "+"

string

string + *string*. . .

Each source *string* can be either a literal string, a string variable, or a string function.

Explanation

A source string is any entity from which a string value is extracted. The value of the source string is as defined under "String Literals," "String Variables," and "String Functions." An example of a literal string is "BASIC" or '10; of a string variable is A$, C5$(2), B$(2,3), B$(2;2), or A1$(5,3,10); of a string function is CHR$(208).

The "+" character, when used between two source strings, is the concatenate operator. The concatenation of two strings produces a temporary string whose characters are those of the first string immediately followed by those of the second. This temporary string can be used in further concatenation operations, in string comparisons, or it can be assigned to a string variable.

The maximum length of any temporary string is 255 characters. The original operands are unaffected by concatenation.

Legal string expressions:

A$ + B$(2) + '93 + '10'23"ABCD" + C5$(4,3;5)

"BASIC" + C5$(2)

"BASIC"

C5$(2)

Example

```
  10 DIM A$[5],B$[10,10]
  20 LET A$="CON",B$[2]="CATENATION"
  30 PRINT A$+B$[2,1;7]+"E"
>RUN
CONCATENATE
```

String Assignment

The assignment operator (=) can be used to assign a string value (defined by a string expression) to one or more string variables (or substrings of string variables). Several different assignments can appear in one LET statement.

Form

Some forms of LET are

LET variable=expression

LET variable=variable=. . .=variable=expression

LET variable=expression,variable=variable=expression,. . .

The word LET is entirely optional and can be left off. The *variable* is an entire string variable (simple or subscripted) or part of a string variable (indicated by a substring designator) into which a string value is to be copied. If several variables are separated by equals (=) each is assigned a copy of the value. Numeric assignments as described in Section II can be mixed with string assignments in the same LET statement.

Explanation

The execution of a LET statement proceeds as follows. The subscripts of variables to be assigned values are evaluated from left to right. The expression is then evaluated and assigned to the variables. The same expression is assigned to each variable from left to right. The manner in which each assignment occurs depends upon the number of substring subscripts specified for the destination variable.

If there is no substring designator, the entire variable is replaced by the string value. If the new value will fit entirely into the variable, the logical length of the variable is set to the length of the new value. If the variable is too small, the value is truncated on the right and the logical length of the string is made equal to the physical length.

If there is one substring subscript, this specifies the starting position for the assignment. The entire string value is copied into the variable starting with the indicated position and continuing to the physical end of the variable or the end of the string value, whichever comes first. The part of the variable preceding the subscript is unchanged. The starting subscript must be no more than one greater than the current *logical* length of the variable (i.e., there can be no undefined character positions in the middle of a string variable). If the variable is too small, the value is truncated on the right.

If two substring subscripts are specified, they define a field within the variable into which the string value is stored. If necessary, the value will be truncated on the right or padded out with blanks to fit exactly in the substring specified. The substring for the destination must not extend beyond the physical length of the string variable and all previously mentioned rules must be followed also. The

new logical length of the variable is the *larger* of the old logical length or the last position of the substring. Any characters from the old value to the left or right of the substring are unchanged.

Example

```
 10 DIM A$[10]
 20 LET A$="1234567890"
 30 PRINT A$
 40 LET A$[5]="ABCDEF"
 50 PRINT A$
 60 LET A$[7;3]="1234"
 70 PRINT A$
 80 A$[6,8]="X"
 90 PRINT A$
100 A$=A$[1,4]+"567890"
110 PRINT A$
>RUN
1234567890
1234ABCDEF
1234AB123F
1234AX  3F
1234567890
```

Note that the literal "1234" in line 60 is truncated to fit in substring A$(7;3).

In line 80, substring A$(6,8) is blank filled since "X" is only one character. The final value of A$ is the same as its original value assigned in line 20.

The example below illustrates variations on assignments to substrings of array elements:

```
 10 DIM A$[3,5]
 20 A$[1]=A$[2]=A$[3]="ABCDE"
 30 LET A$[1,3]=A$[2]
 40 PRINT A$[1],A$[2],A$[3]
 50 LET A$[2,4,5]=A$[3]
 60 PRINT A$[1],A$[2],A$[3]
 70 LET A$[3,2;2]=A$[2]=A$[1,1,1]
 80 PRINT A$[1],A$[2],A$[3]
>RUN
ABABC          ABCDE          ABCDE
ABABC          ABCAB          ABCDE
ABABC          A              AA DE
```

String-Related Functions

There are a number of predefined functions in BASIC/3000 that accept string values as parameters and/or return a string value as their result. (User-defined string functions are described in Section VI.)

CHR$ Function

CHR$(integer expression)

where integer expression given a value in the range 0 to 255 inclusive. The value of CHR$ is the string character that corresponds to the value of the expression in the standard character set (see Appendix A). For example,

```
  10 PRINT CHR$(65)
>RUN
A
```

NUM Function

NUM(string expression)

NUM returns the numeric value of the first character of the string expression according to the standard character code in Appendix A. For example,

```
  10 PRINT NUM("A")
>RUN
 65
```

LEN Function

LEN returns the logical length of the string expression. For example,

```
  10 DIM A$[20]
  20 A$="ABCD"
  30 PRINT LEN(A$)
>RUN
 4
```

POS Function

POS(stringA,stringB)

where *stringA* and *stringB* are any string expressions; POS returns the smallest integer that represents the starting position of a substring in *stringA* that exactly equals *stringB*. If *stringB* is not a substring of *stringA*, then POS equals zero. For example.

```
  10 PRINT POS("12ABC34","ABC")
>RUN
 3
```

WRD Function

WRD(stringA,stringB)

where *stringA* and *stringB* are any string expressions. WRD returns the smallest integer that represents the starting position of a substring in *stringA* that exactly equals the value of *stringB* and is neither immediately preceded nor immediately followed by a letter. If there is no such substring, WRD equals zero. For example,

```
  10 PRINT  WRD("STRING A$ EQUALS X$","EQUALS")
  20 PRINT  WRD("12ABC34","ABC")
  30 PRINT  WRD("12ABC34","BC")
>RUN
 11
 3
 0
```

UPS$ Function

UPS$(string expression)

UPS$ returns a string value equivalent to *string* with all lower case letters upshifted.
UPS$("abcd%12") is "ABCD%12".

DEB$ Function

DEB$(string expression)

DEB$ returns a string equal to the specified string expression, but with all leading and trailing blanks removed. Embedded blanks remain. For example,

```
  10 PRINT DEB$("    A B     ")
>RUN
A B
```

ROW Function

ROW(string array)

where *string array* is any string array name without subscripts. ROW returns the number of elements in the array since a string array is always one-dimensional. For example,

```
  10 DIM B$[15,5]
  20 PRINT ROW(B$)
  30 REDIM B$[10]
  40 PRINT ROW(B$)
>RUN
 15
 10
```

COL Function

COL(string array)

where *string array* is any string array name. COL returns the number of columns in the array. Since string arrays are always one-dimensional, COL always returns a value of 1 for string arrays. For example,

```
  10 DIM B$[15,6]
  20 PRINT COL(B$)
>RUN
 1
```

DAT Function

DAT$(x,y)

where *x* and *y* are integer expressions which specify the first and last character positions, respectively, of a substring within a full string which defines the current date and time. The full date/time string is structured as follows:

Char 1-3:	Day of the week (SUN, MON, TUE, WED, THU, FRI, SAT)
4-5:	Comma and blank
6-8:	Month of the year (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC)
9:	Blank
10-11:	Day of the month (1 to 31)
12-13:	Comma and blank
14-17:	Year
18-19:	Comma and blank

20-21: Hour (1 to 12)
22: Colon
23-24: Minute (0 to 59)
25: Blank
26-27: AM or PM

For example, to print the entire date/time string:

```
  10 PRINT DAT$(1,27)
>RUN
WED, MAY 30, 1973,  2:45 PM
```

To print only the date:

```
  10 PRINT DAT$(6,17)
>RUN
MAY 30, 1973
```

To print only the time:

```
  10 PRINT DAT$(20,27)
>RUN
 2:46 PM
```

INPUTTING STRINGS

The INPUT statement can be used to assign string constants to string variables from the terminal.

The rules for entering strings from the terminal are

1. The apostrophe form of a string literal (e.g., '40) is not allowed.

2. All strings must have quote marks around them except the last one entered on each line.

3. If the last string on a line does not begin with a quote, it starts with the first non-blank character (i.e., it can have no leading blanks) and ends with the last non-blank character before the carriage return (i.e., it can have no trailing blanks).

The rules used to assign the value to the variable are those described under "String Assignment."

Examples

```
  10 DIM A$[16],B$[2,5],C$[40]
  20 INPUT A$,B$[1],B$[2],C$
  30 PRINT A$;B$[1];B$[2];C$
>RUN
?"THE VALUE OF B$=","1234 "," 2X5 ", X5=ABC
THE VALUE OF B$=1234  2X5 X5=ABC
```

Note that the last value input is not quoted. Usually only one string per line is input, in which case quotes are not required.

ENTERING STRINGS

The ENTER statement allows one string variable to be assigned a value from the terminal with control over the input operation. See Section II for how this control is exercised.

All characters typed in (including quotes) are assigned to the variable using the rules defined under String Assignment. ENTER does not prompt for input, and does not provide a linefeed after the user's carriage return.

Examples

```
   10 DIM A$[30],B$[10]
   20 PRINT "YOU HAVE 30 SECONDS TO ENTER 30 CHARACTERS"
   30 ENTER 30,T,A$
   40 PRINT '10,T,A$
   50 PRINT "YOU HAVE 15 SECONDS TO ENTER 10 CHARACTERS"
   60 ENTER 15,X,B$
   70 PRINT '10,X,B$
>RUN
YOU HAVE 30 SECONDS TO ENTER 30 CHARACTERS
LET A$ = "VALUE OF A$" , 012345
 18.59          LET A$ = "VALUE OF A$" , 01234
YOU HAVE 15 SECONDS TO ENTER 10 CHARACTERS
A$=LEN(B$)
 6.44           A$=LEN(B$)
```

A common use of ENTER is for testing:

```
   10 DIM A$[30],B$[10]
   20 PRINT "WHO WAS THE FIRST PRESIDENT OF THE U.S.?"
   30 ENTER 30,T,A$
   40 IF A$="GEORGE WASHINGTON" THEN GOTO 80
   50 PRINT '10"SORRY,TRY AGAIN"
   60 ENTER 20,T,A$
   70 IF A$<>"GEORGE WASHINGTON" THEN GOTO 140
   80 PRINT '10"CORRECT,YOU TOOK";T;"SECONDS TO ANSWER"
   90 PRINT "WHAT WAS HIS WIFE'S NAME?"
  100 ENTER 15,X,B$
  110 IF B$<>"MARTHA" THEN GOTO 160
  120 PRINT '10"CORRECT,YOU TOOK";X;"SECONDS TO ANSWER"
  130 END
  140 PRINT '10"SORRY,THE CORRECT ANSWER IS GEORGE WASHINGTON"
  150 END
  160 PRINT '10"SORRY, THE CORRECT ANSWER IS MARTHA"
>RUN
WHO WAS THE FIRST PRESIDENT OF THE U.S.?
GEORGE WASINGTON
SORRY,TRY AGAIN
GEORGE WASHINGTON
CORRECT,YOU TOOK 7.97       SECONDS TO ANSWER
WHAT WAS HIS WIFE'S NAME?
MARTHA
CORRECT,YOU TOOK 4.13       SECONDS TO ANSWER
```

PRINTING STRINGS

Any string expression can be output to the list device (e.g. the terminal) using the PRINT statement. The size of the output field is the number of printing characters in the string value. If the string expression is preceded by a comma, it is printed starting in the next division. Each print line is divided into 5 divisions, four with a width of 15 characters, one with a width of 12 (see Print Statement, Section II). If the string expression is preceded by a semicolon, it is printed immediately following the preceding output. The semicolon between items in the print list need not appear if the first item ends with a quoted string, and/or the second item begins with one; in this case a semicolon is inserted automatically.

Strings can be output to the terminal with special formats through the PRINT USING statement (see Section IX, Formatted Output). Strings can be output to files as described in Section VIII.

Examples

```
  10 DIM C$[10],N5$[3,5]
  20 LET C$="XK9-753-20",A=2.5,B=1E-19,N5$[1]="ABCDE"
  30 PRINT A,B,C$
  40 PRINT "BOB"+C$,N5$[1]
  50 PRINT C$+"BOB";N5$[1]
  60 PRINT '10'34"LINE"'34'10'13;"-1"
>RUN
 2.5            1.00000E-19    XK9-753-20
BOBXK9-753-20  ABCDE
XK9-753-20BOBABCDE

"LINE"
-1
```

In the first expression in line 60, the '10 (linefeed) causes a linefeed, the '13 (carriage return) causes a carriage return when the line is printed. The '34 (quote) causes a quote to be printed. The actual quote (") before and after the string LINE in the PRINT statement is not printed.

LINPUT Statement

The LINPUT statement accepts all the characters that a user types in at the terminal and assigns them as a string to a specified string variable.

Form

LINPUT string variable

where the *string variable* is the destination of the input. The variable may be simple or subscripted, or it may be a substring.

Explanation

All characters are accepted including quotes and blanks. Input is terminated by a carriage return. No prompt character is printed.

Examples

```
   10 DIM A$[20]
   20 PRINT "TYPE 20 CHARACTERS:"
   30 LINPUT A$
   40 PRINT A$
   50 PRINT "TYPE 5 CHARACTERS:"
   60 LINPUT A$[10;5]
   70 PRINT A$[10;5]
>RUN
TYPE 20 CHARACTERS:
"ANY CHARACTERS" O.K.
"ANY CHARACTERS" O.K
TYPE 5 CHARACTERS:
&"+ "
&"+ "
```

Because more than 20 characters (the size of A$) were input by the user, the final period in the first input line is truncated. In the second input, quotes are entered as part of the string.

String Array Operations

The String MAT Initialize statement sets every element of a string array to the null string. In addition, a new dimension can be specified for the array, as in REDIM. The form for string array initialization is

MAT string array = NUL$

MAT string array = NUL$ (integer expression)

Each element of the *string array* is set to the null string. If the optional *integer expression* is specified, the number of elements in the array is changed to the number of elements specified by the expression. The element size is not changed.

The String MAT Copy statement copies all of the elements of one string array into another. String elements are truncated if necessary. The form for string array copying is

MAT string arrayA = string arrayB

This causes *string arrayB* to be copied into *string arrayA*. *String arrayA* is redimensioned to the size of *string arrayB.*

String arrays can be read, input, and printed with the MAT READ, MAT INPUT, and MAT PRINT statements as described for numeric arrays in Section III.

String arrays can also be printed according to a specified format with the MAT PRINT USING statement (see Section IX).

Examples

```
10 DIM A$[10,5],B$[20,5]
20 MAT A$=NUL$
30 MAT B$=A$
```

Each element in string array A$ is set to the null value, then the first ten elements of string array B$ are set to null values. B$ is redimensioned to the size of A$.

In the example below, B$ is redimensioned and each element is set to the null string:

```
10 DIM B$[20,5]
20 MAT B$=NUL$(10)
```

Convert Statement

The CONVERT statement is used to convert a numeric value to a string of characters that represent the number, or vice versa. In the first case, the conversion is identical to that used when listing. The destination string variable should be long enough to contain the result, as in "String Assignment"; if not, the result is truncated. In the second case, the string expression must represent a valid numeric constant. An error in this case terminates the program, unless a label is specified in the CONVERT, in which case, control transfers to that label.

Form

CONVERT numeric expression TO string variable

CONVERT string expression TO numeric variable [,label]

Explanation

The string variable must be long enough to contain the converted numeric expression; if not, the converted expression is truncated. No blanks appear in the string. If the string expression in the second form does not represent a valid numeric constant, control transfers to the optional statement label. If the label is omitted and the string expression is not convertible, the program terminates.

Examples

```
   10 A=10,B=15
   20 DIM A1$[5]
   30 CONVERT A+B TO A1$
   40 PRINT A1$
   50 CONVERT A1$ TO X,100
   60 PRINT X
   70 END
  100 PRINT "ERROR IN CONVERSION FROM STRING"
>RUN
25
 25
```

Since A1$ contains the converted numeric constant, 25, there is no error in line 50. If A1$ contained a value that could not be converted, the message in line 100 would be printed. For instance, if line 50 is changed as follows:

```
>50 CONVERT "A1" TO X,100
>RUN
25
ERROR IN CONVERSION FROM STRING
```

SECTION VI
User-Defined Functions

A user-defined function is one that is defined within the user program and is called within that program in the same way that a built-in function is called. The name of a function that returns a numeric value consists of three letters, the first two of which are "FN" (e.g., FNA, FNB, etc.). String-valued functions have the same set of names available with the addition of a dollar sign ($) at the end (e.g., FNA$, FNB$,...). Thus, up to 26 numeric-valued functions and 26 string-valued functions are allowed in any one program.

A function is called within an expression by referring to its name and a list of parameter values enclosed in parentheses. The value returned by the function takes its place in the expression.

There are two levels of complexity in the definition of a BASIC/3000 function. At the simple level, a one-line function simply relates a function name and list of parameters to any expression which may use the parameters to calculate the result value. The multiline function is a more complex entity; it can consist of many statements and local variable declarations. It returns its result value with a RETURN statement.

For a discussion of BASIC/3000 built-in functions, see Functions in Section II. A complete list of the built-in functions available to the BASIC/3000 user is contained in Appendix E.

Multiline Function

A multiline function is written as several contiguous statements beginning with a DEF statement and ending with an FNEND statement. Execution of the function ends when a function RETURN statement is encountered; this sends the result value back to the place of call.

Form

A multiline function definition has three parts; the function head, the function body, and the function end.

The function head appears as

DEF function-name(formal parameter list)

DEF type function-name(formal parameter list)

DEF string-function-name(formal parameter list)

The first two forms return numeric values, the third returns a string value.

All parts of these function definitions are the same as described for one-line functions.

The function body consists of a sequence of statements, including at least one function RETURN statement:

RETURN numeric expression

RETURN string expression

The *expression* is numeric or string depending on whether the function is numeric or string. For numeric functions, the RETURN expression is converted to the type of the function.

The function end consists of a one-word statement:

FNEND

This statement must always be the last statement in the function definition.

Explanation

The body of a function can contain any BASIC/3000 statements with the following restrictions:

- Local variables except formal parameters must be declared (even if they are type real) using the type or DIM statements; they can duplicate the name of other variables because they are known only within this function and are created dynamically each time the function is called.
- A function definition cannot appear within a function body, but function calls are allowed, including calls to the same function.
- The function body must be self-contained; FOR loops and DO-blocks must be completed within the body and branches must not occur into or out of the body.

The formal parameters in a multiline function head are specified in the same way as those in the one-line function definition.

Examples

In the following multiline function, there are no local variables. It returns a long value.

```
100 DEF LONG FNX(A,B,INTEGER X,Y,LONG M,N,P)
120   M=A**-Y
130   N=B**Y
140   P=M*N
150   RETURN P
160 FNEND
```

The following multiline function definition returns an integer value:

```
210 DEF INTEGER FNM(A[*],INTEGER N)
215   REM-A(*) IS REAL ARRAY-FNM RETURNS INDEX OF LARGEST ELEMENT
220   REAL I,J
230   J=1
240   FOR I=2 TO N
250     IF A[I]>A[J] THEN J=I
260   NEXT I
270   RETURN J
280 FNEND
```

The use of the variables I and J are local to the function. It is good practice to define local variables within the function definition.

The following multiline function returns a string value; its formal parameter is a string variable:

```
10 DEF FNR$(A$)
20   REM..FNR$ RETURNS THE REVERSE OF A$
30   IF LEN(A$)<=1 THEN RETURN A$
40   RETURN FNR$(A$[2])+A$[1,1]
50 FNEND
```

The functions defined in these three examples and the one-line function definitions in the previous set of examples will be called in the examples under Calling A User-Defined Function.

Calling a User-Defined Function

A user-defined function is called by referring within an expression to the function name followed by a list of actual parameters in parentheses. The function call is replaced by the value returned by the function.

Form

A function call has the form:

function-name(actual parameter list)

string-function-name(actual parameter list)

The first form is a numeric function; a real type result is returned unless the name in the function definition is preceded by a type specification: INTEGER, LONG, or COMPLEX. The second form is a string function and a string result is returned.

The *actual parameter list* contains one or more *actual parameters* separated by commas. An *actual parameter* may be:

- numeric expression
- numeric array name followed by (*) or (*,*)
- string expression
- string variable name optionally followed by (*)
- string array name followed by (*,*)

Explanation

Actual parameters may be used to pass single values or entire arrays to a function, usually to be used within the function although this is not required. Even if parameter values are not needed in the execution of the function, at least one "dummy" parameter must be included in the formal parameter list of the function definition and in the actual parameter list of the function call.

The number of actual parameters in the function call must be the same as the number of formal parameters in the function definition. The names of corresponding parameters need not be the same. Actual and formal parameters correspond according to their positions in the two lists. For instance, the third actual parameter in a function call corresponds to the third formal parameter in the DEF statement.

If the formal parameter is a simple numeric variable (V) then the actual parameter can be a numeric expression resulting in a single value, or a simple or subscripted numeric variable (2*V,V,5*7, V(5)). If the variables are different types or the actual parameter is an expression, any necessary conversion is performed as described in Section IV, Numeric Assignment.

If the formal parameter is a simple string variable (V$ or V$(*)) the corresponding actual parameter must be either a string expression or a simple string variable ("ABC", "X", V$, V$(*), "A"+"B").

If the formal parameter is a numeric array (A(*) or A(*,*)) then the actual parameter must also be a numeric array of the same type and number of dimensions. No conversion is performed in this case.

If the formal parameter is a string array (A$(*,*)) then the corresponding actual parameter also must be a string array.

Examples

To call the one-line function:

```
10 DEF FNA(A,B)=(A*B)+(A/B)
```

the actual parameters are numeric variables of the same type:

```
500 LET X=2.57,Y=7.98
510 PRINT FNA(X,Y)
>RUN
20.8307
```

The actual parameters might also be numeric expressions:

```
520 PRINT FNA(2.57,7.98)
>RUN
20.8307
```

To call the string function:

```
20 DEF FNA$(A$,B$)=A$+B$+"STOP"
```

The actual parameters can be string variables:

```
530 X$="O",Y$=""
540 PRINT FNA$(X$,Y$)
>RUN
OSTOP
```

or string expressions:

```
550 PRINT FNA$("A","1")
>RUN
A1STOP
```

To call the function FNB returning a real value:

```
30 DEF FNB(INTEGER A,X2)=A*X2+(A/X2)
```

the actual parameters can be variables:

```
 500 LET X=150,Y=2*35
 510 PRINT FNB(X,Y)
>RUN
 10502
```

or integer expressions:

```
 520 PRINT FNB(500,2.7)
>RUN
 1666
```

Each of the above examples is a one-line function for which a single value is returned. The formal parameters are not affected by execution of the function. In a multiline function, the formal parameters may be altered in the body of the function. Depending on the type of actual parameter passed to the function, the value of the actual parameter may also be affected by the change to the formal parameter.

In the multiline function below, the values of the formal parameters are not changed by execution of the function:

```
210 DEF INTEGER FNM(A[*],INTEGER N)
215   REM-A(*) IS REAL ARRAY-FNM RETURNS INDEX OF LARGEST ELEMENT
220   REAL I,J
230   J=1
240   FOR I=2 TO N
250     IF A[I]>A[J] THEN J=I
260   NEXT I
270   RETURN J
280 FNEND
```

To call this function, the actual parameters may be an entire numeric array and a numeric variable:

```
300 DIM X[5]
305 LET Q=5
310 READ (FOR N=1 TO 5,X[N])
320 DATA 2732.1,765.32,7905.1,6543.89,195.72
330 PRINT FNM(X[*],Q)
RUN
3
```

```
 510 LET A1=8,B1=3
 520 X1=3,Y1=2
 530 PRINT "VALUE OF FNX=";FNX(A1,B1,X1,Y1,M1,N1,P1)
 540 PRINT "M1=";M1,"N1=";N1
 550 PRINT "P1=";P1
>RUN
VALUE OF FNX= 1.40625000000000L-01
M1= 1.56250000000000L-02      N1= 9.00000000000000L+00
P1= 1.40625000000000L-01
```

Note that M1, N1, and P1 do not have values prior to their use as actual parameters in the function call.

The table below summarizes the relations between actual and formal parameters:

Actual Parameter	Formal Parameter	Attribute
V, A(e), A(e,e)	V_s	reference
V, A(e), A(e,e)	V_d	value
e[1]	V	value
A(*)	A(*)	reference
A(*,*)	A(*,*)	reference
V$ or V$(*), A$(e)	V$ or V$(*)	reference
V$(e), V$(e,e), V$(e;e)	V$ or V$(*)	value
A$(e,e), A$(e,e,e), A$(e,e;e)	V$ or V$(*)	value
e$[1]	V$ or V$(*)	value
A$(*,*)	A$(*,*)	reference

V= simple numeric variable name

V_s=simple numeric variable name, same type

V_d=simple numeric variable name, different type

A= numeric array name

e= numeric expression

V$=simple string variable name

A$=string array variable name

e$=string expression

[1] not including an expression consisting of V, A(e), A(e,e), V$, V$(*), or A$(e) alone.

SECTION VII
Debugging

BASIC/3000 provides commands that allow a program to be debugged while it is running. The path of execution through a program and the change in value of variables can be traced. The dynamic nesting structure of a program can be displayed; variable values can be displayed and modified; tracing can be changed; and the execution sequence can be altered.

Note that once a program has been saved for RUNONLY (see Section II), it cannot be debugged.

TRACE/UNTRACE Commands

The TRACE command is used to turn on the tracing of selected variables, both simple variables and arrays, function references (with or without tracing their local variables), programs called with INVOKE or CHAIN, and statements of the current program. UNTRACE turns off tracing.

Form

The commands have these forms:

TRACE [trace element list]

UNTRACE [trace element list]

The *trace elements* are optional and include *variables* (including those local to functions), *functions*, *labels*, a range of labels *(label-label)*, and the keyword PROG.

Variables include simple variables, string variables, string arrays, and numeric arrays. Numeric arrays are distinguished from simple variables by a (*) or (*,*) following the array name. Examples of variables:

A, B(2)	numeric variable
B$, C$(*)	string variable
C$(*,*)	string array
A(*)	one-dimensional numeric array
A(*,*)	two-dimensional numeric array

Functions can be specified by the function name only or the name followed by a list of local variables in parentheses. For example:

FNA	numeric function name
FNB(A$,B(*))	local string variable and local numeric array; FNB is not traced, only the local variables.

Labels are statement labels consisting of integers in the range 1 through 9999.

Label-label stands for all statements between, and including the statements identified by the two labels. For example:

80	the statement at label 80
100-150	all statements between 100 and 150 inclusive

PROG is a keyword to specify that trace information be printed when a CHAIN or INVOKE is performed.

Explanation

The BASIC/3000 Interpreter keeps track of all items specified in TRACE commands. A range of labels traces all statements within the range. Any change to a variable's value, any reference to a function, or any execution of a statement causes tracing information to be printed. Any change in the value of an array element, except in MAT operations, causes that element to be printed. To trace a function, only the function name is specified. If a list of local variables is included with the function name, changes in their values are traced but not execution of the function. If PROG is specified in a TRACE command, information is printed whenever a program is accessed through an INVOKE or CHAIN statement and also when returning to an invoking program (see Section X, Segmentation).

TRACE with no parameters lists the items currently being traced.

UNTRACE turns off tracing of the items specified. When tracing is turned off for a function it is not turned off for any local variables. When tracing is turned off for variables individually, it does not turn off tracing for the function. For instance, UNTRACE FNA turns off tracing of the function but not of its local variables if they were specified in a TRACE command; UNTRACE FNA (A$) turns off tracing of the local variable A$ but not of the function FNA.

UNTRACE with no parameters stops all tracing specified by previous TRACE commands.

The specified tracing occurs only when the program is executed. When trace output occurs, the following information is printed:

@ *label variable = value*
or
@ *programname label variable = value*

These outputs are printed as a result of a trace of a function or variable.

The programname is printed if the current program has been named.

*TRACE *label*
or
*TRACE *programname label*

These outputs are printed as a result of a trace of a label or labels.

Examples

The program below includes a one-dimensional array, a simple variable, and a function call. This program will be used for succeeding TRACE and UNTRACE examples.

```
  10 DIM X[5]
  20 LET Q=5
  30 READ (FOR N=1 TO 5,X[N])
  40 DATA 2.5,75.6,36.2,15.7,100
  50 PRINT FND(X[*],Q)
 210 DEF FND(A[*],INTEGER N)
 220   REAL I,J
 230   J=1
 240   FOR I=2 TO N
 250     IF A[I]>A[J] THEN J=I
 260   NEXT I
 270   RETURN J
 280 FNEND
>TRACE Q,X(*)
>RUN
@20 Q=5
@30 X[1]=2.5
@30 X[2]=75.6
@30 X[3]=36.2
@30 X[4]=15.7
@30 X[5]=100
 5
```

The command TRACE Q,X(*) traces the variable Q and prints its value at line 20. This is the only value printed for Q since its value is not subsequently changed. Then the five values of the numeric array X(*) are printed with the line at which they assume these values.

The program output in this case is 5, the result returned by the function FND.

This trace is turned off by:

```
>UNTRACE
```

UNTRACE without a trace element list turns off all current traces.

In the trace below of the function FND and its local variables A(*) and N, the only trace output is for the function FND; its local variables are not traced since their values are unchanged by program execution.

```
>TRACE FND,FND(A(*),N)
>RUN
@FND
@270 FND(FND)=5
 5
```

A trace of statement execution is printed when a label or range of labels are used as parameters:

```
>TRACE 50,200-280
>RUN
*TRACE  50
@FND
*TRACE  220
*TRACE  230
*TRACE  240
*TRACE  250
*TRACE  260
*TRACE  250
*TRACE  260
*TRACE  250
*TRACE  260
*TRACE  250
*TRACE  260
*TRACE  270
@270 FND(FND)=5
 5
*TRACE  210
```

Because of the FOR statement in line 240, lines 250 and 260 are repeated 4 times. Following execution of line 270, control passes to line 50 where the value returned by the function is printed. The program then reaches line 210 and halts.

The command TRACE with no list prints the current trace element list.

```
>TRACE
FND(A[* ],N),50,200-280.
```

Following the UNTRACE command, the TRACE command has nothing to list.

```
>UNTRACE
>TRACE
>
```

The use of TRACE PROG is illustrated by the following segmented programs. ALPHA1 calls program BETA1 which in turn calls GAMMA1. Because the INVOKE statement is used, control returns to ALPHA1 (see Section X, Segmentation).

```
GAMMA1
  10 REM      PROGRAM GAMMA1
  20 PRINT "IN GAMMA1  -- RETURN TO BETA1"


BETA1
  10 REM      PROGRAM BETA1
  20 INVOKE "GAMMA1"
  25 PRINT "BACK IN BETA1  --  RETURN TO ALPHA1"


ALPHA1
  10 REM      PROGRAM ALPHA1
  20 INVOKE "BETA1"
  25 PRINT "BACK IN ALPHA1  -- TERMINATE"


>TRACE PROG
>RUN
ALPHA1
*TRACE, INVOKE: BETA1
*TRACE, INVOKE: GAMMA1
IN GAMMA1  -- RETURN TO BETA1
*TRACE, REVERT: BETA1
BACK IN BETA1  --  RETURN TO ALPHA1
*TRACE, REVERT: ALPHA1
BACK IN ALPHA1  -- TERMINATE
```

BREAK/UNBREAK Commands

The BREAK command allows the user to specify points where the execution of a program should be interrupted (or "broken"). A break point is a label, a range of labels, or any point at which a transfer is made from one program to another (through CHAIN, INVOKE, or END). UNBREAK turns off break points.

Forms

The forms of the commands are:

BREAK

BREAK breakpoint list

UNBREAK

UNBREAK breakpoint list

The items in the optional *breakpoint list* include: *label*, *label-label*, and PROG. They are specified in the same way as for TRACE.

Explanation

BREAK or UNBREAK can be specified before the program is run or when it is broken. When the program is run, execution suspends just before execution of a statement whose label is in the breakpoint list. If PROG is specified, execution suspends after a program is brought into the user's work area by CHAIN, INVOKE, or END but before the program is run (see Section X, Segmentation).

When execution suspends as a result of a breakpoint, the statement label about to be executed is printed in the form:

*BREAK *label*

*BREAK *programname label*

The *programname* is listed only if the program has been named.

After this output, a > is printed to indicate that a command can be entered. The legal commands during a break period are listed below.

Execution is resumed with the RESUME or GO command.

BREAK with no parameters causes all the current breakpoints to be listed.

UNBREAK with no parameters deletes all current breakpoints. With parameters, UNBREAK deletes those breakpoints specified by the labels in the breakpoint list.

LEGAL COMMANDS DURING BREAK

Certain commands may be used only during a break period:

ABORT

CALLS

FILES

GO

RESUME

SET

SHOW

These commands are described in this section.

Of the remaining commands, only these are legal during a break period:

BREAK/UNBREAK

CATALOG

CREATE

DUMP

EXIT

KEY

LENGTH

LIST

SPOOL

SYSTEM

TRACE/UNTRACE

XEQ

If the user enters any other command, BASIC/3000 responds:

ILLEGAL WHILE RUN SUSPENDED. DO YOU WANT TO ABORT?

The user enters anything starting with "Y" to abort the current program and carry out the command, or enters anything else not to abort the program and to ignore the command.

During a break period, the user can type ABORT to terminate his current run and return to BASIC command mode where all commands are legal.

The commands SCRATCH and GET (illegal during a break period) will clear all traces and breakpoints. RUN *programname* will also clear traces and breakpoints, but a RUN without the *programname* parameter will not.

The CHAIN and INVOKE commands clear traces and breakpoints except when PROG is used. INVOKE saves traces and breakpoints and restores them upon return to the invoking program.

Examples

```
   10 DIM A[5,10]
   20 MAT READ A
   30 DATA 10,20,30,40,50,100,200,300,400,500
   40 DATA 110,120,130,140,150,210,220,230,240,250
   50 DATA 310,320,330,340,350,410,420,430,440,450
   60 DATA 510,520,530,540,550,610,620,630,640,650
   70 DATA 710,720,730,740,750,810,820,830,840,850
   80 RESTORE 50
   90 READ (FOR X=1 TO 3,(FOR Y=1 TO 10,A[X,Y]))
  100 END
>BREAK 30,100
>RUN
*BREAK  30
>SHOW A(1,1),A(5,10)
A[1,1]=10
A[5,10]=850
>GO
*BREAK  100
>SHOW A(1,1),A(5,10)
A[1,1]=310
A[5,10]=850
>UNBREAK
>ABORT
```

During the breakpoints, the command SHOW (described later in this section) causes the values of the specified elements to be printed. After the program has run with two breakpoints at line 30 and 100, the breakpoints are deleted with UNBREAK. ABORT is used to return the user to the BASIC command mode. He may then run the program without breakpoints.

In the example below, the same program is named BRK1 and then run with the same breakpoints. The breakpoints are listed with BREAK during the second breakpoint and then deleted individually. GO finishes execution of the program after which the user is returned to BASIC command mode and runs the program without breakpoints:

```
 10 DIM A[5,10]
 20 MAT READ A
 30 DATA 10,20,30,40,50,100,200,300,400,500
 40 DATA 110,120,130,140,150,210,220,230,240,250
 50 DATA 310,320,330,340,350,410,420,430,440,450
 60 DATA 510,520,530,540,550,610,620,630,640,650
 70 DATA 710,720,730,740,750,810,820,830,840,850
 80 RESTORE 50
 90 READ (FOR X=1 TO 3,(FOR Y=1 TO 10,A[X,Y]))
100 END


>NAME BRK1
>BREAK 30,100
>RUN
BRK1
*BREAK BRK1 30
>SHOW A(1,1)
A[1,1]=10
>GO
*BREAK BRK1 100
>SHOW A(1,1)
A[1,1]=310
>UNBREAK 100
>BREAK
30.
>UNBREAK
>GO

>RUN
BRK1
```

ABORT Command

The ABORT command is legal only during a break period; it terminates the suspended program and returns the user to a normal state where all commands are legal.

Form

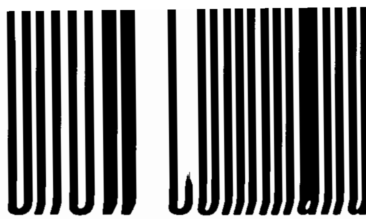
ABORT

Explanation

When ABORT is specified, the break period is ended and the run terminated. The user can now enter any command legal during normal BASIC execution, but cannot enter the commands legal only during a break period.

Examples

```
  10 DIM A[5,10]
  20 MAT READ A
  30 DATA 10,20,30,40,50,100,200,300,400,500
  40 DATA 110,120,130,140,150,210,220,230,240,250
  50 DATA 310,320,330,340,350,410,420,430,440,450
  60 DATA 510,520,530,540,550,610,620,630,640,650
  70 DATA 710,720,730,740,750,810,820,830,840,850
  80 RESTORE 50
  90 READ (FOR X=1 TO 3,(FOR Y=1 TO 10,A[X,Y]))
 100 END
>BREAK 30,100
>RUN
*BREAK  30
>SHOW A(5,10)
A[5,10]=850
>ABORT
```



The SHOW command prints the values of the items specified; this command is legal only during a break period.

Form

The form of SHOW is

SHOW item list

The list can include:

- variables (numeric or string)
- array elements
- entire arrays (*name* (*) for one-dimensional array or *name* (*,*) for two-dimensional array)
- local variables (*function name* (*variable list*))

Explanation

An array is printed as in the MAT PRINT statement (see Section III), except that undefined values are noted with the word UNDEFINED. The variable list in parentheses that follows a function name can include only local variables of that function. The function must be active; that is, the function must have been called and not be completed.

Examples

The example below specifies breakpoints for line 20 and lines 70 through 90. Since line 80 is not executed, breaks actually occur in lines 70 and 90. SHOW commands are used to print the contents of the variable X$ at the break in line 70 and the contents of the local variable A$ in function FNR$ at the break in line 20. At the break in line 90, an attempt is made to show the contents of the non-existent array and of an existing array that has not been given any values. A new breakpoint at line 100 is specified where the SHOW command is used to print the previously undefined array B. GO continues execution of the program until it ends.

After the FORTRAN segment BOOLEAN is compiled, it is added to the SL (Segmented Library) using the Segmenter. An SL list is then requested that shows the segment BOOLEAN as the only segment in the SL of the user's group/account STUDENTS.CLASS:

```
:FORTRAN  BOOLEAN

PAGE 0001   HEWLETT-PACKARD 32102A.00.2   FORTRAN/3000     TUE, MAR 27,
            1973,  5:55 AM

00200000  $CONTROL  USLINIT, NOLIST, SEGMENT=BOOLEAN

END OF PROGRAM


:SEGMENTER

SEGMENTER SUBSYSTEM (1.2)
-USL $OLDPASS
-SL SL
-ADDSL BOOLEAN
-LISTSL

SL FILE SL.STUDENTS.CLASS

SEGMENT   0 BOOLEAN           LENGTH    30

   ENTRY POINTS     CHECK CAL STT  ADR
   LOR             141400   C   2     6
   LNOT            141000   C   1     0
   LAND            141400   C   3    15

   EXTERNALS        CHECK STT SEG

1

USED                   2200           AVAILABLE          375600

-EXIT

END OF PROGRAM
```

Notice, the -SL SL command works only if the SL exists. To build an SL enter the command:

BUILDSL SL[.group], records, extents

then enter the -SL SL command.

Library usage is described in the MPE/3000 Operating System, Reference Manual.

2. Calling an SPL Procedure

The SPL procedure WHOM is called by the BASIC program ZELDA. WHOM returns the user name, group, and account with which the user logged on, as well as the user's home group. The BASIC program uses this information plus the current time and date from the DAT$ function to print output as if from Zelda, the fortune teller.

The username, account, group, and home group are stored in string variables used as actual parameters in the call to WHOM.

The BASIC program:

```
ZELDA
  10 D$=DAT$(1,27)
  20 MAT READ M$
  30 CALL WHOM(U$,G$,A$,H$)
  40 REM
  50 REM
  60 PRINT LIN(2);"I'M ZELDA THE FORTUNE TELLER,"
  70 PRINT "I CAN TELL YOU A LOT."
  80 FOR I=1 TO 11
  90   IF POS(M$[I],D$[6;3]) THEN 110
 100 NEXT I
 110 PRINT LIN(1);"ON THIS ";M$[I];" ";DEB$(D$[10;2]);FNC$(D$[10;2]);","
 120 PRINT "IT'S ";DEB$(D$[20;5]);" O'CLOCK."
 130 PRINT LIN(1);"YOUR ACCOUNT NAME IS "'34;DEB$(A$);'34","
 140 PRINT "AND YOU'RE IN GROUP "'34;DEB$(G$);'34"."
 150 PRINT LIN(1);"YOUR SIGN-ON WAS "'34;DEB$(U$);'34","
 160 IF LEN(DEB$(H$)) THEN DO
 170   IF G$<>H$ THEN PRINT "AND "'34;DEB$(H$);'34" IS YOUR HOME."
 180   ELSE PRINT "AND YOU'RE IN YOUR HOME GROUP."
 190 DOEND
 200 ELSE PRINT "AND YOU HAVE NO HOME GROUP."
 210 PRINT LIN(1);"BUT ENOUGH SAID FOR NOW.";LIN(2)
 220 REM
 230 REM            ********************
 240 REM
 250 DEF FNC$(C$)
 255   IF C$="11" OR C$="12" OR C$="13" THEN RETURN "TH"
 260   IF C$[2]="1" THEN RETURN "ST"
 270   IF C$[2]="2" THEN RETURN "ND"
 280   IF C$[2]="3" THEN RETURN "RD"
 290   RETURN "TH"
 300 FNEND
 310 DIM D$[27],U$[8],A$[8],G$[8],H$[8],M$[12,9]
 320 INTEGER I
 330 DATA "JANUARY","FEBRUARY","MARCH","APRIL","MAY","JUNE","JULY"
 340 DATA "AUGUST","SEPTEMBER","OCTOBER","NOVEMBER","DECEMBER"
```

In these examples the user's name is JOHNDOE, his home group is STUDENTS, and his account is CLASS. When run under the group STUDENTS, the result is:

```
I'M ZELDA THE FORTUNE TELLER,
I CAN TELL YOU A LOT.

ON THIS MARCH 3ØTH,
IT'S 4:35 O'CLOCK.

YOUR ACCOUNT NAME IS "CLASS",
AND YOU'RE IN GROUP "STUDENTS".

YOUR SIGN-ON WAS "JOHNDOE",
AND YOU'RE IN YOUR HOME GROUP.

BUT ENOUGH SAID FOR NOW.
```

When ZELDA is run under the group PUB, the result is:

```
I'M ZELDA THE FORTUNE TELLER,
I CAN TELL YOU A LOT.

ON THIS MARCH 3ØTH,
IT'S 4:3Ø O'CLOCK.

YOUR ACCOUNT NAME IS "CLASS",
AND YOU'RE IN GROUP "PUB".

YOUR SIGN-ON WAS "JOHNDOE",
AND "STUDENTS" IS YOUR HOME.

BUT ENOUGH SAID FOR NOW.
```

The SPL procedure WHOM, compiled into segment WHOMSEG, is listed below:

```
$CONTROL   SEGMENT=WHOMSEG
$CONTROL   USLINIT, NOLIST, SUBPROGRAM
BEGIN
PROCEDURE WHOM(USERNAME,LOGONGROUP,LOGONACCT,HOMEGROUP);
          BYTE ARRAY USERNAME,LOGONGROUP,LOGONACCT,HOMEGROUP;
  BEGIN
    EQUATE CODES3=[6/1,5/1,5/1],        <<CODEWORD FOR THREE STRINGS>>
           CODE1=[6/1,10/0];        <<CODEWORD FOR ONE STRING>>
    INTEGER DELTAQ=Q-0;        <<DELTA-Q IN STACK MARKER>>
    INTEGER POINTER NUMBER;        <<WILL POINT TO # OF PARAMS PASSED.
                                     "NUMBER(1)" AND "NUMBER(2)" ARE
                                     CODEWORDS PASSED BY INTERPRETER   >>
    INTRINSIC WHO;
      <<>>
    @NUMBER:=@DELTAQ-DELTAQ+1;
    IF  NUMBER=4 AND NUMBER(1)=CODES3 AND NUMBER(2)=CODE1  THEN
      IF  USERNAME(-2)>=8 AND LOGONGROUP(-2)>=8 AND LOGONACCT(-2)>=8
          AND HOMEGROUP(-2)>=8      <<"...(-2)" IS PHYSICAL LENGTH>>
        THEN BEGIN
          WHO(,,,USERNAME,LOGONGROUP,LOGONACCT,HOMEGROUP);
          USERNAME(-1):=LOGONGROUP(-1):=LOGONACCT(-1):=HOMEGROUP(-1):=8;
                                              <<SET LOGICAL LENGTH OF STRINGS>>
        END;
    RETURN 0;                 <<IN CASE OF ERROR IN CALLING SEQUENCE, LET
                                 INTERPRETER CLEAN UP STACK                >>
  END;  <<PROCEDURE WHOM>>
END.
```

The WHO instrinsic called by WHOM is an option variable procedure provided by MPE/3000. It cannot be called directly from BASIC/3000, but can be accessed indirectly, as in this example, through an SPL procedure.

The SPL procedure verifies the calling sequence by first checking the number of parameters, then by looking at the type codes in the code words. This routine verifies that the physical length of the strings is at least large enough to contain the strings returned by WHO.

When called by WHOM, WHO returns the log-on user name, the account and group names, and the user's home group. WHOM passes this information to the BASIC calling program ZELDA after setting the logical length of the strings returned by WHO to 8.

The RETURN 0 (rather than RETURN) leaves the parameters in the stack in case there was some error in the calling sequence. Whenever there is a possibility of an error in the calling sequence, RETURN 0 should be used to exit from an SPL procedure.

The user should consult the SPL/3000 Reference Manual for instructions on writing an SPL procedure.

The file WHOMPROG contains the source of WHOM. WHOM is compiled into the segment WHOMSEG. After WHOM is compiled it is added to the SL (Segmented Library) using the Segmenter. It is then listed, showing entry points, external procedures, its length, and so forth.

The command :SPL WHOMPROG requests compilation of WHOMPROG:

```
:SPL  WHOMPROG

PAGE 0001   HP32100A.02.0


00200100  00000 0   $CONTROL   SEGMENT=WHOMSEG
00200200  00000 0   $CONTROL   USLINIT, NOLIST, SUBPROGRAM
 PRIMARY DB STORAGE=%000;   SECONDARY DB STORAGE=%00000
 NO. ERRORS=000;            NO. WARNINGS=000
 PROCESSOR TIME=0:00:02;    ELAPSED TIME=0:00:30

END OF PROGRAM
```

The command :SEGMENTER and subsequent commands add the segment WHOMSEG to the account SL called SL.PUB, and then lists the SL:

```
:SEGMENTER

SEGMENTER SUBSYSTEM (2.0)
-SL SL.PUB
-USL $OLDPASS
-ADDSL WHOMSEG
-LISTSL

SL FILE SL.PUB.CLASS

SEGMENT   0 WHOMSEG          LENGTH   100

   ENTRY POINTS     CHECK CAL STT  ADR
   WHOM               0    C    1     0

   EXTERNALS        CHECK STT SEG
   WHO                0    2   ?

1

USED                  1600               AVAILABLE                 27200


-EXIT

END OF PROGRAM
```

For SL library usage, see the MPE/3000 Operating System, Reference Manual.

SYSTEM/RESUME Commands

The SYSTEM command is used to enter control of the MPE/3000 Operating System. The user's activity in BASIC/3000 is suspended and may be resumed with the RESUME command.

Form

The form of SYSTEM is:

SYSTEM

The form of RESUME is:

RESUME

Like all BASIC/3000 commands, the prompt for SYSTEM is >. The prompt for RESUME is : since RESUME is entered from MPE.

Explanation

When SYSTEM is typed, BASIC is suspended and the user enters MPE/3000. A colon (:) is output as a prompt signal and he may then type any MPE commands. The BREAK key may also be used to suspend BASIC. Usually there is no difference between the BREAK key and the SYSTEM command. However, when using BASIC in the batch processing mode (see Section XI) or at a terminal that does not have a BREAK key, the SYSTEM command is the only way to enter MPE without terminating BASIC.

When through with MPE, the user returns to the suspended BASIC operation by typing RESUME.

Example

```
>10 DIM A$(72)
>20 READ A1,A2,A3,A4,A5
>SYSTEM

:BUILD AA;REC=-72,,,ASCII;DISC=100
:RESUME
>30 PRINT #1,1;A1,A2,A3,A4,A5
>40 RESTORE #1
>50 LINPUT #1;A$
>60 PRINT A$
>70 DATA 10,20,30,40,50
>80 FILES AA
>RUN
 10               20               30               40               50
```

In this example, the user leaves the BASIC/3000 Interpreter to create an ASCII file. This can only be done with the MPE BUILD command. After creating the ASCII file, he returns to BASIC with RESUME and uses the ASCII file.

SYSTEM Statement

The SYSTEM statement provides a means to dynamically enter an MPE/3000 system command from a BASIC/3000 program. The command will be executed at the position in the program occupied by the SYSTEM statement.

Form

SYSTEM numeric variable,system command

The *system command* is specified as a string expression. The initial colon is not included in the command specification.

The *numeric variable* returns 0 if the command succeeds, or the MPE command error number if the command fails.

Explanation

When a system command is required within a BASIC program, the SYSTEM statement can be used to execute such a command during execution of the BASIC program.

Example

```
 10 REM..USE SYSTEM STATEMENT TO ENTER MPE COMMAND
 20 SYSTEM X,"FILE ABC;DEV=TAPE"
 30 IF X<>0 THEN 120
 40 FILES A
 50 DIM X[10]
 60 MAT READ X
 70 DATA 10,20,30,40,50,60,70,80,90,100
 80 MAT PRINT #1;X
 85 ASSIGN *,1
 90 SYSTEM Y,"STORE A;*ABC"
100 IF Y<>0 THEN 120
110 END
120 PRINT "SYSTEM STATEMENT FAILED"
>RUN
 FILES STORED = 1

 FILES NOT STORED = 0
```

The SYSTEM statement in line 20 causes execution of the MPE/3000 FILE command that assigns a file ABC to tape. In line 90 the SYSTEM statement causes execution of the MPE/3000 STORE command that stores the BASIC file A on the tape file ABC.

SECTION XII
Non-Interactive Programming

BASIC/3000 has the capability to enter programs in a non-interactive manner. This section describes how the user may input from a card reader or paper tape, and how he may print output on a line printer or punch it on paper tape. It also describes how to input commands or programs stored on an ASCII file.

Card Reader/Line Printer

If the user has access to a card reader and a line printer, he may punch his BASIC program on cards and input it through the card reader and receive output on the line printer. In addition to the BASIC program cards, he will need a :BASIC command card preceding his program deck. The deck may include any BASIC commands as well as statements.

:BASIC Command

This command causes the MPE/3000 Operating System to invoke the BASIC Interpreter. The :BASIC command has the form:

:BASIC commandfile,inputfile,listfile

Any or all of the parameters may be omitted. Position of parameters is significant so commas must be included when leading parameters are omitted. Each parameter must be an existing ASCII file with the following meaning:

commandfile	BASIC subsystem input; original source of all commands and statements of BASIC program. Default is $STDINX.
inputfile	BASIC program input; contains data input to BASIC through INPUT, ENTER, and LINPUT statements. Default is $STDINX.
listfile	BASIC program output; receives data output from BASIC program. Default is $STDLIST.

$STDINX is the job/session input device. For example, when input is from the card reader, $STDINX is the card reader; when input is from the terminal, $STDINX is the terminal.

$STDLIST is the job/session output device. When the card reader is used for input, the job/session output device is normally the line printer although this may vary between HP 3000 installations. When the terminal is used, the output device is also the terminal.

Examples:

1. Suppose the user inputs his program from the card reader with output to the line printer, but all the data used by the program is stored on an ASCII disc file called IN. He uses the :BASIC command:

```
:BASIC ,IN
```

The comma signals that the commandfile is the default file $STDINX. The default $STDLIST is used for output but since this is at the end, no comma is needed.

Note that this card, if entered as a command from a terminal, would perform the same function if program input and output were both at the terminal with data input from IN.

2. Suppose the user has a disc file named COMMAND which contains a set of BASIC commands, and he wants output on the line printer:

```
:FILE PRINTER;DEV=LP

:BASIC COMMAND,,*PRINTER
```

The file named PRINTER is associated with the line printer by the :FILE command. It is named in the :BASIC command to replace the $STDLIST which, in this case, would have been the terminal. COMMAND is a disc file; the commands and statements of the BASIC program are read from this file.

> EOD Command

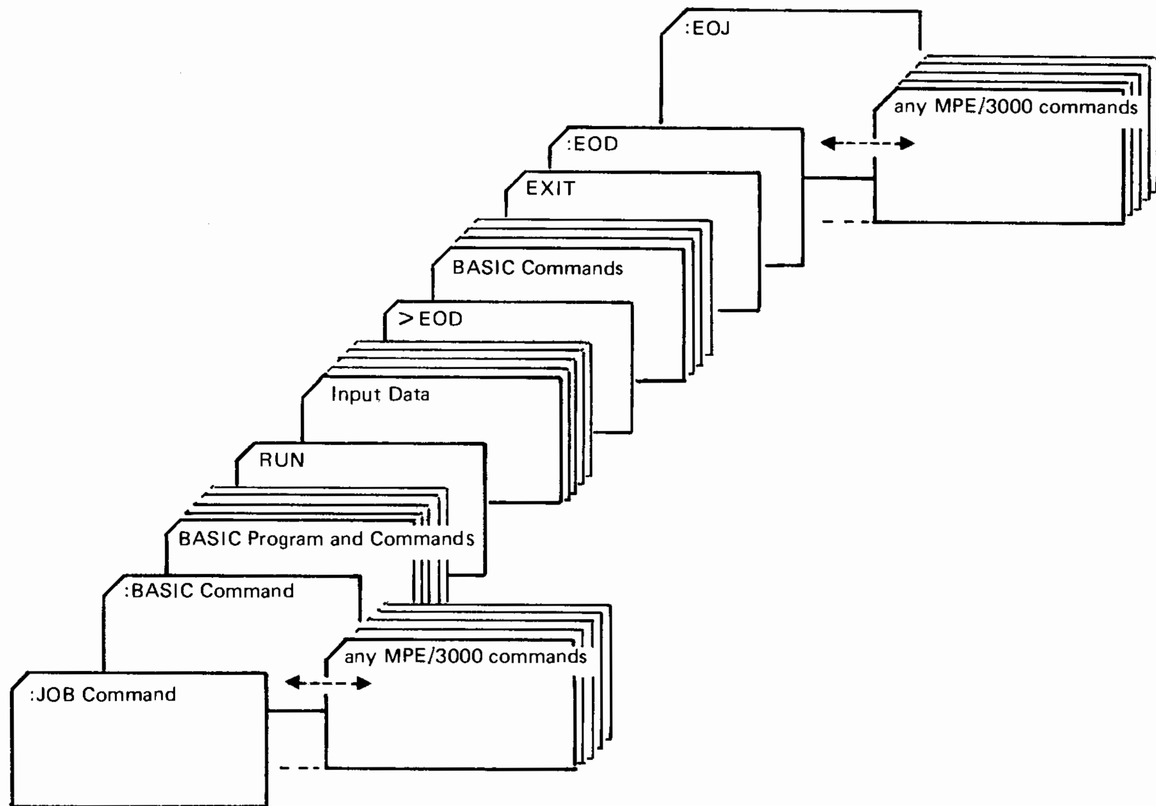
If a BASIC card deck contains a RUN command, a >EOD command must follow RUN. When the BASIC program contains an INPUT, ENTER, or LINPUT statement that requests input, the input data is punched on cards that are placed in the deck between the RUN card and the >EOD card.

Deck Structure

In addition to the :BASIC command and the program deck, the standard MPE/3000 command cards for running a job must be used. These are the :JOB card preceding all other cards in the job, and the :EOJ card that terminates the job. Also, an EXIT card must terminate the BASIC program deck followed by an :EOD card.

Besides these cards, any other MPE/3000 cards needed for the job may be included.

A deck structure for a BASIC program that requests input with INPUT, ENTER, or LINPUT:



If no input data is required, the input data cards are omitted, but >EOD is left. The >EOD card must follow the input data of each run.

Paper Tape

At a terminal with a paper tape punch and reader, the user may write his programs on paper tape and read them from paper tape. Commands are provided by BASIC/3000 that enable the terminal user to prepare and use paper tapes.

PREPARING A PAPER TAPE

A paper tape may be punched on-line using the PUNCH command, or it may be punched off-line.

PUNCH Command

The PUNCH command allows the user to punch a program on paper tape while operating in interactive (on-line) mode at the terminal. Following each line, PUNCH automatically inserts an X-OFF character preceding the carriage return and linefeed. A paper tape prepared by PUNCH should be read with the TAPE command.

Form

PUNCH [first [-last]] [, OUT = asciifile] [,RECSIZE = number] [,NONAME]

where *first* and *last* specify the range of statements to punch, and *asciifile* specifies an output file. Normally, the output file is the standard list file. If neither *first* nor *last* is specified, the entire program is punched. If only *first* is specified, just that statement is punched. Control characters are inserted, as required, to allow reading the punched program back through a tape reader.

Explanation

The PUNCH command is identical to LIST except that the output is preceded and followed by headers and trailers of null characters. If the OUT parameter is omitted, the punched program is listed at the terminal. Otherwise it is output to the specified ASCII file.

If only a portion of the program is to be punched, the first and, optionally, the last lines to be punched are specified. If a maximum record size other than 72 is desired, it can be specified with the RECSIZE parameter. If *NONAME* is specified, the program name is not punched; this is useful when punching programs to be read back with the XEQ command.

Each punched record contains a program statement. PUNCH automatically terminates each record with an X-OFF character. The X-OFF precedes the carriage return on the tape, and linefeed follows the carriage return. The form is:

output record X-OFF carriage return linefeed

Examples:

```
PUNCH 10-200
```

Lines 10 through 200 of the current BASIC/3000 program are punched on paper tape.

```
PUNCH,OUT=AA
```

The entire current program is punched on the ASCII file AA.

```
PUNCH 500,RECSIZE=132
```

Line 500 of the program is punched. A record size of 132 characters is used.

PUNCHING PAPER TAPE OFF-LINE

To prepare a BASIC program on paper tape, the user must:

1. Turn teleprinter control knob to "LOCAL".
2. Turn on the tape punch by pressing the "ON" button on the punch.
3. Type a series of null characters using the "HERE IS" key or control shift P ($@^c$). This punches leading holes on the tape.
4. Type the program as usual following each line with a carriage return.
5. Type a series of null characters using the "HERE IS" key or control shift P ($@^c$). This punches trailing holes on the tape.
6. Turn off the tape punch by pressing the "OFF" button on the punch.

When programs are punched off-line, the H^c and X^c keys may be used for corrections. If X^c is used to delete a line, it must be followed by X-OFF and a carriage return and linefeed.

If the punched tape is to be read by the TAPE command, the user must press the X-OFF character following each line *before* he presses the carriage return. X-OFF (S^c) is a key on the teleprinter keyboard.

READING A PAPER TAPE

Two commands are provided to read paper tapes at a terminal equipped with a paper tape reader; they are TAPE and SPOOL.

TAPE Command

The TAPE command allows the user to read commands, programs, and data through a paper tape reader connected to the terminal. Only tapes that contain an X-OFF character after each record can be read with TAPE. Also, in order to read with TAPE, the terminal must be equipped with a reader that recognizes X-OFF.

Form

TAPE

Explanation

After typing TAPE, turn on the tape reader. The tape will be read until the end of the program or programs on the tape. If the tape contains data to be input, as much data is read as was requested by INPUT, ENTER, or LINPUT, or until the end of the tape.

When through reading from tape, the user returns to terminal mode with the KEY command.

KEY Command

The KEY command returns the terminal user to terminal mode following completion of a tape read using TAPE.

Form

KEY

Explanation

When KEY is typed, the input mode entered with TAPE is terminated and the user is returned to the terminal for further interactive execution or to log off.

SPOOL Command

The SPOOL command is used to read paper tapes that have not been punched with X-OFFs preceding the carriage return and linefeed.

Form

SPOOL

Explanation

After typing SPOOL, turn on the tape reader. When the tape is through, the user types the control key Y^c to terminate the tape. At this point, any error messages are printed at the terminal.

Command Input from Files

If commands or programs are stored on an ASCII file, the XEQ command may be used to cause the BASIC/3000 Interpreter to read this file.

Form

XEQ asciifile ,ECHO

The *asciifile* has been created and contains commands or programs the user needs. If ECHO is specified, the records from the ASCII file are listed on the terminal as they are input.

Explanation

When the XEQ command is entered, the specified file is read and executed until it reaches an end-of-file. Any program input is still read from the *inputfile.* When the end-of-file of the *asciifile* is reached, control returns to the original command file. For instance, if the job was entered on cards with the :BASIC command, control returns to the *commandfile* specified in that command; if the user entered XEQ from the terminal, control resumes at the terminal.

An XEQ command within the XEQ file will close the first file and open a new one.

Examples:

An ASCII file BATCHJOB contains the commands to GET and RUN three programs (AA, BB, and CCC) stored in the user's library. In order to run all three programs at a terminal or in a BASIC card deck, only the command XEQ BATCHJOB is required.

This is illustrated in the examples below, one with the ECHO parameter, one without:

```
>XEQ BATCHJOB,ECHO
GET AA
RUN
AA
END OF AA

GET BB
RUN
BB
END OF BB

GET CCC
RUN
CCC
END OF CCC
```

ECHO causes a list of the commands in BATCHJOB. When each program is run, it outputs a message that it is through. This is part of the individual programs, not BATCHJOB.

The example below without ECHO runs each program consecutively in the same way as the previous example, but it does not list the commands contained in the file BATCHJOB:

```
>XEQ BATCHJOB
AA
END OF AA

BB
END OF BB

CCC
END OF CCC
```

APPENDIX A
ASCII Character Set

Graphic	Decimal Value	Comments
	0	Null
	1	Start of heading
	2	Start of text
	3	End of text
	4	End of transmission
	5	Enquiry
	6	Acknowledge
	7	Bell
	8	Backspace
	9	Horizontal tabulation
	10	Line feed
	11	Vertical tabulation
	12	Form feed
	13	Carriage return
	14	Shift out
	15	Shift in
	16	Data link escape
	17	Device control 1
	18	Device control 2
	19	Device control 3
	20	Device control 4
	21	Negative acknowledge
	22	Synchronous idle
	23	End of transmission block
	24	Cancel
	25	End of medium
	26	Substitute
	27	Escape
	28	File separator
	29	Group separator
	30	Record separator
	31	Unit separator
	32	Space
!	33	Exclamation point
"	34	Quotation mark

Graphic	Decimal Value	Comments
#	35	Number sign
$	36	Dollar sign
%	37	Percent sign
&	38	Ampersand
'	39	Apostrophe
(	40	Opening parenthesis
)	41	Closing parenthesis
*	42	Asterisk
+	43	Plus
,	44	Comma
–	45	Hyphen (Minus)
.	46	Period (Decimal)
/	47	Slant
0	48	Zero
1	49	One
2	50	Two
3	51	Three
4	52	Four
5	53	Five
6	54	Six
7	55	Seven
8	56	Eight
9	57	Nine
:	58	Colon
;	59	Semicolon
<	60	Less than
=	61	Equals
>	62	Greater than
?	63	Question mark
@	64	Commercial at
A	65	Uppercase A
B	66	Uppercase B
C	67	Uppercase C
D	68	Uppercase D
E	69	Uppercase E
F	70	Uppercase F
G	71	Uppercase G
H	72	Uppercase H
I	73	Uppercase I
J	74	Uppercase J
K	75	Uppercase K
L	76	Uppercase L
M	77	Uppercase M
N	78	Uppercase N
O	79	Uppercase O
P	80	Uppercase P
Q	81	Uppercase Q
R	82	Uppercase R

Graphic	Decimal Value	Comments
S	83	Uppercase S
T	84	Uppercase T
U	85	Uppercase U
V	86	Uppercase V
W	87	Uppercase W
X	88	Uppercase X
Y	89	Uppercase Y
Z	90	Uppercase Z
[	91	Opening bracket
\	92	Reverse slant
]	93	Closing bracket
^	94	Circumflex
_	95	Underscore
`	96	Grave accent
a	97	Lowercase a
b	98	Lowercase b
c	99	Lowercase c
d	100	Lowercase d
e	101	Lowercase e
f	102	Lowercase f
g	103	Lowercase g
h	104	Lowercase h
i	105	Lowercase i
j	106	Lowercase j
k	107	Lowercase k
l	108	Lowercase l
m	109	Lowercase m
n	110	Lowercase n
o	111	Lowercase o
p	112	Lowercase p
q	113	Lowercase q
r	114	Lowercase r
s	115	Lowercase s
t	116	Lowercase t
u	117	Lowercase u
v	118	Lowercase v
w	119	Lowercase w
x	120	Lowercase x
y	121	Lowercase y
z	122	Lowercase z
{	123	Opening (left) brace
|	124	Vertical line
}	125	Closing (right) brace
~	126	Tilde
	127	Delete

APPENDIX B
Error Messages

Four types of errors may cause error messages: command errors, statement syntax errors, compile errors, and run errors resulting from program execution.

Command Errors

Command error messages are printed following the command that caused the error. If the message is preceded by the word "WARNING:", the command is accepted. Otherwise, the command will be dropped and must be entered again.

Syntax Errors

When a syntax error in a statement is detected, the following message is printed:

ERROR@*integer*

where *integer* is the number of non-blank characters successfully processed before the error was detected. The user may type a carriage return and enter the statement correctly, or he may type any other character to request printing of the syntax error message. If the message is preceded by the word "WARNING:", the line is accepted and need not be re-entered.

Compile Errors

These errors are detected following a RUN command but before execution of the program. If the error message is preceded by the word "WARNING:", compilation continues. If compilation results in no message or only WARNING messages, the program will be executed. Otherwise, compilation terminates with no attempt to run the program.

Whenever possible, the line number in which the error occurred will be appended to the message in the form: IN LINE *n* DETECTED IN LINE *n* or DETECTED AT END, whichever is pertinent.

Compile messages will be printed during a run if a compile error is detected in a subprogram called by CHAIN or INVOKE. The message is printed before execution of the program.

Run Errors

These errors are detected during program execution and printed as they occur. If the error message is preceded by the word "WARNING:", the run continues. Otherwise, the run terminates. WARNING messages may be suppressed during a run by including the NOWARN parameter in the RUN command (see Section II).

The line number where the error occurred will be appended to most run error messages in the form: IN LINE *n*, where *n* is the line number. If the program is named, this message is followed by IN *programname.*

The WARNING messages for run errors generally are in response to arithmetic errors such as underflow, overflow, division by zero, and so forth. In each of these cases, BASIC/3000 will automatically assign a result. This result is printed as part of the message. For instance, for integer overflow the result is ±32767, for all other overflow the result is ±1E77, for division by zero the result is ±1E77, and for underflow the result is zero.

APPENDIX C
BNF Syntax for BASIC/3000

The Backus-Naur Form (BNF) syntax is used to describe the BASIC/3000 language. BNF notation consists of a number of "productions", each of which has the form:

<entity>	: : = <expression>

where the syntactic entity on the left side is defined by or may be replaced by the syntactic expression on the right side. The expression may be a sequence of syntactic terms or several of these sequences separated by the character "|". When more than one sequence appears, it means that the entity may be replaced by one, and only one, of the sequences of syntactic terms.

The following additions have been made to the standard BNF for simplicity and conciseness:

- Brackets ("[" and "]") surrounding an expression indicate that the expression is optional.
- Braces ("{" and "}") surrounding an expression are used to indicate that the expression is to be considered as a single term. Brackets are also used in this way.
- An ellipsis ("...") following a term indicates that the term may be repeated indefinitely.
- A symbol whose name has the form <something list> has an implied definition of <something>[,<something>]... unless stated otherwise.

<constant>	: : = [<sign>] { <integer> | <fixed> | <float> | <long> }
<sign>	: : = + | -
<unsigned constant>	: : = <integer> | <fixed> | <float> | <complex> | <long>
<integer>	: : = <digit> ...
<digit>	: : = 0| 1| 2| 3| 4| 5| 6| 7| 9
<fixed>	: : = <integer>. | .<integer> | <integer>.<integer>

<float> ::= <numpart> E [<sign>] <integer9

<numpart> ::= <integer> | <fixed>

<complex> ::= (<number part>, <number part>)

<number part> ::= [<sign>] {<integer> | <fixed> | <float>}

<long> ::= <numpart> L [<sign>] <integer>

<variable> ::= <numeric variable> | <string variable>

<numeric variable> ::= <simple variable> | <subscripted variable>

<simple variable> ::= <letter> [<digit>]

<letter> ::= A| B| C| D| E| F| G| H| I| J| K| L| M| N| O| P| Q| R| S| T| U| V| W| X| Y| Z

<subscripted variable> ::= <numeric array id> (<sublist>)

<numeric array id> ::= <letter> [<digit>]

<sublist> ::= <subscript> [,<subscript>]

<subscript> ::= <integer expression>

<integer expression> ::= <numeric expression having an integer value, possibly by conversion from a real, long, or complex value>

<expression> ::= <numeric expression> | <string expression>

<numeric expression> ::= <conjunction> [OR<conjunction>]...

<conjunction> ::= <relation> [AND<relation>]...

<relation> ::= <minmax> [<relational operator><minmax>]... |
<string expression><relational operator><string expression>

<minmax> ::= <sum> [{ MIN | MAX } <sum>]...

<sum> ::= <unary sign> <term> [{ + | - } <unary sign> <term>]...

<term> ::= [NOT <unop>] <factor> [{ * | / | MOD } <unop> <factor>]...

<factor> ::= <primary> [{ ** | ^ } <unop> <primary>]...

<unop> ::= [+ | - | NOT]...

<unary sign> ::= [+ | -]...

<primary> ::= <numeric variable> | <unsigned constant> |
<numeric function reference> | (<numeric expression>)

```
<relational operator>               ::= < | <= | = | <> | > | >= | #

<numeric function reference>        ::= <numeric built-in function name> (<argument list>) |
                                        <numeric user-defined function name> (<actual parameter list>)

<numeric built-in function name>    ::= <name of any BASIC/3000 built-in function that
                                        returns a numeric value>

<argument>                          ::= <numeric expression> | <string expression> |
                                        <numeric array id> | <string array id> |

<actual parameter>                  ::= <expression> | <string simpvar id> (*) |
                                        <numeric array id> { (*) | (*,*) } | <string array id> (*,*)

<string expression>                 ::= <source string> [+<source string>]...

<source string>                     ::= <string variable> | <literal string> | <string function reference>

<string variable>                   ::= <string simple variable> | <string array variable>

<string simple variable>            ::= <string simpvar id> [(<substring designator>)]

<string array variable>             ::= <string array id> (<subscript> [,<substring designator>])

<string simpvar id>                 ::= <letter> [<digit>]$

<string array id>                   ::= <letter> [<digit>]$

<substring designator>              ::= <first character position> [,<last character position>] |
                                        <first character position>;<number of characters>

<first character position>          ::= <integer expression>

<last character position>           ::= <integer expression>

<number of characters>              ::= <integer expression>

<literal string>                    ::= <quoted string> |' <integer> [<quoted string>] |
                                        <literal string> ' <integer> [<quoted string>]

<quoted string>                     ::= "[<character>]..."

<character>                         ::= <any ASCII graphic character other than ">

<string function reference>         ::= <string built-in function name> (<argument list>) |
                                        <string user-defined function name (<actual parameter list>)

<string built-in function name>     ::= <name of any BASIC/3000 built-in function that returns
                                        a string value>

<LET statement>                     ::= [LET] <let part> [,<let part>]...
```

<let part> : : = <num let> | <string let>

<num let> : : = <num left part> <numeric expression>

<num left part> : : = {<numeric variable> = } ...

<string let> : : = <string left part> <string expression>

<string left part> : : = {<destination string> =} ...

<destination string> : : = <string variable>

<REM statement> : : = REM <character string>

<character string> : : = <any ASCII graphic character>

<GO TO Statement> : : = <single-branch GO TO> | <multibranch GO TO>

<single-branch GO TO> : : = GO TO <label>

<multibranch GO TO> : : = GO TO <integer expression> OF <label list>

<label> : : = <integer>

<GOSUB statement> : : = <single-branch GOSUB> | <multi-branch GOSUB>

<single-branch GOSUB> : : = GOSUB <label>

<multi-branch GOSUB> : : = GOSUB <integer expression> OF <label list>

<RETURN statement> : : = <gosub return> | <function return>

<gosub return> : : = RETURN

<function return> : : = RETURN <expression>

<END Statement> : : = END

<STOP Statement> : : = STOP

<FOR statement> : : = FOR <for variable> = <initial value> TO <final value> [STEP <step size>]

<NEXT statement> : : = NEXT <for variable>

<for variable> : : = <simple variable>

<initial value> : : = <numeric expression>

<final value> : : = <numeric expression>

<step size> : : = <numeric expression>

<IF body> ::= <IF part> [<ELSE part>]

<IF part> ::= <IF statement> | <IF DO statement> <DO part>

<ELSE part> ::= <ELSE statement> | <ELSE DO statement> <DO part>

<IF statement> ::= IF <numeric expression> THEN { <label> | <clause> }

<IF DO statement> ::= IF <numeric expression> THEN DO

<ELSE statement> ::= ELSE { <label> | <clause> }

<ELSE DO statement> ::= ELSE DO

<DO part> ::= [<statement>]... <DOEND statement>

<DOEND statement> ::= DOEND

<clause> ::= <any executable statement other than IF, FOR, NEXT, ELSE, or DOEND>

<PRINT Statement> ::= PRINT [<print list> [, | ;]]

<print list> ::= <print element> [{ , | ; } <print element>]...

<print element> ::= <expression> | <print function> | (<FOR statement>, <print list> [, | ;])

<print function> ::= <print function name> (<integer expression>)

<print function name> ::= TAB | LIN | SPA | CTL

<READ statement> ::= READ <read item list>

<read item> ::= <variable> | (<FOR statement>, <read item list>)

<DATA statement> ::= DATA <data constant list>

<data constant> ::= <constant> | <literal string>

<RESTORE statement> ::= RESTORE [<label>]

<INPUT statement> ::= INPUT [[:] <input item list>] [:]

<input item> ::= <variable> | <literal string> | (<FOR statement>, <input item list>)

<ENTER statement> ::= ENTER # <terminal> |
ENTER # <terminal>, <allotment>, <time>, <variable>

<allotment> ::= <integer expression>

<terminal> ::= <numeric variable>

<time> : : = <numeric variable>

<DIM statement> : : = DIM <dimspec list>

<dimspec> : : = <numeric dimspec> | <string dimspec>

<numeric dimspec> : : = <numeric array id> (<bound> [, <bound>])

<string dimspec> : : = <string array id> (<bound>, <size>)| <string simpvar id> (<size>)

<bound> : : = <integer>

<size> : : = <integer>

<REDIM statement> : : = REDIM <redimspec> [,<redimspec>]

<redimspec> : : = <numeric redimspec> | <string redimspec>

<numeric redimspec> : : = <numeric array id>(<integer expression>[,<integer expression>])

<string redimspec> : : = <string array id> (<integer expression>)

<Type statement> : : = <type> <typespec list>

<type> : : = INTEGER | COMPLEX | LONG | REAL

<typespec> : : = <simple variable> | <numeric dimspec>

<MAT READ statement> : : = MAT READ <mat read item list>

<MAT INPUT statement> : : = MAT INPUT <mat read item list>

<mat read item> : : = <numeric array id> | <string array id> | <redimspec>

<MAT PRINT statement> : : = MAT PRINT <mat print list> [, | ;]

<mat print list> : : = <mat print item> [{ , | ; } <mat print item { , | ; }]...

<mat print item> : : = <numeric array id> | <string array id> | <print function>

<MAT initialization statement> : : = MAT <numeric array id>
= <initialization function> [(<integer expression>[,<integer expression>])]

<initialization function> : : = ZER | CON | IDN

<string MAT initialization statement>: : = MAT <string array> = NUL$(<integer expression>)

<MAT assignment statement>	: : = MAT <numeric array id> = <numeric array id> | MAT <numeric array id> = <numeric array id>+<numeric array id>| MAT <numeric array id>=<numeric array id>–<numeric array id>| MAT <numeric array id>=<numeric array id>*<numeric array id>| MAT <numeric array id>=INV(<numeric array id>)| MAT <numeric array id>=TRN(<numeric array id>)| MAT <numeric array id>=(<numeric expression>)*<numeric array id> MAT <string array id>=<string array id>
<CONVERT statement>	: : = CONVERT <numeric expression> TO <destination string>| CONVERT <string expression> TO <numeric variable>[,<label>]
<LINPUT statement>	: : = LINPUT <destination string>
<multiline function>	: : = <multiline DEF statement> <multiline function body>
<multiline DEF statement>	: : = DEF[<type>]<numeric function name>(<formal parameter list>)| DEF <string function name> (<formal parameter list>)
<multiline function body>	: : = <statement> ... <FNEND statement>
<FNEND statement>	: : = FNEND
<formal parameter>	: : = [<type>] <variable parameter> | <string parameter>
<string parameter>	: : = <string simpvar id> [(*)] | string array id> (*,*)
<variable parameter>	: : = <simple variable> | <numeric array id> { (*) | (*,*) }
<numeric function name>	: : = FN <letter>
<string function name>	: : = FN <letter>$
<one-line DEF statement>	: : = DEF[<type>] <numeric function name>(<formal parameter list>) = <numeric expression> | = DEF = string function name>)<formal parameter list> = <string expression>
<CREATE statement>	: : = CREATE <numeric variable>,<string expression>, <file size> [, <record size>]
<filesize>	: : = <integer expression>
<record size>	: : = <integer expression>
<PURGE statement>	: : = PURGE <numeric variable>,<string expression>
<FILES statement>	: : = FILES <file designator list>

```
<file designator>             ::= <qualified file name> | * | #<integer>

<qualified file name>         ::= <local file reference> [.<group name>[.<account name>]]

<local file reference>        ::= <file name> [/<lockword>]

<ASSIGN statement>            ::= ASSIGN <string file name>, <file number>,
                                  <numeric variable> [,<protect mask>] [,<restriction>] |
                              ::= ASSIGN *, <file number>

<string file name>            ::= <string expression>

<file number>                 ::= <integer expression>

<protect mask>                ::= <string expression>

<restriction>                 ::= RR | WR | NR | WL | NL

<file PRINT statement>        ::= PRINT # <file number>[,<record number>] [; <print list> [,| ;]] |
                                 PRINT # <file number>[,<record number>]; [ <print list>{,| ;}] END

<file READ statement>         ::= READ # <file number> [,<record number>] [;<read item list>]

<record number>               ::= <integer expression>

<ON END statement>            ::= { ON | IF } END # <file number> THEN <label>

<ADVANCE statement>           ::= ADVANCE # <file number>;<integer expression>,<numeric variable>

<UPDATE statement>            ::= UPDATE # <file number>;<expression>

<LOCK statement>              ::= LOCK # <file number>

<UNLOCK statement>            ::= UNLOCK # <file number>

<file LINPUT statement>       ::= LINPUT # <file number> [,<record number> ] ; <destination string>

<file RESTORE statement>      ::= RESTORE #<file number>

<file MAT READ statement>     ::= MAT READ #<file number> [,<record number>]
                                  [;<mat read item list>]

<file MAT PRINT statement>    ::= MAT PRINT #<file number>[,<record number>]
                                  [,<mat print list>[,| ; ]] |

<PRINT USING statement>       ::= <print using> [;<print using element list>]

<print using>                 ::= PRINT USING { <string expression> | <label> }

<print using element>         ::= <expression> | <print function> |
                                  (<FOR statement>,<print using element list>)
```

```
<MAT PRINT USING statement>   ::= MAT <print using> [;<mat print item list>]

<IMAGE statement>             ::= IMAGE <format string>

<format string>               ::= [<carriage control>,] <format list>

<carriage control>            ::= + | - | #

<format list>                 ::= [/|,] <format element> [ { / | , } <format element>]... [/|,]

<format element>              ::= <format spec> | <replicator> (<format list>)

<replicator>                  ::= <integer>

<format spec>                 ::= <string spec>|<fixed spec>|<float spec>|
                                  <integer spec>|<complex spec>|<K spec>|<literal spec>

<literal spec>                ::= <lit>

<lit>                         ::= [ <literal string> | [< replicator>] { X | I | $ } ]

<string spec>                 ::= <lit> { [<replicator>] A <lit> } ...

<K spec>                      ::= <lit> K <lit>

<integer spec>                ::= <unsigned integer spec> | <signed integer spec>

<unsigned integer spec>       ::= <lit> { [<replicator> D <lit> } ...

<signed integer spec>         ::= <lit> { S|M } <unsigned integer spec> |
                                  <unsigned integer spec> { S|M }
                                  [<unsigned integer spec> | <lit>]

<fixed spec>                  ::= <signed fixed spec> | <unsigned fixed spec>

<signed fixed spec>           ::= <signed integer spec>. {<unsigned integer spec>|<lit>} |
                                  <lit> { S|M } <lit>.<unsigned integer spec>|
                                  { <unsigned integer spec>|<lit> } . <signed integer spec> |
                                  <unsigned integer spec> . <lit> { S|M } <lit>

<unsigned fixed spec>         ::= <unsigned integer spec> .
                                  { <unsigned integer spec> | <lit> } |
                                  <lit> . <unsigned integer spec>

<float spec>                  ::= <unsigned float spec> | <signed float spec>

<unsigned float spec>         ::= {<unsigned integer spec> | <unsigned fixed spec> } E<lit>

<signed float spec>           ::= {<signed integer spec> | <signed fixed spec> } E<lit> |
                                  <unsigned float spec> { S|M } <lit>
```

<simple spec> ::= <fixed spec> | <float spec> | <integer spec>

<complex spec> ::= <lit> C (<simple spec>, <simple spec>) <lit> |
{<lit> | <simple spec>} {+ | -} {<unsigned integer spec> |
>unsigned fixed spec> | <unsigned float spec>}

<CHAIN statement> ::= CHAIN <string expression> [,<integer expression>]

<INVOKE statement> ::= INVOKE <string expression> [,<integer expression>]

<COM statement> ::= COM [(<nonzero digit>)] <com item list>

<nonzero digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<com item> ::= [<type>] <numeric com item> | <string com item>

<string com item> ::= <string parameter> | <string dimspec>

<numeric com item> ::= <variable parameter> | <typespec>

<CALL statement> ::= {CALL | *}<external procedure name>[(<actual parameter list>)]

<external procedure name> ::= <the name of a procedure in the group SL, account SL,
or system SL>

<SYSTEM statement> ::= SYSTEM <numeric variable>, <string expression>

APPENDIX D
Summary of BASIC/3000 Statements and Commands

STATEMENT SUMMARY

This summary of BASIC/3000 statements provides the statement names in alphabetic order with a brief description and a reference to the section or sections containing a complete statement description.

Statement	Description	Reference
ADVANCE #	Skips the specified number of items in a forward or backward direction on a file.	Section VIII
ASSIGN	Dynamically assigns a file name to a file number and opens the file; may also be used to close files during execution.	Section VIII
CALL or *	Calls for execution of a procedure stored in a segmented procedure library (SL), optionally passing parameters to the procedure.	Section XI
CHAIN	Terminates the current program and calls for execution of the BASIC/3000 program named in the CHAIN statement. Variables are shared between programs if named in COM statements.	Section X
COM	Declares a common block to contain specified variables used in common by more than one program. Effective when one program calls another with CHAIN or INVOKE.	Section X
COMPLEX	Declares the following variable or variables to be type complex.	Section IV
CONVERT	Converts a numeric expression to a string representation, or converts a string expression to a numeric representation.	Section V
CREATE	Creates a formatted file with a specified length and, optionally, a record size.	Section VIII
DATA	Provides data to be read by READ statements.	Section II
DEF	Introduces a function definition.	Section VI

Statement	Description	Reference
DIM	Reserves storage for arrays and sets the upper bounds on the number of elements.	Section III
	DIM also reserves storage for strings and sets their maximum character length.	Section V
DO...DOEND	Used only after IF...THEN or ELSE, they enclose statements to be executed when an IF or ELSE condition is satisfied. (See IF...THEN)	Section II
ELSE	Used only in conjunction with IF...THEN, it introduces a statement to be executed when the IF condition is false. (See IF...THEN)	Section II
END	Terminates execution of the current program; may be omitted since last line of program provides an implicit END.	Section II
ENTER	Provides for user input with a timed response. Returns the actual response time and, optionally, the logical terminal number. One numeric or string constant can be input.	Section II
FILES	Allocates file numbers to file names or reserves file numbers for later assignment with ASSIGN. FILES is declarative and, unlike ASSIGN, is not executed.	Section VIII
FNEND	Terminates a multi-line function definition.	Section VI
FOR...NEXT	Allows repetition of a group of statements between FOR and NEXT. The number of repetitions is determined by the initial and final values of a FOR variable, and by an optional step specification.	Section II
GOTO	Transfers control to a specified statement label.	Section II
GOTO...OF	Multibranch GOTO transfers control to one of a list of statement labels depending on the value of an integer expression.	Section II
GOSUB	Causes execution of a subroutine beginning at a specified statement label. Following a RETURN statement in the subroutine, control returns to the statement following GOSUB.	Section II
GOSUB...OF	Multibranch GOSUB executes one of a list of subroutines depending on the value of an integer expression.	Section II
IF END #	Specifies action to be taken when an end-of-file condition occurs; IF END # is used interchangeably with ON END #.	Section VIII

Statement	Description	Reference
IF...THEN	Evaluates a conditional expression and specifies action to be taken if condition is true. The condition is a numeric expression considered true if its value is nonzero, false if its value is zero. The action may be transfer to a statement label, a single executable statement, or a DO. . . DOEND group.	Section II
IMAGE	Provides format specifications for PRINT USING or MAT PRINT USING statements.	Section IX
INPUT	Requests user input to one or more variables by printing a ? and accepts string or numeric data from the terminal.	Section II
INTEGER	Declares the following variables or arrays to be type integer.	Section IV
INVOKE	Suspends the current program and calls for execution of a BASIC/3000 program, and returns to statement following INVOKE after execution of the invoked program. Variables are saved and files remain open; data may be passed with COM statement.	Section X
LET	Introduces assignment statement that assigns one or more values to a variable or array element. The word LET may be omitted.	Section II
LINPUT	Requests a line of input from the terminal, all of which is assigned to a single string variable.	Section V
LINPUT #	Accepts contents of a record on a data file as input to a string variable. Used only with ASCII files.	Section VIII
LOCK #	Dynamically locks file during execution; all write operations will be completed and no other user can lock that file until an UNLOCK # statement is executed.	Section VIII
LONG	Declares the following variables or arrays to be type long.	Section IV
MAT Add	Performs array addition element by element upon arrays of identical logical size, and assigns result to another array.	Section III
MAT Copy	Copies one array into another array with at least as many elements and the same number of dimensions. Any redimensioning is automatic.	Section III
MAT Initialize	Initializes a numeric array with values specified by the functions ZER (zero), CON (ones), or IDN (identity array).	Section III
MAT INPUT	Inputs values to arrays from the terminal; optionally an array can be redimensioned.	Section III

Statement	Description	Reference
MAT Inverse	Assigns the inverse of a square array to another array using the function INV. Any redimensioning is automatic.	Section III
MAT Multiply	Performs array multiplication on an array with dimensions m by n and an array of dimensions n by p resulting in a new array with dimensions m by p.	Section III
MAT PRINT	Prints arrays by rows according to array dimensions; a semicolon after the array name will pack the rows in a line.	Section III
MAT PRINT #	Prints contents of arrays by rows in a specified file.	Section VIII
MAT PRINT USING	Prints arrays according to format specifications in MAT PRINT USING statement or in an IMAGE statement.	Section IX
MAT READ	Reads data from DATA statements into one or more arrays.	Section III
MAT READ #	Reads data from a file into one or more arrays.	Section VIII
MAT Scalar Multiply	Multiplies each element in an array by a specified numeric expression. Any redimensioning is automatic.	Section III
MAT Subtract	Performs array subtraction element by element upon arrays of identical logical size, and assigns result to another array.	Section III
MAT Transpose	Transposes an n by m array to an m by n array using the function TRN. Any redimensioning is performed automatically.	Section III
NEXT	Terminates a loop introduced by a FOR statement. Specifies a variable that must match the FOR variable.	Section II
ON END #	Specifies action to be taken when an end-of-file condition occurs.	Section VIII
PRINT	Prints the contents of a list of numeric or string expressions on the list device.	Section II
PRINT #	Outputs the contents of a list of numeric or string variables to the specified file.	Section VIII
PRINT USING	Prints the contents of a list of numeric or string variables with format controlled by format specifications included in the PRINT USING statement or in an IMAGE statement.	Section IX
PURGE	Purges a specified file from the system.	Section VIII
READ	Assigns constants and string literals from one or more DATA statements to the variables specified in READ. Treats contents of all DATA statements as a single data list.	Section II

Statement	Description	Reference
READ #	Reads one or more items from a file into specified variables.	Section VIII
REAL	Declares the following variables and arrays to be type real. This type declaration is not generally required because the real representation is the default case.	Section IV
REDIM	Redimensions the rows and columns of an array.	Section III
	Redimensions the size of a string array without changing the element size.	Section V
REM	Introduces remarks and comments in the program listing.	Section II
RESTORE	Resets the data pointer to the beginning of the program or to the first DATA statement following a specified label.	Section II
RESTORE #	Repositions the file pointer to the start of the file; can only be used on files that can be rewound.	Section VIII
RETURN	Returns control from a GOSUB subroutine to the statement following the last GOSUB.	Section II
	Terminates execution of a multiline user-defined function and returns the value of the function.	Section VI
STOP	Terminates execution of the run.	Section II
SYSTEM	Dynamically executes an MPE/3000 command from a BASIC/3000 program.	Section XI
UNLOCK #	Unlocks a file that was locked with LOCK # enabling other programs to lock and/or write on that file.	Section VIII
UPDATE #	Modifies one item in a file without affecting other items.	Section VIII

COMMAND SUMMARY

Each command is listed by name in alphabetical order followed by a brief description and a reference to the section or sections containing a complete description of the command.

Command	Description	Reference
ABORT	Legal only in break period; terminates the suspended program and returns to BASIC/3000 control where all commands are legal.	Section VII
APPEND	Appends a specified program to the end of the current program.	Section II
>BASIC	Interrupts input requested by INPUT or ENTER and enters a new level of BASIC/3000.	Section II
BREAK	Specifies breakpoints where execution of program will be interrupted to enter debugging commands.	Section VII
CALLS	Legal only in break period; lists functions and programs called by INVOKE that have not been completed.	Section VII
CATALOG or CAT	Lists name, type, file size, and record size of programs and files in the specified fileset.	Section II
CREATE	Creates a BASIC/3000 formatted file with a specified length, and optionally, record size.	Section VIII
DELETE or DEL	Deletes one or a range of more than one statement from current program.	Section II
DUMP	Displays the contents of a BASIC/3000 formatted file at the terminal or on a specified ASCII file.	Section VIII
>EOD	Terminates batch input.	Section XII
EXIT	Terminates the current BASIC/3000 program.	Section I
FILES	Legal only in break period; lists all files for the executing program.	Section VII
GET	Gets the specified BASIC/3000 program from the user's library, replacing the current program.	Section II
GO	Legal only in break period; terminates the debugging mode and resumes the suspended program. RESUME may be used wherever GO is used.	Section VII
KEY	Returns from TAPE mode to terminal mode.	Section XII
LENGTH OR LEN	Prints the number of words in the current program.	Section II

Command	Description	Reference
LIST	Lists the contents of the current program at the terminal or on a specified ASCII file.	Section II
NAME	Assigns a name to the current program.	Section II
PUNCH	Punches a program on paper tape and inserts control characters as needed to read the tape.	Section XII
PURGE	Deletes the specified data or program file from the system	Section II Section VIII
RENUMBER or RENUM	Renumbers any group of statements in the current program, optionally from a new first line number with a specified increment. By default, renumbering starts at 10 with increments of 10.	Section II
RESUME	Resumes normal BASIC/3000 operation following a SYSTEM command break, pressing Y^c, or a debugging break.	Section I Section VII Section XI
RUN	Executes the current program or gets and executes a specified program file in a library.	Section II
SAVE	Saves the current program as a program file in a library.	Section II
SCRATCH or SCR	Deletes entire current program and its name. Clears all break points and traces.	Section II
SET	Legal only in break period; sets any program variable to a constant value.	Section VII
SHOW	Legal only in break period; lists the values of the specified items.	Section VII
SPOOL	Reads paper tapes that have not been punched with X-OFFs.	Section XII
SYSTEM	Suspends BASIC/3000 and transfers control to MPE/3000; the RESUME command returns control to BASIC/3000.	Section I Section XI
TAPE	Reads paper tapes that have been punched with X-OFFs.	Section XII
TRACE	Traces variable and array values, and the execution of statements and segmented programs.	Section VII
UNBREAK	Deletes any or all breakpoints specified with the BREAK command.	Section VII
UNTRACE	Deletes tracing specified by TRACE command.	Section VII
XEQ	Inputs commands and program statements from a specified file; the end-of-file terminates XEQ.	Section XII

APPENDIX E
Built-In Functions

A set of built-in (or predefined) functions is available for reference by the BASIC/3000 user. These functions with their class and meaning are listed below in alphabetic order. If usage is described in this manual, a section number follows the description. Built-in functions are separated into eight classes. The function result (numeric type or string) is based on the class of the function and the argument type. The table below shows the type of the result based on the function class and argument type:

Function Class	Type of Argument: INTEGER/ REAL	LONG	COMPLEX	STRING
1	REAL	LONG	REAL	–
2	REAL	LONG	COMPLEX	–
3	COMPLEX	COMPLEX	COMPLEX	–
4	REAL	REAL	REAL	–
5	STRING	STRING	STRING	–
6	–	–	–	REAL
7	–	–	–	STRING
8	argument is an array or string array, result is real			

Note that an argument for a trigonometric function must be expressed in radians with 1 radian equal to $\frac{180}{\pi}$ or 57.1958 degrees.

A variable argument is shown by a capital letter, an expression by a lower-case letter.

Name and Parameters	Class	Meaning
ABS(x)	1	Absolute value of x: when x is complex: ABS(x) = SQR(REA (x)**2+IMG (x)**2)
ATN(x)	1	Arctangent x; when x is complex, the result is the angular argument of x, or $\tan^{-1}$(IMG(x)/REA(x)) adjusted to the appropriate quadrant. x is expressed in radians.
BRK(x)	4	Allows programmatic control of breaks; use with caution. If $x < 0$, returns current setting only. If x = 0, >BASIC, the break key, and Y^C break are disabled. If $x > 0$, these functions are enabled. BRK returns 0 if traps were previously disabled or 1 if they were enabled.
BUF(x)	4	Test input buffer for : option of INPUT. For this function, x is a dummy parameter. (Section II)
CEI(x)	1	Ceiling of x; smallest integer > = x. When x is complex, only the real part is used.
CHR$(x)	5	Generates a one-character ASCII string; x is in the range 0-255. (Section V).
CNJ(x)	3	Complex conjugate of x; that is, it reverses the sign of the imaginary part of x. (Section IV).
COL(A)	8	Number of columns in array A. If A is one-dimensional, COL(A)=1. (Section III).
COS(x)	2	Cosine of x; x must be expressed in radians.
CPX(x,y)	3	Complex number = x + yi. If x or y is complex only the real part is used. (Section IV).
CPU(x)	4	Number of seconds of CPU time (± .001 sec.) that the program has run.
CSH(x)	2	Hyperbolic cosine of x; CSH(x) is $(e^x + e^{-x})/2$.
DAT$(x,y)	5	Generates date string. x,y selects substring: String Position: 1-3, 6-11, 14-17, 20-27 Contents: Day, Date, Year, Time Time is expressed as hours 0-12, minutes 0-59. (Section V).
DEB$(s)	7	Returns s with leading and trailing blanks removed. (Section V).
EXP(x)	2	e^x
IMG(x)	4	Imaginary part of x. (Section IV).

Name and Parameters	Class	Meaning
INT(x)	1	Largest integer $<=$ x. If x is complex, only the real part is used.
LEN(s)	6	Logical length of string expression s. (Section V).
LOG(x)	2	Natural logarithm ($\log_e x$). If x is complex, it must not be zero. If x is not complex, it must be greater than zero.
NUM(s)	6	ASCII code for first character of string expression s. (Section V).
PIX(x)	2	PI function = $\pi * x$
POS(s_1,s_2)	6	Smallest integer representing starting position in s_1 of substring identical to s_2. If no such substring, then equals zero. (Section V).
REA(x)	4	Real part of x. (Section IV).
REC(x)	4	Current record number of file x. (Section VIII).
ROW(A)	8	Number of rows in array A. If A is one-dimensional, it returns the dimension. (Section III).
RND(x)	4	Pseudo-random number between 0 and 1 but not equal 1. If x $>=$ 0, the number is determined from the previous random number, except on the first call when an unpredictable (totally random) number is generated. If x $<$ 0, the random number is determined by x. To generate a repeatable sequence of random numbers make the first call with x $<$ 0, and subsequent calls with x $>=$ 0. To repeat the sequence, use the value of x from the first call. To generate a non-repeatable sequence, use x $>=$ 0 for all calls, including the first.
SGN(x)	4	Sign function; equals 1 for x $>$ 0, 0 for x = 0, and -1 for x $<$ 0.
SIN(x)	2	Sine x; x must be expressed in radians.
SNH(x)	2	Hyperbolic sine x; SNH(x) is $(e^x - e^{-x})/2$.
SQR(x)	2	Square root of x; x must be $>=$ 0.
TAN(x)	2	Tangent x; x must be expressed in radians.
TNH(x)	2	Hyperbolic tangent x; TNH(x) is SNH(x)/CSH(x).
TIM(x)	4	Time, where the value is determined by x: if x $<$ 0, number of seconds since program began x = 0, current minute (0-59) x = 1, current hour (0-23) x = 2, current day (1-366) x $>=$ 3, current year (0-99)

Name and Parameter	Class	Meaning
TYP(x)	4	Returns type of next data item in file $\|x\|$, or in DATA list if x = 0. (Section VIII).
UND(X)	4	X must be a numeric variable, UND(X) returns 1 if X has undefined value, 0 otherwise.
UPS$(s)	7	Upshift alphabetic lower case to upper case in string expression s. (Section V).
WRD(s_1,s_2)	6	Smallest integer representing starting position in s_1 of a substring that is surrounded by non-alphabetic characters and is identical to s_2. If there is no such substring, 0 is returned. (Section V).

APPENDIX F
Parameter Format

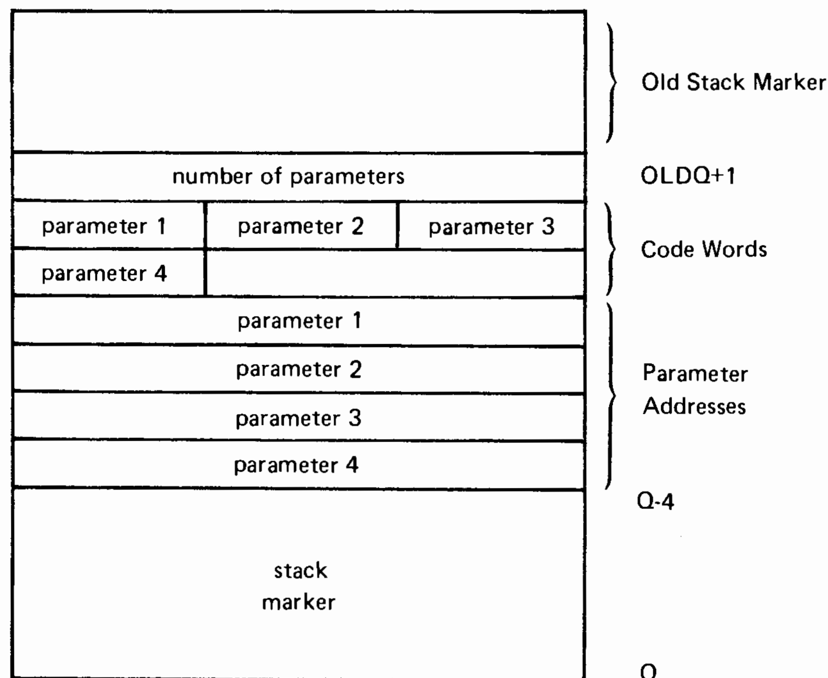
When parameters are specified in the CALL statement, the BASIC/3000 Interpreter sets up a table of the parameter addresses with a pointer to the first address. The parameter addresses are preceded by a code word for each parameter to specify the data type and whether the parameter is simple numeric, string, or an array. This enables the procedure to check if the calling sequence is correct.

The addresses point to the parameter values. These values are stored differently depending on the type of the parameter.

The user who writes the SPL or FORTRAN procedures that he calls from BASIC needs to know the format of the parameter table and also how the values are stored.

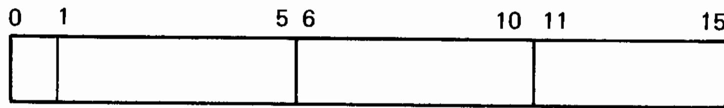
PARAMETER ADDRESS TABLE

This sample table is in the HP 3000 data stack. It contains the number of parameters, a code word for each parameter, and the parameter addresses:

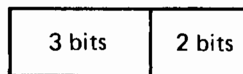


The user should refer to the HP 3000 Computer System Reference Manual for details on stack operation.

Each code word has three fields of five bits each:



Each field has two subfields of three and two bits each:



The three-bit field gives the data type of the parameter:

0 - string

1 - integer

2 - real

3 - long

4 - complex

The two-bit field specifies:

0 - simple numeric

1 - simple string or one-dimensional numeric array

2 - two-dimensional numeric array or one-dimensional string array

The code words in the stack following the procedure call are in the same order as the parameter addresses that follow.

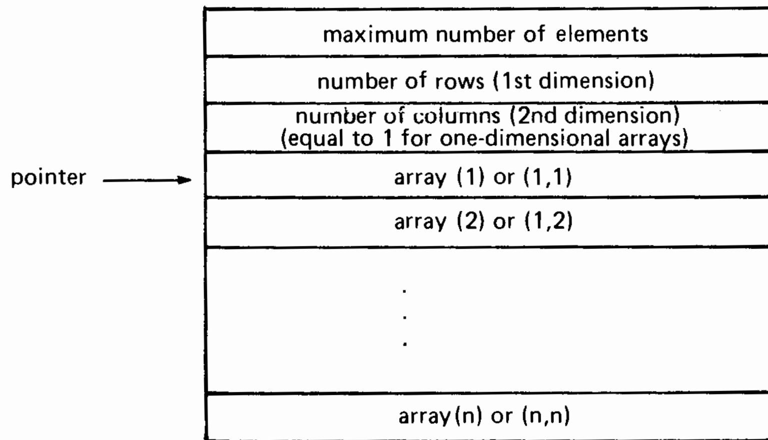
PARAMETER STORAGE

What the parameter address points to depends on the type of the parameter; whether it is simple numeric, numeric array, simple string, or string array:

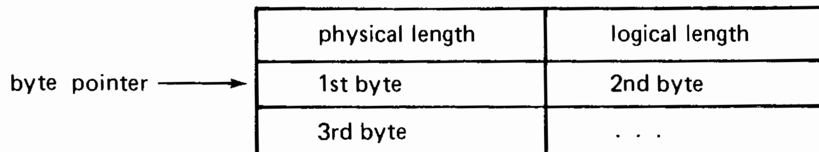
1. For a simple numeric expression, including simple variables and subscripted variables, the address points to the first word of the value. The number of words needed for a value depends on the data type:

integer	-	1 word
real	-	2 words
long	-	4 words
complex	-	4 words

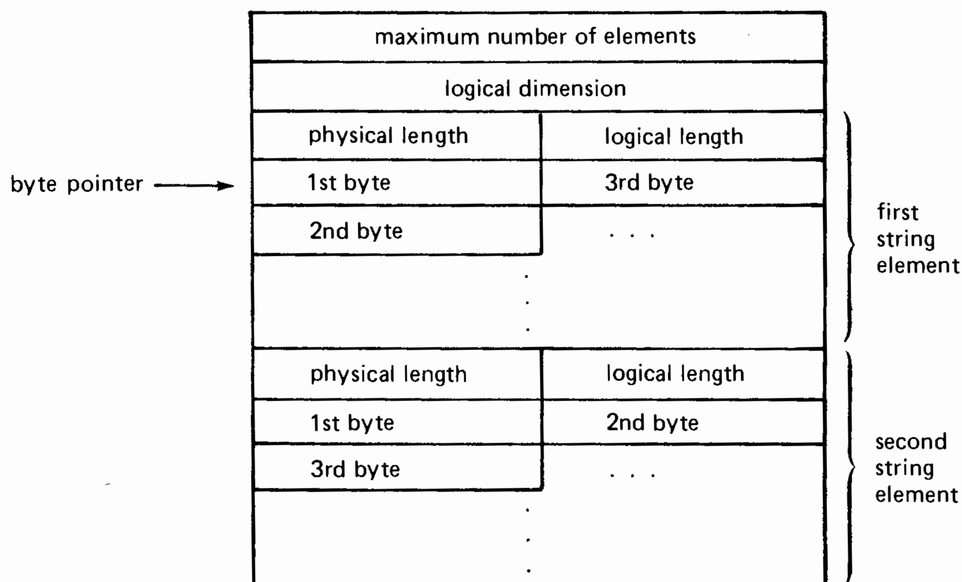
2. For numeric arrays, the address points to the first value of the array (array(1) or array(1,1)). There are three words prior to this value that describe the array. Arrays are stored by rows as follows:



3. For simple string variables, string array elements, and string expressions, the address is a byte pointer that points to the first byte of the string. The two preceding bytes define the physical and logical lengths of the string:



4. For string arrays, the address points to the first element of the array. Each element has the form of a string value with two bytes specifying physical and logical length respectively, followed by the bytes containing the actual value. This string value is preceded by two words that define the array:



APPENDIX G
Compatibility Between BASIC/2000 and BASIC/3000

With four exceptions, BASIC/2000 is a compatible subset of BASIC/3000. This means that a BASIC/2000 program can be run under control of the BASIC/3000 Interpreter and will compile and execute correctly. But, due to the many new features available in BASIC/3000, a BASIC/3000 program will not necessarily run on a BASIC/2000 system.

The four exceptions to compatibility are described here. None of these exceptions will affect compilation, but they might affect the result when a BASIC/2000 program is run on BASIC/3000.

The exceptions are:

	BASIC/3000	BASIC/2000
1.	A COM statement is valid during one run only. It does not remain valid between runs. COM blocks must have compatible structure.	A COM statement remains valid between runs. COM blocks need not have compatible structure.
2.	Files are closed when a program calls a program with the CHAIN statement.	Files remain open when a program calls a program with the CHAIN statement.
3.	MAT PRINT prints a one-dimensional array as a row of elements, thereby saving space and printing time.	MAT PRINT prints a one-dimensional array as a column of elements.
4.	S or M is required in a floating point specification of a format string if the number is negative.	S or M is not required.

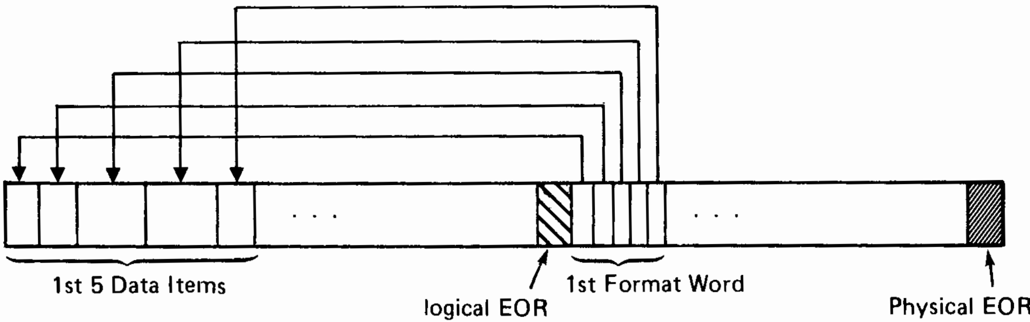
APPENDIX H
File Structure

BASIC/3000 FORMATTED FILES

A formatted BASIC/3000 file contains format words provided by the Interpreter to indicate the type of the data items in the file. Space for these format words is allocated automatically in addition to the record size specified by the user. The format words are placed in each record following the logical end-of-record, but before the physical end-of-record.

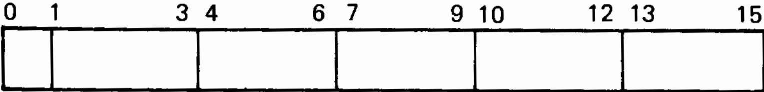
Formatted files can have a record size between 4 and 319 words. The recommended (and default) record size is 106 words per record since this yields 128 words when the format words are added. The standard system size for records is 128 words. Records are numbered starting with 1, not 0.

Each record consists of an area for data items and an area for format words:



Format Word

Each format word consists of five 3-bit flags. The first bit is not used.



The first format word corresponds to the first five data items, with the first flag in the format word corresponding to the first data item, the second flag to the second item, and so forth.

The item types are specified in the format word flags as:

0 - end-of-file

1 - end-of-record

2 - string

3 - integer

4 - real

5 - long

6 - complex

The logical end-of-record delimits the record size specified by the user to include all the data items. The physical end-of-record delimits the BASIC-created record size that is sufficient to contain the format words as well as the data items.

Record Size

The space requirements for a data item differs depending on the data type. The number of 16-bit words required for each data type is:

Data Type	Number of Words
Integer	1
Real	2
Long	3
Complex	4
String	(length +1)/2 + 1

In each case, an additional 1/5 of a format word is added for each data item to provide room for the format words.

The user can determine the physical record size created for the file by BASIC from the logical record size he has used to contain his data items. The formula is:

P = R + INT(R/5) + 1

where P is the physical record size created by BASIC
R is the logical record size assigned by the user

If the physical record size is known, the user can determine the logical record size of a record with another formula:

R = CEI(5*(P-1)/6)

File Attributes

The user may need to know the MPE/3000 file codes for BASIC files. These codes differ depending on how the file was saved and whether it is a program file or a BASIC file. The file code for any file can be requested with the MPE command FGETINFO.

Program File (SAVE)	1026
Program File (FAST SAVE)	1027
BASIC Formatted File	1025

Other file attributes may be obtained with the MPE command :LISTF *filename*,2

The number 2 is a code that provides detailed file information for each file listed. The *filename* may be fully qualified with the user's lockword, group, and account names.

ASCII FILES

ASCII files contain data in ASCII character code. Each 16-bit word contains two characters.

BINARY FILES

Binary files have no format words or string headers. The number of words needed for each data item depends on the type of the item, as follows:

Data Type	Number of Words
Integer	1
Real	2
Long	4
Complex	4
String	(length + 1)/2

INDEX

A

A, in formatted output: 9-7, 9-8
ABORT command: 7-11
ADVANCE syntax: C-8
ADVANCE #, formatted files: 8-29
ALL, CATALOG: 2-62
AND: 2-7
APPEND command: 2-62
arithmetic operator: 2-6
array addition: 3-13
array copying: 3-10
array function: 3-20
array initialization: 3-10
array inversion: 3-16
array multiplication: 3-14
array redimensioning: 3-4
array scalar multiplication: 3-19
array size: 3-2
array substraction: 3-13
array transposition: 3-18
arrays: 3-1
arrays, direct file print: 8-36
arrays, direct file read: 8-37
arrays, formatted print: 9-4
arrays, numeric: 4-10
arrays, serial file print: 8-35
arrays, serial file read: 8-35
ASCII characters: A-1
ASCII file access: 8-22
ASCII file input: 8-23, 12-9
ASCII file read: 8-23
ASCII file structure: H-3
ASCII files: 8-1
ASSIGN statement: 8-9
ASSIGN syntax: C-8
assignment statement: 2-11

B

>BASIC: 2-49
:BASIC command: 1-4, 12-2
BASIC formatted files: 8-1
BASIC program: 1-10
BASIC/2000 compatibility: G-1
batch processing: 12-2
binary file access: 8-24
binary file structure: H-3
binary files: 8-2
binary operator: 2-6
BNF syntax: C-1
Boolean operator: 2-7
BREAK: 1-2
BREAK command: 7-7
breakpoint commands: 7-8
BUF function: 2-45
buffering input: 2-42
built-in functions: E-1
:BYE: 1-5

C

C, formatted output: 9-7, 9-11
CALL statement: 11-2
calling FORTRAN subprogram: 11-3
calling SPL procedure: 11-6
CALLS command, during break: 7-20
card reader control: 12-2
carriage control characters: 9-14
carriage control function: 2-37
carriage return: 1-2
CATALOG command: 2-62
CHAIN statement: 10-2
CHAIN syntax: C-10
changing statements: 1-9
character set: A-1
CHRS function: 5-12
class of functions: 4-11, E-1
closing files: 8-6
COL function: 3-20
COL function, string arrays: 5-14
columns: 3-1
COM statement: 10-9
COM syntax: C-10
command errors: B-1

command summary: D-6
commands: 1-7
commands illegal during break: 7-8
commands legal during break: 7-8
common blocks: 10-9
comparing strings: 5-16
compile errors: B-1
complex form: 4-4
complex formatted output: 9-11
COMPLEX statement: 4-2
compressed formats: 9-12
CON function: 3-10
concatenation: 2-8, 5-9
conditional statements: 2-25
constant,: 2-2
constant, numeric: 2-2
constant, string: 2-4
continuation lines: 1-8
conversion of data: 4-8
CONVERT statement: 5-23
CONVERT syntax: C-7
correcting errors: 1-6
CREATE command: 8-3
CREATE statement: 8-3
CREATE syntax: C-7
CTL function: 2-37
CTRL: 1-2
CTRL H: 1-2
CTRL X: 1-2
CTRL Y: 1-2

D

D, formatted output: 9-7, 9-9
data representation: 4-1
DATA statement: 2-39, 5-17
DATA syntax: C-5
DAT$ function: 5-14
debugging commands: 7-1
DEB$ function: 5-13
decimal, formatted output: 9-7, 9-9
deck structure: 12-3
DEF statement: 6-2, 6-4
DEF syntax: C-7
DELETE command: 2-55
deleting files: 8-5
deleting programs: 1-14
deleting statements: 1-9
diagnostics: B-1
DIM statement: 3-3
DIM, strings: 5-3
DIM syntax: C-6
direct file access: 8-12
direct file, MAT READ statement: 8-37
direct file MAT PRINT statement: 8-36
direct file PRINT statement: 8-18
direct file READ statement: 8-20
displaying formatted files: 8-31
DO . . . DOEND group: 2-25
DUMP command: 8-31

E

E, formatted output: 9-7, 9-9
editing commands: 2-54
editing statements: 1-9
editing symbols: 9-8
ELSE statement: 2-25
ELSE syntax: C-5
END, segmented programs: 10-4
END statement: 2-19
end-of-file condition: 8-27
end-of-file, direct files: 8-18
end-of-file, serial file: 8-13
end-of-record, direct files: 8-18
end-of-record, serial file: 8-13
ENTER statement: 2-47
ENTER statement, strings: 5-18
ENTER syntax: C-5
entering BASIC: 1-4
>EOD command: 12-3
error messages: 1-8, B-1
execution errors: B-2
>EXIT: 1-5
expressions: 2-2
expressions, evaluation of: 2-8

F

false value: 2-7
FAST, SAVE: 2-60
fastsaved program: 2-60
file access: 8-12
file access, ASCII: 8-22
file access, binary: 8-24
file ADVANCE statement: 8-29
file codes: H-3
file dump: 8-31
file functions: 8-32
file length: 8-3
file LINPUT statement: 8-23
file LINPUT syntax: C-8
file MAT PRINT syntax: C-8
file MAT READ syntax: C-8
file name: 8-2
file numbers: 8-6
file numbers, segmented programs: 10-7
file print, direct files: 8-18
file print, serial files: 8-13
file PRINT syntax: C-8
file read, direct files: 8-20
file read, serial files: 8-15
file READ syntax: C-8
file RESTORE statement: 8-17
file RESTORE syntax: C-8
file UPDATE statement: 8-30
file UPDATE syntax: C-8
files: 8-1
FILES command, during break: 7-18
files, dynamic locking: 8-25
FILES statement: 8-7

FILES syntax: C-7
fileset: 2-62
fixed-point form: 4-3
fixed-point formatted output: 9-10
fixed-point number: 2-2
floating-point form: 4-3
floating-point formatted output: 9-10
floating-point number: 2-3
FNEND statement: 6-4
FOR loop, input item: 2-42
FOR loop, print item: 2-31
FOR statement: 2-22
FOR syntax: C-4
format strings: 9-7
format symbols: 9-7
formatted file creation: 8-3
formatted file structure: H-1
formatted files: 8-1
formatted printing: 9-1
FORTRAN subprograms: 11-2
FREQ, RUN: 2-51
function: 2-5
function call: 6-7
function class: 4-11, E-1
function definition, multiline: 6-4
function definition, one-line: 6-2
functions, built-in: E-1

G

GET command: 2-61
GO command: 7-12
GOSUB statement: 2-16
GOSUB syntax: C-4
GOTO statement: 2-13
GOTO syntax: C-4
grouping, formatted output: 9-13

H

:HELLO; 1-4

I

I, formatted output: 9-7, 9-8, 9-11
IDN function: 3-10
IF END #, files: 8-27
IF syntax: C-4
IF . . . THEN statement: 2-25
IMAGE statement: 9-6
IMAGE syntax: C-9
input data: 2-39
input interrupt: 2-49
INPUT statement: 2-42
INPUT statement, strings: 5-18
INPUT syntax: C-5
integer: 2-2
integer expression: 2-2
integer form: 4-3
integer formatted output: 9-10
INTEGER statement: 4-2
internal file numbers: 10-7
interprogram transfer: 10-1
INVOKE statement: 10-4
INVOKE syntax: C-10

K

K, formatted output: 9-7, 9-12
KEY command: 12-7
keys, special: 1-2

L

leaving BASIC: 1-5
LEN function: 5-12
LENGTH command: 2-56
LET statement: 2-11
LET statement, strings: 5-10
LET syntax: C-3
library commands: 2-59
LIN function: 2-36
linefeed: 1-2
line-printer control: 12-2
LINPUT statement: 5-21
LINPUT syntax: C-7
LINPUT #,ASCII files: 8-23
LIST command: 2-54
listing a program: 1-12
literal formatted output: 9-8
literal string: 2-4
literal string, formatted output: 9-8
local file numbers: 10-7
LOCK syntax: C-8
LOCK #, files: 8-25
locking files: 8-25
logging off: 1-5
logging on: 1-4
logical operator: 2-7
long form: 4-4
LONG statement: 4-2
loops: 2-22

M

M, formatted output: 9-7, 9-9
magnitude: 2-3
MAT Add statement: 3-13
MAT Assignment syntax: C-7
MAT Copy statement: 3-12
MAT Initialization syntax: C-6
MAT INPUT statement: 3-6
MAT INPUT syntax: C-6
MAT Inverse statement: 3-16

MAT Multiply statement: 3-14
MAT PRINT statement: 3-8
MAT PRINT syntax: C-6
MAT PRINT USING statement: 9-4
MAT PRINT USING syntax: C-9
MAT PRINT #, direct files: 8-36
MAT PRINT #, serial files: 8-35
MAT READ statement: 3-6
MAT READ syntax: C-6
MAT READ #, direct files: 8-37
MAT READ #, serial files: 8-35
MAT Scalar Multiply statement: 3-19
MAT Substract statement: 3-13
MAT Transpose statement: 3-18
matrix (see arrays, MAT statements): 3-1
MAX: 2-7
MIN: 2-7
mixed-mode arithmetic: 4-7
MOD: 2-6
modifying formatted files: 8-30
MPE/3000 interface: 11-10
multiline function: 6-4

N

NAME command: 2-59
NEXT statement: 2-22
NEXT syntax: C-4
NOECHO, RUN: 2-51
non-BASIC programs: 11-1
non-interactive programming: 12-1
nonprinting characters: 5-1, A-1
NOT: 2-7
NOWARN, RUN: 2-51
NUL$ function: 5-22
NUM function: 5-12
numeric assignment: 4-9
numeric constants: 4-2
numeric expressions: 4-7
numeric formatted output: 9-9
numeric to string conversion: 5-23

O

ON END syntax: C-8
ON END #, files: 8-27
one-dimensional array: 3-1
one-line function definition: 6-2
opening files: 8-6
operator hierarchy: 2-8
operators: 2-6
OR: 2-7
order of execution: 1-7
OUT=, CATALOG: 2-63
OUT=, DUMP: 8-31
OUT=, LIST: 2-54
OUT=, RUN: 2-51
output formats: 2-33
output, formatted: 9-1

P

paper tape control: 12-5
paper tape read: 12-7
parameter format: F-1
parameters, actual: 6-7
parameters, formal: 6-2
parameters, passing: 6-11
passing data, segmented programs: 10-9
passing parameters: 6-10
password: 1-4
POS function: 5-13
print formats: 2-33
print functions: 2-36
print list: 2-31
PRINT statement: 2-31
PRINT statement, strings: 5-20
PRINT syntax: C-5
PRINT USING statement: 9-2
PRINT USING syntax: C-8
PRINT #, direct files: 8-18
PRINT #, serial file: 8-13
printing complex numbers: 4-6
printing long numbers: 4-6
PROG, BREAK: 7-7
PROG, TRACE: 7-2
program: 1-10
program execution: 1-13
program termination: 2-19
pseudocompile: 2-60
PUNCH command: 12-5
punching paper tape, off-line: 12-6
punching paper tape, PUNCH: 12-5
PURGE command: 8-5, 2-61
PURGE statement: 8-5
PURGE syntax: C-7

Q

quoted strings: 5-1

R

READ statement: 2-39
READ statement, strings: 5-17
READ syntax: C-5
READ #, direct files: 8-20
READ #, serial file: 8-15
reading paper tape: 12-7
REAL statement: 4-2
REC function: 8-34
record size: 8-3, H-2
RECSIZE, CATALOG: 2-62
RECSIZE, LIST: 2-54
REDIM statement, arrays: 3-4
REDIM, strings: 5-5
REDIM syntax: C-6
relational operator: 2-7

relational value: 2-7
REM statement: 2-13
REM syntax: C-4
remarks: 2-13
RENUMBER command: 2-56
replicator, formatted output: 9-8
RESTORE data statement: 2-39
RESTORE statement,strings: 5-17
RESTORE syntax: C-5
RESTORE # statement: 8-17
RESUME command: 7-12, 11-10
return: 1-2
RETURN, function: 6-4
RETURN, subroutine: 2-16
rewind files: 8-17
ROW function: 3-20
ROW function, string arrays: 5-14
rows: 3-1
RUN command: 2-51
run errors: B-2
running a program: 1-13
RUNONLY, SAVE: 2-60

S

S, formatted output: 9-7, 9-9
SAVE command: 2-59
SCRATCH command: 2-55
scratching a program: 1-14
segmented libraries: 11-2
segmenting programs: 10-1
separators, formatted output: 9-12
serial file access: 8-12
serial file MAT PRINT statement: 8-35
serial file MAT READ statement: 8-35
serial file PRINT statement: 8-13
serial file READ statement: 8-15
SET command: 7-16
sharing files: 8-25
SHOW command: 7-14
skipping items in file: 8-29
SL: 11-2
SPA function: 2-36
SPL procedures: 11-2
SPOOL command: 12-8
START=, CATALOG: 2-63
statement label (see statement number)
statement number: 1-7
statement summary: D-1
statements: 1-7
STOP, segmented programs: 10-4
STOP statement: 2-19
stopping listing: 2-63
stopping output: 2-63
string array: 5-6
string array initialization: 5-22
string array operations: 5-22
string assignment: 5-10
string comparison: 5-16
string constants: 5-1
string expressions: 5-9
string formatted output: 9-8
string functions: 5-12
string literals: 5-1
string MAT Initialize statement: 5-22
string to numeric conversion: 5-23
string size: 5-6
subscripts: 3-1
substring designator: 5-6
substrings: 5-7
suspending BASIC: 1-5
syntax: C-1
syntax errors: B-1
SYSTEM command: 11-10
SYSTEM statement: 11-12
SYSTEM syntax: C-10
:SYSTEM command: 1-5

T

TAB function: 2-36
TAPE command: 12-7
terminating a program: 2-19
timed input: 2-47
TRACE command: 7-2
true value: 2-7
two-dimensional array: 3-1
TYP function: 8-32
type conversion: 4-8
Type statements: 4-2
Type syntax: C-6

U

unary operator: 2-6
UNBREAK command: 7-7
UNLOCK syntax: C-8
UNLOCK#, files: 8-25
UNTRACE command: 7-2
UPDATE syntax: C-8
UPDATE #, formatted files: 8-30
UPS$ function: 5-13
user-defined functions: 6-1
user's library: 2-59
user's work area: 1-11

V

variable: 2-4
variable types: 4-1

W

work area: 1-11
WRD function: 5-13
write direct file: 8-18
write serial file: 8-13

X

X, formatted output: 9-7, 9-8
XEQ command: 12-9

Z

ZER function: 3-10

Special Characters

&: 1-8
$, formatted output: 9-7, 9-8

Sales and service from 172 offices in 65 countries.
5303 Stevens Creek Blvd., Santa Clara, California 95050

Part No. 30000-90026
Printed in U.S.A. 6/76