Language
Manual

B 1000 Systems
Interactive
BASIC (IBASIC)

(Relative to Mark 11.0 System Software Release)

Burroughs

Language Manual

B 1000 Systems Interactive BASIC (IBASIC)

1152105

LIST OF EFFECTIVE PAGES

Page	Issue	Page	Issue
Title	Original	9-1 thru 9-33	Original
ii	Original	9-34	Blank
iii	Original	10-1 thru 10-2	Original
iv	Blank	11-1 thru 11-27	Original
v thru xiv	Original	11-28	Blank
1-1 thru 1-5	Original	12-1 thru 12-3	Original
1-6	Blank	12-4	Blank
2-1 thru 2-9	Original	A-1 thru A-11	Original
2-10	Blank	A-12	Blank
3-1 thru 3-3	Original	B-1 thru B-3	Original
3-4	Blank	B-4	Blank
4-1 thru 4-12	Original	C-1 thru C-5	Original
5-1 thru 5-9	Original	C-6	Blank
5-10	Blank	D-1 thru D-24	Original
6-1 thru 6-14	Original	E-1 thru E-9	Original
7-1 thru 7-12	Original	E-10	Blank
8-1 thru 8-5	Original	F-1 thru F-5	Original
8-6	Blank	F-6	Blank
		1 thru 10	Original

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS

Section	**Title**	**Page**

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

LIST OF ILLUSTRATIONS

LIST OF TABLES

SECTION 1
INTRODUCTION

PURPOSE OF MANUAL

The purpose of this manual is to provide a description of the Interactive BASIC System (IBASIC) as implemented on the Burroughs B 1000 systems. The name BASIC is an acronym for Beginners All-purpose Symbolic Instruction Code. BASIC was initially developed at Dartmouth College in New Hampshire. The American National Standards Institute (ANSI) developed a standard for BASIC using the original Dartmouth BASIC plus additional features. Burroughs Corporation has implemented the minimal BASIC language and significant extensions to BASIC according to the standard developed by ANSI. In this manual, the term IBASIC refers to the entire interactive system, that is, all of the programs and files necessary to run IBASIC, while the term BASIC refers only to the BASIC language.

The BASIC language is designed for use not only by individuals who have little previous knowledge of computers but also by individuals with considerable programming experience. BASIC can be used in educational, engineering, and scientific environments. A distinct advantage of BASIC is that the rules of form and grammar are easily learned.

Burroughs IBASIC is implemented in a conversational mode: the user enters BASIC statements and interactive commands through a terminal to the IBASIC system, whereupon IBASIC processes the input and responds with (1) the output for the command, (2) a request for more input, or (3) a message informing the user of any syntax errors. Burroughs IBASIC is especially suited for the learning process because response is nearly immediate for many of the errors commonly made by novice programmers.

ORGANIZATION OF MANUAL

The organization and writing of this manual was influenced by two fundamental considerations: (1) that many of the users of Burroughs IBASIC would be using the BASIC language and, possibly, a computer for the first time, and (2) that the user base would also include experienced BASIC users who would need a reference manual only to describe the syntax of a particular command or statement, to learn the commands that comprise the interactive portion of Burroughs IBASIC, or to learn about a command or statement that the programmer never had the opportunity to use before.

The organization of this manual is designed to facilitate learning and using IBASIC.

Section 1 is the introduction to the manual and should be read at least once by all users.

Section 2, entitled Beginning IBASIC, introduces the use of IBASIC.

Sections 3 through 10 provide detailed information about BASIC language syntax.

Sections 11 and 12 describe the IBASIC system commands and capabilities.

Appendices A through F contain the glossary, information on how to set up IBASIC, and other technical information that is generally beyond the interest of the average IBASIC user.

When statements, commands, and syntax are described, the following format is used.

    1. The name of the item being described.
    2. A brief functional description.
    3. The railroad syntax and any necessary verbal description of that syntax.

4. The semantics, which may be omitted depending on the complexity of the item. If this section is omitted, the semantic description of the item is the brief functional description that follows the item name.
5. Examples and explanations of the examples, if necessary.

SYNTAX CONVENTIONS (RAILROAD DIAGRAMS)

Railroad diagrams show how syntactically valid statements can be constructed.

Traversing a railroad diagram from left to right, or in the direction of the arrow heads, and adhering to the limits illustrated by bridges will produce a syntactically valid statement. Continuation from one line of a diagram to another is represented by a right arrow (→) appearing at the end of the current line and beginning of the next line. The complete syntax diagram is terminated by a vertical bar (|).

Items contained in broken brackets (< >) are syntactic variables which are further defined, or require the user to supply the requested information. Upper-case items must appear literally. Minimum abbreviations of upper-case items are underlined.
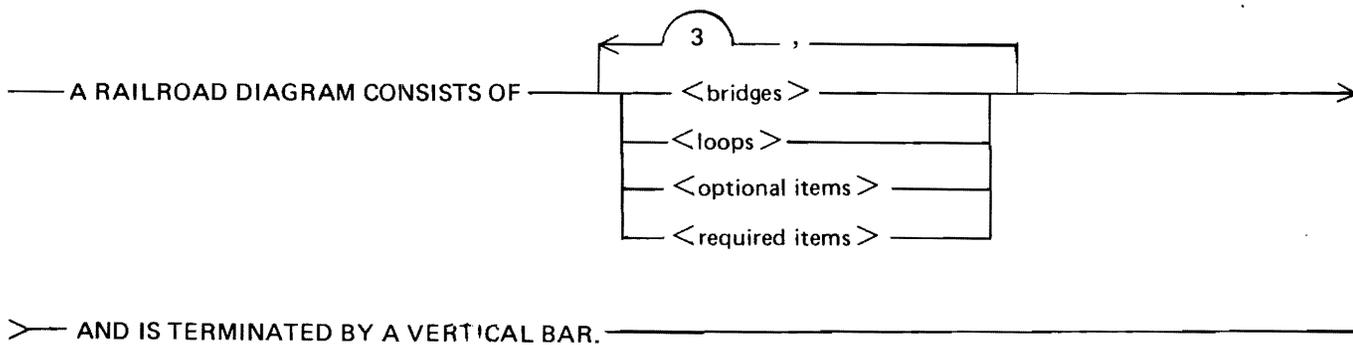
G50051

The following syntactically valid statements may be constructed from the above diagram:

A RAILROAD DIAGRAM CONSISTS OF <bridges> AND IS TERMINATED BY A VERTICAL BAR.

A RAILROAD DIAGRAM CONSISTS OF <optional-items> AND IS TERMINATED BY A VERTICAL BAR.

A RAILROAD DIAGRAM CONSISTS OF <bridges>, <loops> AND IS TERMINATED BY A VERTICAL BAR.

A RAILROAD DIAGRAM CONSISTS OF <optional-items>, <required-items>, <bridges>, <loops> AND IS TERMINATED BY A VERTICAL BAR.

Required Items

No alternate path through the railroad diagram exists for required items or required punctuation.

E. ample:

```
—— REQUIRED ITEM —————————————————————————————————————————————————|
```
G50052
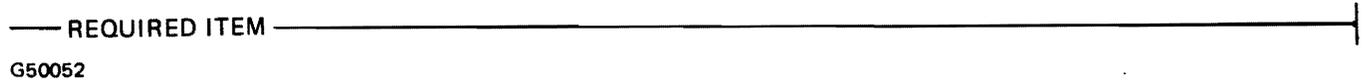
Optional Items

Items shown as a vertical list indicate that the user must make a choice of the items specified. An empty path through the list allows the optional item to be absent.

Example:

```
—— REQUIRED ITEM ——T——————————————————————————————————————————————|
                    ├—— <optional item-1> ——┤
                    └—— <optional item-2> ——┘
```
G50053
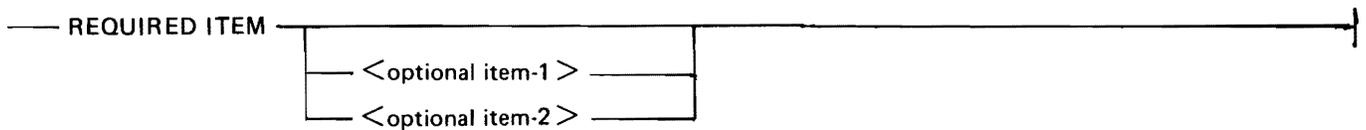
The following valid statements may be constructed from the above diagram:

REQUIRED ITEM

REQUIRED ITEM <optional-item-1>

REQUIRED ITEM <optional-item-2>

Loops

A loop is a recurrent path through a railroad diagram and has the following general format:
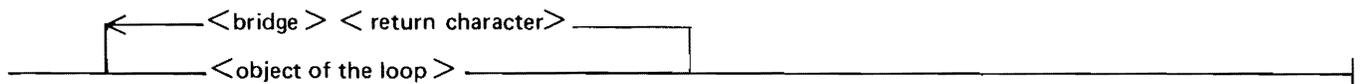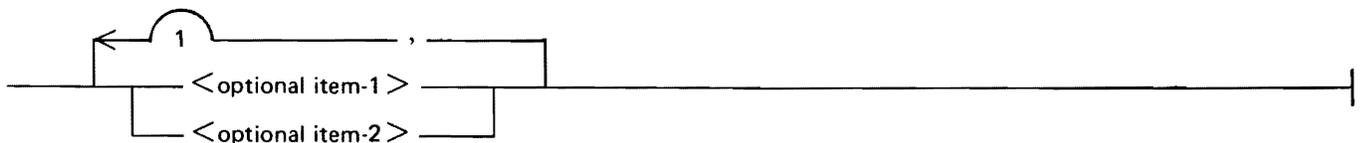
```
        ┌←——— <bridge> < return character> ——————┐
————————┴———— <object of the loop> ——————————————┴—————————————————|
```
G50054

Example:

```
        ┌←—1———————————— , ———┐
————————┴—— <optional item-1> ——┤
          └—— <optional item-2> ——┘——————————————————————————————|
```
G50055

The following statements can be constructed from the railroad diagram in the example.

<optional-item-1>

<optional-item-2>

<optional-item-1>,<optional-item-1>

<optional-item-1>,<optional-item-2>

<optional-item-2>,<optional-item-1>

<optional-item-2>,<optional-item-2>

A <loop> must be traversed in the direction of the arrow heads, and the limits specified by bridges cannot be exceeded.

Bridges

A bridge illustrates the minimum or maximum number of times a path may be traversed in a railroad diagram.

There are two forms of <bridges>.

n   n is an integer which specifies the maximum
      number of times the path may be traversed.

n*   n is an integer which specifies the minimum
      number of times the path must be traversed.

G50056

Example:

```
        ┌────── 2 ──────── , ──┐
   ┌────┤                      ├──────────────────────────┤
   │    └──────<optional item-1>──────┘
   │    ┌─ 1* ─── <optional item-2> ──┘
```
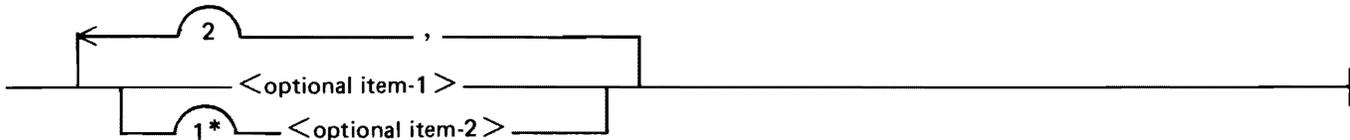
G50057

The loop may be traversed a maximum of two times; however, the path for <optional-item-2> must be traversed at least one time.

The following statements can be constructed from the railroad diagram in the example.

<optional-item-1>,<optional-item-2>

<optional-item-2>,<optional-item-2>,<optional-item-1>

<optional-item-2>

RELATED DOCUMENTATION

The following manuals are referenced in this document:

B 1000 Systems Command and Edit (CANDE) Language User's Manual, form number 1090586.
B 1000 Systems Supervisory Message Control System (SMCS) Reference Manual, form number 1108891.
B 1000 Systems System Software Operation Guide, Volume 2, form number 1152097.

SECTION 2
BEGINNING IBASIC

The purpose of this section of the manual is to help beginners as well as those who need a review of the fundamental commands in the use of IBASIC. After some introductory comments about IBA-SIC, this section guides the user through several examples that teach most of the fundamental commands of IBASIC. For maximum benefit, the user should have access to a terminal through which the commands and statements presented in this section can be entered. After the user becomes familiar with the commands used in this section, this manual will generally be needed only as a reference document.

The easiest way to learn about IBASIC is to use it. Its use is easily learned because IBASIC "talks" to the user. As the user enters information to IBASIC, IBASIC tells the user whether the input is correct by responding with the corresponding output or with an error message. This dialogue is referred to in this manual as "interaction."

Two fundamental types of instructions can be presented to IBASIC: (1) system commands, and (2) BASIC language commands and statements. With system commands, the user requests information about the program currently being written, requests that a particular interactive operation be performed, or requests other information concerning the IBASIC system. Some examples are RUN, MAKE, LIST, and DELETE.

BASIC language statements and commands are instructions that are carried out by that portion of IBA-SIC that executes or performs specified operations of the BASIC language. BASIC statements are entered and stored for later execution (Entry mode). BASIC commands are executed immediately (Command mode). These two modes are used in the examples in this section. They are briefly explained under Command Mode in this section, and are fully described in Section 11 under BASIC Commands and BASIC Statement Entry. Examples of BASIC language commands and/or statements are PRINT, READ, GOTO, and LET.

Through entry of BASIC commands and statements, the user changes the set of data upon which the IBASIC system operates. This set of data is referred to as the BASIC environment. It consists of two parts: (1) the set of BASIC statements that make up a program and (2) the data that is stored in BA-SIC variables, that is, data names whose value can be changed. Many of the commands and statements that enable the user to operate on this environment are introduced in this section. The statements and commands in this section were tested on a Burroughs TD832 terminal. Output may be slightly different for other types of terminals.

IBASIC USE

The paragraphs that follow guide the user through the fundamental commands of IBASIC and teach some of the commands and statements from the BASIC language. When asked to enter a command, the user must type the command on one of the three uppermost lines of the terminal, most conveniently beginning in the upper left-hand corner, and then press the transmit (XMT) key. For each operation that the user is asked to perform, the manual explains exactly what is taking place.

Executing IBASIC

There are several ways to initiate execution of IBASIC. Only one method is shown in this section. A description of all methods is found in Appendix B. (Appendix B should be consulted in order to initially configure IBASIC for the examples in this section as well as for normal execution. In this section, it is assumed that IBASIC has been configured to be executed under the Supervisory Message Control System (SMCS) using the SIGN ON syntax as shown in Appendix B.)

To start execution of IBASIC, enter the following from a terminal:

USER <usercode>/<password> ON IBASIC

The computer responds with several messages. The ones pertinent to IBASIC are similar to the following:

MESSAGE QUEUED FOR "IBASIC": WAITING STARTUP
SIGNED ON TO "IBASIC", SIGNAL = *
B1000 BASIC MARK 10.0.0I (04/16/81 13:20)
(<usercode>) logged on at 15:24:29.2

These are some of the initial messages displayed when the SMCS SIGN ON command is used to start up IBASIC. It is not necessary to understand exactly what they mean in order to use IBASIC.

After these initial messages, a status line is displayed on the bottom line of the terminal and a number sign (#) is displayed in the upper left-hand corner of the terminal. The status line tells the user what the IBASIC system is doing. The number sign signifies that the system has finished a previous instruction and is waiting to receive input.

NOTE
If the status line is not displayed on the bottom line of the terminal, or if the number sign is not in the upper left-hand corner, the status line option must be switched off. Refer to STATUSLINE in Section 12.

Making a File

To begin writing a BASIC program, enter the following:

MAKE PROGRAM1

This command creates a workfile named PROGRAM1 into which user-entered program statements are stored. After the number sign (#) is displayed again, the system is ready for entry of BASIC language statements.

Enter the following statements one line at a time, pressing the transmit key (XMT) after typing each line. Be sure to include the line numbers (10, 20, 30, 40) as they appear below.

10 PRINT "PROGRAM1 ADDS TWO NUMBERS AND PRINTS THE RESULT."
20 LET A = 9 + 4
30 PRINT "9 + 4 =";A
40 END

Each of these statements is entered into the workfile PROGRAM1 after transmission. They are stored there until the user explicitly removes one or more of the statements or until the workfile is removed. These statements comprise a BASIC program. When the program is run, the first PRINT statement causes display of the string of characters that appears between the quotation marks. The LET statement adds 9 and 4 and assigns the sum to variable A. The second PRINT statement causes display of the string between quotation marks and the value assigned to variable A. The END statement signifies the end of the program.

Before instructing the computer to execute PROGRAM1, the user should consider what happens in the case of an accidentally misspelled BASIC keyword, one of the predefined words which make up the BASIC language. Assume that the keyword PRINT was accidentally entered as PINT on line 30. To see the effects, enter the following:

    30 PINT "9 + 4 = ";A

The user statement and the output will look like the following:

    #30 PINT "9 + 4 = ";A
    – – – – >
    error 20 – incomprehensible statement

The arrow points to the beginning of the portion of the statement that contains the error. The error message will help the user to determine exactly what is wrong with the statement. In this case IBASIC could not understand what statement was entered since PINT is not a BASIC keyword. This new statement 30 replaced the old statement 30, even though the new statement was in error. Enter the following to list PROGRAM1 and see the erroneous statement in relation to the correct statements:

    LIST

The LIST command lists the program statements. The statement in error is highlighted. In order to correct the error, re-enter the line as follows.

    30 PRINT "9 + 4 = ";A

The program is now the same as it was originally.

There are many errors that can be detected upon transmission of a line. Each of them is accompanied by a descriptive message to help the user determine the cause.

Executing a BASIC Program

In order to instruct the computer to execute the program just written, enter the following:

    RUN

The RUN command initiates execution of the program. In PROGRAM1, the first statement executed is the PRINT statement on line 10. Line 20 is executed next, then line 30, and line 40 last. The flow of execution continues in this manner unless the programmer specifies that it be changed. Statements that change the flow are described in detail in Section 7. One of these statements, the GOTO statement, is briefly mentioned in this section.

The output from the RUN command previously entered is similar to the following:

    running "PROGRAM1" from line 10 at 11:06:46.4
    PROGRAM1 ADDS TWO NUMBERS AND PRINTS THE RESULT.
    9 + 4 = 13
    end run of "PROGRAM1" at line 40

Editing a Program

It is obvious that PROGRAM1 executes correctly, since there were no errors and the output was what was expected, but one hardly needs a computer to find the sum of 9 and 4. The program can easily be changed to handle more significant problems by the addition and deletion of a few statements. The following paragraphs direct the user to change or edit this program in such a way that it will compute and display the sum of almost any two numbers. The numbers it will not be able to compute are those that are too large for the computer to store.

First, do a LIST command to see exactly what the program looks like.

Line 20 is not needed because it only works with the integers 9 and 4. So, delete it by entering the following:

DELETE 20

This command deletes line 20 from the file currently being written. Now replace the deleted statement with the following:

20 INPUT A

This statement allows the user to enter a numeric value into variable A from a terminal. The DELETE command was not really necessary since re-entering a new line 20 would have written over the old line 20. The DELETE command is included here so that the user would become familiar with it. To allow a second input value, enter the following statement:

25 INPUT B

This statement performs the same operation as line 20, except that the input value is assigned to variable B. Enter another LIST command to see what has been done.

Notice that the IBASIC system automatically put the two lines into their proper numerical order. Now, re-enter the PRINT statement on line 30 in the following way:

30 PRINT A; "+"; B; "="; A + B

When run, this PRINT statement causes display of the value stored in variable A, a plus sign (+), the value stored in variable B, an equal sign (=), and the value A + B. Now add the following statement so that the addition can be done repeatedly:

35 GOTO 20

Now execute the program again by entering the RUN command.

A question mark (?) is displayed in the upper left-hand corner of the terminal. This is a prompt, issued as a result of the appearance of the INPUT statement. A prompt tells the user to enter the required data. In this case, the data is a number. Enter a number. Enter a second number when the next prompt is displayed. The program displays the sum. Enter several pairs of numbers to make sure that the program works correctly.

Stopping Execution of a Program

This program would go on forever, if allowed, but the user probably has better things to do. To stop the program, depress the SPCFY key on the keyboard of a TD series terminal. For other types of terminals, refer to the Pseudo BREAK Feature subsection in Section 11.

IBASIC responds with a message similar to the following:

> > > BREAK received − INPUT terminated at line 20

A More Complex Example

Remove PROGRAM1 and make a new file with a name of your choice by entering the following two instructions.

    SCRATCH

    MAKE < file-name >

< file-name > must be created by the user. For example, the first or last name of the user will probably be valid. The maximum length allowed is ten characters.

The following sample program calculates the greatest common divisor (GCD) of two integers. Enter the following statements. As before, transmit after each statement is entered. If a line of input is too long for one line on the terminal, continue typing onto the next terminal line − IBASIC allows up to 256 characters to be entered per line.

```
10 PRINT "THIS PROGRAM CALCULATES THE GREATEST COMMON DIVISOR OF TWO
INTEGERS. IT IS ONLY ACCURATE UP TO 16,777,215."
20 PRINT
30 PRINT "ENTER THE FIRST INTEGER."
40 INPUT A
50 LET C = A
60 PRINT "ENTER THE SECOND INTEGER."
70 INPUT B
80 LET D = B
90 IF A = B THEN 150
100 IF A < B THEN 130
110 LET A = A − B
120 GOTO 90
130 LET B = B − A   ·
140 GOTO 90
150 PRINT "THE GREATEST COMMON DIVISOR OF"; C; "AND"; D; "IS"; A
160 PRINT
170 PRINT "DO YOU WANT TO CONTINUE? Y OR N"
180 INPUT E$
190 IF E$ = "Y" THEN 30
200 IF E$ = "N" THEN 230
210 PRINT "COME ON, Y OR N CANNOT BE THAT HARD...TRY AGAIN."
220 GOTO 180
230 END
```

Before proceeding to an explanation of the new statements used in this program, enter the RUN command to see how it works.

Instructions for the program are displayed by the program on the user's terminal. Follow these instructions and enter several pairs of integers to validate the correctness of the program and to become familiar with it.

Unlike the previous example in which outside intervention (BREAK) was necessary to stop execution, the GCD program can be stopped programmatically. In other words, the user can instruct the program to stop itself without any outside intervention. Stop the program by following the instructions displayed by the program.

Each of the statements in the current program — program that is currently loaded into the BASIC environment — is now briefly described. In order to see each statement as it is described, do a LIST in the following manner:

    LIST 10 TO 200

The PRINT statement on line 10 causes display of the string between the quotation marks. This string contains instructions for the operator of the program. Notice that the string of characters in the sample program is too long to fit on one terminal line so it has been continued on the next terminal line.

The PRINT statement on line 20 causes display of a blank line to improve readability of the program as it is executed.

The PRINT statement on line 30 causes display of the string between the quotation marks. This quoted string (string within quotation marks) instructs the user to enter an integer value.

Line 40 contains the INPUT statement that allows input to the program. It also causes the prompt character (?) to be displayed on the terminal.

Line 50 causes the contents of variable A to be assigned to variable C.

The PRINT statement on line 60 causes display of the string within the quotation marks. The string contains additional instructions.

Line 70 accomplishes the same task that line 40 did, except that variable B is used.

Line 80 causes the contents of variable B to be assigned to variable D.

Line 90 contains a statement that has not yet been described, the IF statement. The IF statement tests a condition to see whether it is true or false. In this case, the condition is "Is A equal to B?" If the condition is true, the statement following the word THEN is executed. When this statement is a line number, the next statement executed is the statement whose line number appears after the word THEN. Line 90 is read in the following way: If the contents of variable A are equal to the contents of variable B, then go to line number 150. If A is not equal to B, the statement following the IF statement is executed.

Line 100 contains another IF statement. This statement is read as follows: If the contents of variable A are less than the contents of variable B, then go to line number 130. If A is not less than B, the statement on line 110 is executed.

On line 110, the contents of B are subtracted from the contents of A and the difference is stored in variable A.

Line 120 jumps execution back to line 90.

On line 130, the contents of A are subtracted from the contents of B and the difference is stored in variable B.

Line 140 is a duplicate of line 120.

The PRINT statement on line 150 causes the answer to be displayed.

The PRINT statement on line 160, like line 20, is for readability.

Line 170 asks if the program is to be continued.

Line 180 contains another INPUT statement. This time, though, a dollar sign ($) comes after the variable. The variables that have been used up to now (A, B, C, D) only allow the use of numbers with them. A variable with a dollar sign following it allows the use of alphabetic characters. Variable E$ receives the N or Y that the user enters.

Line 190 tests the contents of variable E$. If E$ contains a Y, then the program continues at line 30.

Line 200 also tests the contents of variable E$. If E$ contains an N, then line 230 is executed next.

Do another LIST:

    LIST 210 TO 230

Now the remainder of the statements can be seen.

Line 210 displays an instructive message to the user. This line is executed only if E$ does not contain an N or a Y.

Line 220 causes execution to continue with line 180.

Line 230 ends the program.

Even though all the statements have been discussed separately, the user may still not understand how the program actually calculates the GCD of two integers. In order to understand this, each statement must be understood in relation to the other statements in the program. This exercise, left to the user, entails simulating the computer; that is, thinking through what the computer would do with each statement of the program.

Program Debugging Commands

There are two additional types of commands described in this section. The first type, IBASIC debugging commands, involves several more IBASIC commands. The second, command mode, involves using the BASIC language statements in a different manner than they have been used up to this point in the manual. The GCD program is used to explain these features.

Some IBASIC Commands for Debugging

Enter the RUN command again. Now enter the integers 16777215 and 16777214. The user will certainly tire of waiting for this answer, but the delay will give us time to examine the two types of commands mentioned. Begin by depressing the SPCFY key as in the previous example. The use of the SPCFY key in this manner is known as a BREAK. The user will now be guided in the use of several commands useful in examining the execution of a program.

First, find out where execution was terminated by entering the following IBASIC command:

    WHERE

IBASIC responds with a message similar to the following:

    you are stopped − ready to continue at line 130

1152105   2-7

After a program is stopped, it may be executed one statement at a time with the STEP command. The STEP command may be entered in two ways: (1) by explicitly entering the word STEP, or (2) by pressing the SPCFY key. Enter the STEP command if it has not been entered already.

IBASIC executes the next statement, displays it, and stops before the execution of the subsequent statement. An entire program may be executed in this manner, if desired. STEP is useful when the execution of sections of code need to be closely observed. To resume normal execution, use the CONTINUE command.

CONTINUE

Before this program terminates, stop execution again by using the BREAK feature (SPCFY key).

Command Mode

As mentioned in the first part of this section, BASIC language statements may be entered in two ways: Command mode and Entry mode. All of the BASIC statements entered up to this point in the manual have been in the Entry mode. Entry mode statements are entered into a file and stored for subsequent use and are characterized by a line number at the beginning of the line.

In Command mode, the line number is omitted. This omission tells IBASIC that the statement is to be executed immediately. Many of the BASIC statements may be used in Command mode. A list of the statements that cannot be used in Command mode appears in BASIC Commands in Section 11. To understand how Command mode can help in analyzing a program, enter the following:

PRINT A; B

The values displayed are the current values of the variables A and B from the GCD program. They are only of value to those users who understand how the program calculates the greatest common divisor. These users know that A is continually subtracted from B, or vice versa, until the two variables are equal. As one can see from the output of the PRINT statement previously entered in command mode, one of the variables has the value 1 and the other contains a very large number. Obviously, the program takes quite a while to run to termination. The time necessary for this program to finish can be reduced if the larger of the two variables is made smaller. So, using another BASIC statement in Command mode, change the value of the larger variable. You can determine which of the two variables is the larger by using the PRINT command (Command mode).

Change the value of the larger variable to 1000 by entering the following statement:

LET <larger variable> = 1000

Resume execution by entering the CONTINUE command. The program will terminate quite rapidly.

SAVE, SCRATCH, and BYE Commands

The last three commands that the user will want to learn about before leaving the terminal are the SAVE, SCRATCH, and BYE commands.

If you want to save the program for later use, enter the following:

SAVE

The SAVE command creates a copy of the current workfile and stores it on disk. If you do not want to SAVE the program, enter the following:

SCRATCH

The SCRATCH command, as previously shown, removes the current program.

In order to terminate the IBASIC system, enter the following command:

BYE

The BYE command causes termination of the IBASIC system. The output is similar to the following:

connect time = 00:05:36.0, cpu time = 2 units
(<usercode>) logged off at 15:27:29.6
goodbye
REMOTE FILE CLOSED BY "IBASIC".

In order to sign off SMCS, enter BYE again.

SECTION 3
PROGRAM COMPOSITION

A BASIC program is made up of a number of lines. Each line consists of characters from the character set for BASIC. The concepts of lines, characters, and the fundamental rules for combining these elements in order to write BASIC programs are described in this section.

STATEMENT LINES

A statement line consists of a line number followed by a BASIC language statement. The maximum length allowed for a statement line is 256 characters. There may be a maximum of 1979 statement lines in a program.

Unless otherwise specified, the term "line" in this manual is not synonymous with the same term as used in relation to a terminal or a printer. For example, one statement line may occupy three "lines" on a terminal.

Syntax:

G18000
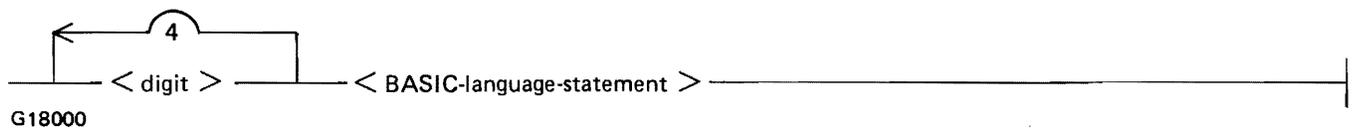
< digit > is any decimal digit. At least one < digit > must be nonzero. Leading zeros have no effect, other than counting as a digit in the line number. < BASIC-language-statement > s are described in Sections 4 through 10.

Examples of statement lines:

```
99999 PRINT A
    1 REM THIS IS THE START OF THE PROGRAM.
 0001 PRINT "ENTER YOUR NAME."
02340 LET A$(X*Y) = "25" & A1$
  100 BREAK
```

CHARACTER SET

A BASIC statement line consists of characters from the character set for BASIC. The standard character set for BASIC is contained in the International Reference Version of ISO Standard 646, 7-Bit Input/Output Coded Character Set, 4th Edition. Table E-1 in Appendix E contains this character set. BASIC can also use the EBCDIC character set if the user desires (refer to the OPTION statement in Section 5). Table E-2 in Appendix E contains this character set. All lower-case characters are translated directly to their upper-case equivalent everywhere within the IBASIC system except for strings enclosed within quotation marks (for example, print a,b is equivalent to PRINT A,B).

PROGRAM DOCUMENTATION

BASIC programs may be documented with tail comments at the end of statement lines or with a separate statement called a REM statement.

Tail Comments

A tail comment can be entered at the end of a line in order to provide a clear description of what a BASIC statement or a group of BASIC statements does. Tail comments have no effect on the execution of a program.

Syntax:

```
─┬─┌─ < statement-line > ─┬─┌─ < exclamation-mark >  < remark-string > ─────────────────────┤
 │  └──────────────────────┘
```

G18001

<statement-line> is described under Statement Lines in this section. <exclamation-mark> indicates the beginning of the tail comment. <remark-string> is any string of characters that the programmer chooses in order to explain the statements of the program.

Examples of tail comments:

```
        PRINT A, B, C        PRINT THE ANSWERS
        LET T = T2 – T1      SUB INITIAL TIME FROM TERMINAL TIME
```

REM Statement

The REM statement must occur on a line by itself. It serves to document a program and has no effect on the execution of the program.

Syntax:

```
─────── REM  < remark-string > ──────────────────────────────────────────────────────┤
```
G18002

<remark-string> is any string of characters that the programmer chooses in order to explain the statements of the program.

Examples of REM statements:

```
    REM     THIS PROGRAM CALCULATES THE GREATEST COMMON DIVISOR
    REM        OF TWO INTEGERS.
```

STOP Statement

The STOP statement causes termination of the program.

Syntax:

```
─────── STOP ─────────────────────────────────────────────────────────────────────────┤
```
G18003

The STOP statement may occur anywhere within a program.

END Statement

The END statement marks the physical end of the program and causes termination of execution of the program when encountered.

Syntax:

```
———— END ————————————————————————————————————————|
```

G18004

The END statement may occur only at the physical end of the main program.

GENERAL SYNTAX RULES

The following rules must be observed in writing a BASIC program.

1. Each statement of a program must begin with a unique, positive, nonzero line number and must contain one BASIC statement.
2. Spaces must not occur within keywords, within the word TAB in a tab call, within numeric constants, within line numbers, within variable names, or within multicharacter relation symbols ($> =$, $= <$). Spaces can occur anywhere else within a program to enhance readability.
3. Spaces must occur around keywords. A keyword is a character string that provides a distinctive identification of a statement or a component of a statement. Examples of keywords include ELSE, GOTO, MAT, PRINT, READ, THEN, USING and WRITE.
4. Each program must terminate with an END statement.
5. Lines are executed in sequential order, starting with the first line in the program, and continuing until some other action is dictated by execution of a control statement, until a STOP or END statement is executed, or until the occurrence of a fatal error (an error which stops the execution of a program).
6. Upper-case and lower-case characters are interchangeable. Lower-case characters are translated to their upper-case equivalents everywhere within the IBASIC system except in strings within quotation marks.

SECTION 4
NUMERIC DATA CONSTRUCTS

There are two data types in BASIC: numeric and string. Associated with each of these data types are constants, variables, and intrinsic functions from which expressions can be formed. This section deals with data type numeric and the expressions that can be formed from the fundamental numeric constructs. Strings are explained in Section 5.

NUMERIC CONSTANTS

Numeric constants are used to denote numeric values. Unlike some programming languages, BASIC does not distinguish between numbers containing a decimal point (real numbers) and those written without a decimal point (integers): all numeric values in IBASIC are stored internally in floating-point form, that is, as a sign, exponent, and fraction, and are handled as real numbers.

Syntax:

```
  ┬─ < sign > ─┬─┬─ < integer > ─┬─ . ─┬─ < integer > ─┬─┬─ E ─┬─ < sign > ─┬─ < integer > ─┬──┤
   └───────────┘ │                     └───────────────┘ │      └───────────┘                 │
                 │                                        │                                     │
                 └───────────── . < integer > ───────────┘
G18005
```
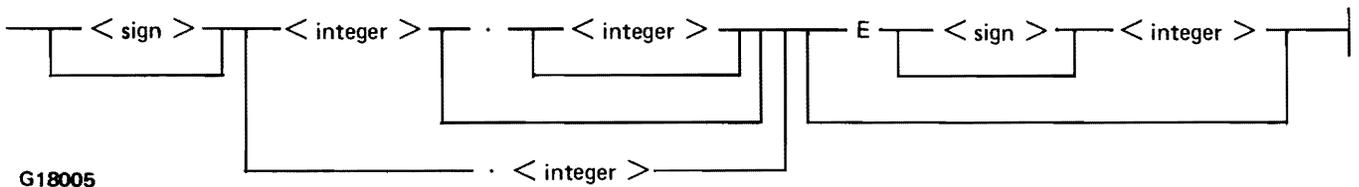
< sign > is a plus sign (+) or a minus sign (-). < integer > is a series of decimal digits.

Semantics:

Numeric values are maintained with a precision of at least six decimal digits (21 to 24 binary digits depending on the value). They may range from approximately 5.39761E-79 to 7.23701E+75.

E signifies "times ten to the power." For example, 2.145E-4 is read as 2.145 times ten to the power −4 and represents the value .0002145. If the < sign > is omitted, plus (+) is assumed.

Examples of numeric constants:

    −21.
    1E10
    5E-2
    .4E+1
    500
    1
    .255

NUMERIC VARIABLES

A numeric variable is a symbolic name used to represent a numeric value which may be changed during program execution by a numeric assignment statement. (Refer to Numeric Assignment Statement in this section.) Numeric variables may either be simple or subscripted. All numeric variables, whether simple or subscripted, are of type real. A subscripted variable is one element of an array. Arrays are described in Section 6.

Syntax:
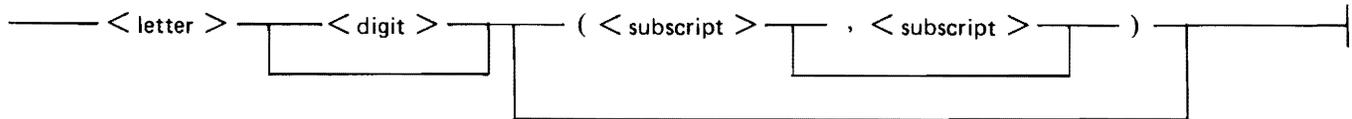
G18006

<letter> is any English alphabet character (A through Z). <digit> is any decimal digit (0 through 9). The same <letter> or <letter> <digit> combination cannot be used as the name of both a simple numeric variable and a numeric array, nor the name of both a 1-dimensional (one <subscript>) and a 2-dimensional (two <subscript>s) numeric array. <subscript> is any numeric expression. It is an index into the array. <subscript> is always rounded to the nearest integer. The rounded value is defined as INT(<subscript> + .5), where INT signifies "the largest integer not greater than." A <subscript> must be in the allowable range of subscripts for the array being referenced.

Semantics:

Explicit declarations of numeric variables in BASIC are not necessary except in the case of certain subscripted numeric variables. These declarations are described in Section 6. Numeric variables with no explicit declaration are implicitly declared through their appearance in a program unit. Of these implicitly declared numeric variables, those followed by one or two subscripts are numeric arrays whose subscripts can range in value from zero, or one, to ten. A subscripted numeric variable refers to the element in the 1- or 2-dimensional array selected by the value(s) of the subscript(s). A numeric variable that has no subscripts and does not occur in a MAT statement (refer to Section 6) is a simple variable.

The initial value of each numeric variable at execution time is zero.

Examples of numeric variables:

    X
    A5
    V(4)
    W(X,X + Y/2)

NUMERIC ASSIGNMENT STATEMENT

The numeric assignment statement is used to assign a computed value to a list of simple or subscripted variables.

Syntax:

```
        ┌─────────── , ───────────┐
──┬─ LET ─┬─┴─< numeric-variable >─┴── = < numeric-expression > ──────────────┤
  └───────┘
```
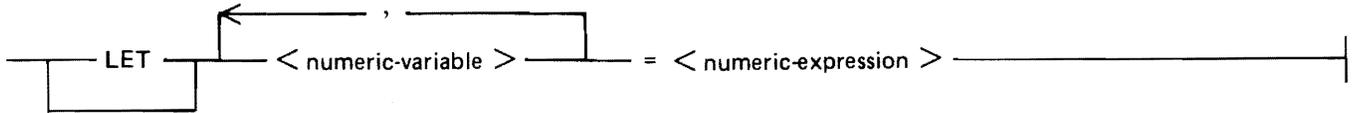
G18007

<numeric-variable> is described under Numeric Variables in this section. Simple and subscripted variables may appear in the same list of <numeric-variable>s. <numeric-expression> is any numeric expression valid in BASIC.

Semantics:

A numeric assignment statement is evaluated in the following manner: The subscripts, if any, of variables in the <numeric-variable> list are evaluated in sequence from left to right. Next, the expression on the right of the equal sign (=) is evaluated. Finally, the value of that expression is assigned to each variable in the <numeric-variable> list.

Examples of numeric assignment statements:

```
LET P = 3.14159            ! P gets an approximation of PI.

LET A(X,3) = SIN(X)*Y + 1  ! Array element A(X,3) is
                           ! assigned the expression value.

LET A, Y(I), Z = I + 1     ! A, Y(I), and Z are assigned
                           ! I + 1.

A = B                      ! A is assigned the value of B.

LET T(I,J), I, J = I + J   ! The listed variables are
                           ! assigned the sum of I and J.
```

In the last example, understanding the order of evaluation of a numeric assignment statement is necessary in order to fully understand the statement. Assume that variables I and J have the values 1 and 2 respectively. The subscripts, I and J, in the reference to array T are evaluated first. Thus, the array element being referenced is T(1,2). The numeric expression, I + J, is then evaluated. Its value is 3. Thus, the assignment statement results in the assignment of 3 to variables T(1,2), I, and J. In contrast, if subscripts in the <numeric-variable> list were evaluated after the <numeric expression> had been assigned to I and J, the array element referenced would have been T(3,3).

NUMERIC EXPRESSIONS

A numeric expression is any numeric constant, numeric variable, numeric function reference, or a combination of these separated by the operators representing addition, subtraction, multiplication, division and exponentiation.

Syntax:
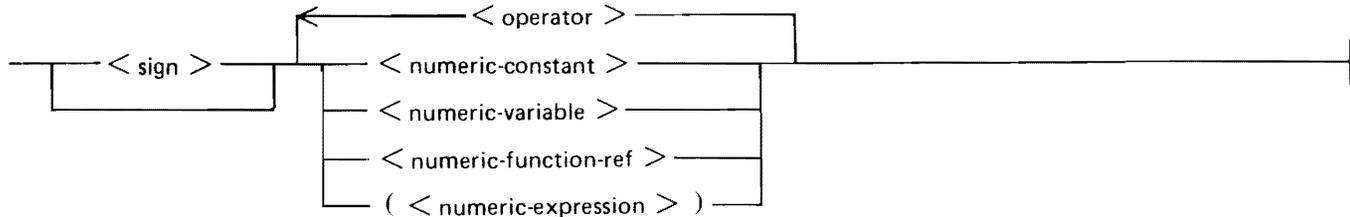
G18008

<operator> is one of the following:

Operator and Name		Meaning
^	Circumflex accent	Exponentiation
**	Double asterisk	Exponentiation
+	Unary plus	No action
−	Unary minus	Negation
*	Asterisk	Multiplication
/	Solidus	Division
+	Plus sign	Addition
−	Minus sign	Subtraction

<numeric-constant> is defined under Numeric Constants in this section. <numeric-variable> is defined under Numeric Variables in this section. <numeric-function-ref> includes the intrinsic numeric functions as defined under Intrinsic Numeric Functions in this section and the user-defined numeric functions as defined under User-Defined Functions in Section 8. User-defined functions must be defined in the program unit in which they are referenced. The appearance of a <numeric-expression> between parentheses signifies that a numeric expression may be used as an operand in a larger numeric expression which encompasses the numeric expression in parentheses.

Semantics:

The rules for formation and evaluation of numeric expressions follow normal algebraic rules. There are four levels of precedence. These levels, listed in order from highest precedence to lowest, follow.

1. Exponentiation
2. Unary plus and minus
3. Multiplication and division
4. Addition and subtraction

The order of evaluation can be changed by the use of parentheses. Operations within parentheses are performed first. Operations on the same precedence level are performed left to right unless parentheses dictate otherwise. Refer to the examples that follow in this subsection for detailed explanations of specific cases.

When numeric overflow or underflow occurs, that is, when a numeric value either exceeds or is less than the allowable limits for numeric values, the condition is reported and execution continues. The resultant value when overflow occurs is the largest representable value, $7.237E+75$. The resultant value when underflow occurs is zero. Division by zero, and zero raised to a negative power are treated as overflows. $0**0$ is defined as 1. When a negative number is raised to a nonintegral power a fatal error occurs.

Examples of numeric expressions:

$3*X - Y**2$	! Y squared is subtracted from the ! product of 3 and X.
$A(1) + A(2) + A(3)$	! The array elements listed are ! added.
$-X/Y$	! Negative X divided by Y.
$2**(-X)$	! 2 raised to the negative Xth ! power.
$SQR(X**2 + Y**2)$	! The square root of the sum of X ! squared and Y squared.
$A - B - C$	! (A − B) − C
$A**B**C$	! (A ** B) ** C
$A/B/C$	! (A / B) / C
$-A**B$	! − (A ** B)

INTRINSIC NUMERIC FUNCTIONS

Predefined functions are supplied as part of the IBASIC system for the evaluation of commonly used numeric functions. The general syntax for the intrinsic numeric functions follows.

Syntax:
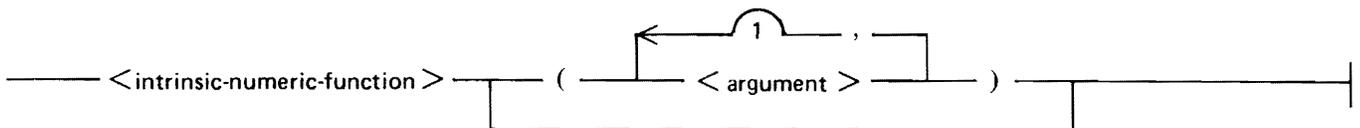
G18009

A description of each of the <intrinsic-numeric-function>s with its meaning follows. Zero, one, or two <argument>s can be included depending on the function. In all cases, X and Y represent numeric expressions and A represents a numeric or string array.

ABS(X)

The absolute value of X.

Example:

```
LET X = 25.5
PRINT "ABSOLUTE VALUE OF A =";ABS(X)
PRINT "ABSOLUTE VALUE OF - 123.45 =";ABS(- 123.45)
```

When the statements in this example are run, the following output is produced.

```
ABSOLUTE VALUE OF A = 25.5
ABSOLUTE VALUE OF - 123.45 = 123.45
```

ACOS(X)

The arccosine of X in radians, where $0 <= ACOS(X) <= PI$; X must be in the range $-1 <= X <= 1$, $PI = 3.14159$.

ANGLE(X,Y)

The angle in radians between the positive x-axis and the vector joining the origin to the point with coordinates (X,Y), where $-PI < ANGLE(X,Y) <= PI$. The values of X and Y cannot both be zero.

Example:

```
PRINT "ANGLE BETWEEN (1,0) AND (1,1) IS";ANGLE(1,1)
```

Execution of this example causes the following to be displayed.

```
ANGLE BETWEEN (1.0) AND (1,1) IS .785398
```

ASIN(X)

The arcsine of X in radians, where $-PI/2 <= ASIN(X) <= PI/2$; X must be in the range $-1 <= X <= 1$.

ATN(X)

The arctangent of X in radians; that is, the angle whose tangent is X, where $-(PI/2) < ATN(X) < (PI/2)$.

CEIL(X)

The smallest integer not less than X.

Examples:

```
PRINT "CEIL OF 1.5 IS";CEIL(1.5)
PRINT "CEIL OF - 1.5 IS ";CEIL(- 1.5)
```

Execution of these statements gives the following output:

```
CEIL OF 1.5 IS 2
CEIL OF - 1.5 IS - 1
```

COS(X)

The cosine of X, where X is in radians.

COSH(X)

The hyperbolic cosine of X, where X is in radians.

COT(X)

The cotangent of X, where X is in radians.

CSC(X)

The cosecant of X, where X is in radians.

DATE

The current date in decimal form YYDDD, where YY is the last two digits of the year and DDD is the number of days elapsed in the year.

Example:

    PRINT DATE

If the date was May 17, 1980, this example would give the following output:

    80138

DEG(X)

The number of degrees in X radians.

Examples:

    PRINT "DEG OF PI IS";DEG(PI)
    PRINT "DEG OF PI/6 IS";DEG(PI/6)

Execution of these examples gives the following output:

    DEG OF PI IS 180
    DEG OF PI/6 IS 30

DET FUNCTION

The DET function produces the determinant of a 2-dimensional square array. The syntax and semantics of the DET function follow.

Syntax:

```
—— DET ——┬—— ( —— <array-name> —— ) ——┬——|
          └──────────────────────────────┘
```

<array-name> is a 2-dimensional array that follows the formation rules of a simple numeric variable.

Semantics:

The DET function returns the value of the determinant of its argument. If there is no argument explicitly named, the DET function returns the value of the determinant of the array most recently used as an argument of a MAT INV function.

If the DET function is used without an argument before the MAT INV function is invoked, a fatal error occurs.

A fatal error also occurs if the array referred to in the DET function is not a 2-dimensional square array.

Examples of the DET function:

    DET
    DET(B)

In the first example, no argument is explicitly named. In this case, the DET function returns the value of the determinant of the array most recently used as an argument of a MAT INV function. In the second example, the DET function returns the value of the determinant of array B.

In using the second example, assume that array B is a 2-by-2 array and contains the following values:

    2   5
    4   8

In this case, DET(B) returns the value -4.

EPS

The smallest positive nonzero number that can be represented by the machine: 5.39761E-79

EXP(X)

The exponential of X; that is, the value of the base of natural logarithms (2.71828) raised to the power X.

FP(X)

The fractional part of X, FP(X), is equivalent to X $-$ IP(X), where IP signifies "integer part of."

Example:

    PRINT "FP OF 17.358795 IS";FP(17.358795)

Execution of this example produces the following output:

    FP OF 17.358795 IS .358795

INF

The largest positive number that can be represented by the machine: 7.237E+75

INT(X)

The largest integer not greater than X.

Example:

```
A = 6.9
B = 6
C = -6.14
PRINT "LARGEST INTEGER NOT GREATER THAN"; A;" IS"; INT(A)
PRINT "LARGEST INTEGER NOT GREATER THAN"; B;" IS"; INT(B)
PRINT "LARGEST INTEGER NOT GREATER THAN ";C;" IS "; INT(C)
PRINT "LARGEST INTEGER NOT GREATER THAN -2 IS "; INT(-2)
```

Execution of the above example causes the following to be displayed.

```
LARGEST INTEGER NOT GREATER THAN 6.9 IS 6
LARGEST INTEGER NOT GREATER THAN 6 IS 6
LARGEST INTEGER NOT GREATER THAN -6.14 IS -7
LARGEST INTEGER NOT GREATER THAN -2 IS -2
```

IP(X)

The integer part of X. IP(X) is equivalent to SGN(X)*INT(ABS(X)).

Examples:

```
PRINT "IP OF -6.14 IS";IP(-6.14)
PRINT "IP OF 6.9 IS";IP(6.9)
```

Execution of these examples causes the following to be displayed.

```
IP OF -6.14 IS -6
IP OF 6.9 IS 6
```

LDIM(A,X)

The lower bound for the Xth subscript of array A.

Example:

```
DIM A(10,10)
PRINT "LDIM OF THE 2ND SUBSCRIPT OF ARRAY A IS";LDIM(A,2)
```

Execution of this example gives the following output:

```
LDIM OF THE 2ND SUBSCRIPT OF ARRAY A IS 0.
```

LOG(X)

The natural logarithm of X. X must be greater than zero.

LOG10(X)

The common logarithm of X. X must be greater than zero.

LOG2(X)

The base 2 logarithm of X. X must be greater than zero.

MAX(X,Y)

The larger of X and Y.

MIN(X,Y)

The smaller of X and Y.

MOD(X,Y)

The modulo function, MOD(X,Y), is equivalent to $X - Y*INT(X/Y)$ if Y is nonzero, and is equivalent to 0 if Y is zero.

Example:

```
PRINT "MOD OF 100 AND 90 IS";MOD(100,90)
PRINT "MOD OF 10 AND  -4 IS ";MOD(10,-4)
```

Execution of this example causes the following to be displayed.

```
MOD OF 100 AND 90 IS 10
MOD OF 10 AND  -4 IS  -2
```

PI

The constant 3.14159, which is the ratio of the circumference of a circle to its diameter.

RAD(X)

The number of radians in X degrees.

Example:

```
PRINT "RAD OF 30 DEGREES IS";RAD(30)
```

Execution of this example gives the following output:

```
RAD OF 30 DEGREES IS .523599
```

REM(X,Y)

The remainder function. REM(X,Y) is equivalent to $X - Y*IP(X/Y)$ if Y is nonzero, and is equivalent to 0 if Y = 0.

Examples:

```
PRINT "REM OF 100 AND 90 IS";REM(100,90)
PRINT "REM OF 10 AND  -4 IS";REM(10,-4)
```

Execution of these examples causes the following to be displayed.

```
REM OF 100 AND 90 IS 10
REM OF 10 AND  -4 IS 2
```

RND

The next pseudo-random number in the sequence of pseudo-random numbers uniformly distributed in the range 0 < = RND < 1. The RANDOMIZE statement can be used in conjunction with the RND function to initiate a different and unpredictable sequence of pseudo-random numbers. Refer to the subsection entitled RANDOMIZE statement in this section.

SEC(X)

The secant of X, where X is in radians.

SGN(X)

The sign of X, SGN(X), is -1 if $X < 0$, is 0 if $X = 0$, and is $+1$ if $X > 0$.

SIN(X)

The sine of X, where X is in radians.

SINH(X)

The hyperbolic sine of X, where X is in radians.

SQR(X)

The nonnegative square root of X. X must be nonnegative.

TAN(X)

The tangent of X, where X is in radians.

TANH(X)

The hyperbolic tangent of X, where X is in radians.

TIME

The time elapsed since the previous midnight, expressed in seconds; for example, the value of TIME at 11:15 AM is 40500.

Example:

```
T = TIME
FOR X = 1 TO 5E5
    LET A = X
NEXT X
PRINT "ELAPSED TIME IS";TIME - T;"SECONDS."
```

Execution of this program segment causes the following to be displayed.

ELAPSED TIME IS 123 SECONDS.

UDIM(A,X)

The upper bound for the Xth subscript of array A.

Example:

```
DIM  B(100,10)
PRINT "UDIM OF THE 1ST SUBSCRIPT OF B IS";UDIM(B,1)
```

Execution of this partial program causes the following to be displayed.

UDIM OF THE 1ST SUBSCRIPT OF B IS 100

As with numeric expressions, in case of an overflow or underflow in a numeric function, the condition is reported, the appropriate value (the largest machine value for overflow and zero for underflow), is substituted and execution continues.

The result of evaluating a numeric function is a scalar numeric value (quantity characterized by a single value) which replaces the <numeric-function-ref> in the numeric expression.

RANDOMIZE STATEMENT

The RANDOMIZE statement overrides the normal sequence of pseudo-random numbers used as values for the RND function, generating a different and unpredictable sequence of pseudo-random numbers used subsequently by the RND function.

Syntax:

```
——— RANDOMIZE ————————————————————————————————————————————————|
G18010
```

SECTION 5
STRING DATA CONSTRUCTS

There are two data types in BASIC: numeric and string. Numeric constructs are described in Section 4. Strings may contain arbitrary sequences of characters, and their lengths are variable. The constructs used with strings are described in detail in this section.

STRING CONSTANTS

A string constant is a string of characters enclosed within quotation marks (").

Syntax:

G18011
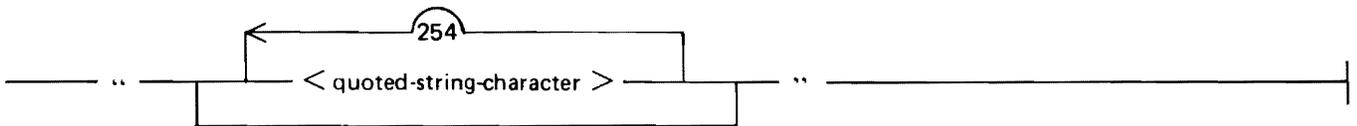
< quoted-string-character > is any character in Table E-1, Appendix E, except those characters in ordinal positions 0 through 31, 34, 64, 91 through 93, 96, and 123 through 127. A quotation mark, ordinal position 34, may be included in a string constant by representing it by two adjacent quotation marks. The length of a string constant is limited to 255 bytes.

Examples:

```
"XYZ"
"1E10"
"He said,""Don't."""   ! Two adjacent quotation marks,
                        ! used to represent one quotation
                        ! mark inside the string.
```

STRING VARIABLES

A string variable is a symbolic name used to represent a string value. A string variable may be changed during program execution by a string assignment statement (refer to String Assignment Statement in this section). String variables may either be simple or subscripted. The proper syntax for a string variable follows.

Syntax:
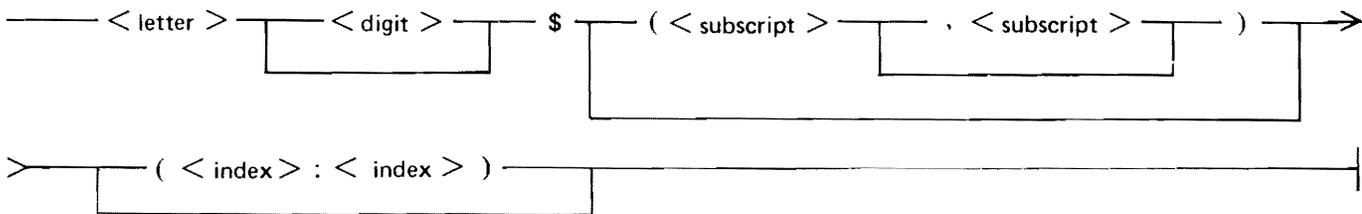
G18012

<letter> is any English alphabet character (A through Z). <digit> is any decimal digit (0 through 9). The same <letter> or <letter> <digit> combination cannot be used as the name of both a simple string variable and a string array, nor as the name of both a 1-dimensional and a 2-dimensional string array. <subscript> is any numeric expression. <subscript> is always rounded to the nearest integer. The rounded value is defined as INT(<subscript> + .5). A <subscript> must be in the allowable range of subscripts for the array being referenced. <index> is an index to a byte of the string variable.

Semantics:

The length of the character string associated with a string variable can vary during the execution of a program from a length of zero characters, signifying the null or empty string, to a maximum length of 255 characters, or to the length defined in a string declaration. Refer to String Declarations in this section for further information on length specification of a string variable.

Explicit declarations of string variables in BASIC are not necessary except when it is desired to set a maximum assignable length for the string to a value less than 255. Declarations involving subscripts are described in Section 6. The description of the method for declaring a specific length is under String Declarations in this section.

String variables with no explicit declaration are implicitly declared through their appearance in a program. Of these implicitly declared string variables, those followed by one or two subscripts are string arrays whose subscripts can range in value from zero or one to ten. A subscripted string variable refers to the element in the 1- or 2-dimensional array selected by the value(s) of the subscript(s). A string variable with no subscripts is a simple variable.

The substring qualifier specifies a portion of the value associated with a string variable. <string-variable> (<m>:<n>) specifies that portion of the value associated with <string-variable> from its <m>th to its <n>th characters, inclusive. Characters in a string are numbered from the left starting with one. No errors can occur with substring qualifiers; if either <m> or <n> is not in the range from 1 to LEN(<string-variable>), then <m> is MAX(<m>,1) and <n> is MIN(<n>,LEN(<string-variable>)). If <m> < <n>, even after this adjustment, then <string-variable> (<m>:<n>) is the null string. For example, if A$ = "1234", then A$(1:1) = "1", A$(1:3) = "123", A$(0:3) = "123", A$(2:5) = "234", and A$(3:2) = A$(5:7) = "".

5-2

The initial value of each string variable at execution time is the null string (" ").

Examples of string variables:

K$
R2$(15:20)
A$(4)
B1$(I,J)

STRING ASSIGNMENT STATEMENT

The string assignment statement is used to assign a string value to a list of simple or subscripted variables.

Syntax:
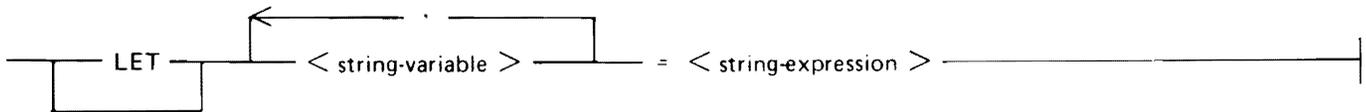
G18013

<string-variable> is described under String Variables in this section. Simple and subscripted variables may appear in the same list of <string-variable>s. <string-expression> is any valid string expression in BASIC.

Semantics:

A string assignment statement is evaluated in the following manner: The subscripts, if any, of variables in the <string-variable> list are evaluated in sequence from left to right. Next, the expression on the right of the equal sign (=) is evaluated. Finally, the value of that expression is assigned to each variable in the <string-variable> list.

The length of a string variable can change as a result of an assignment to a string variable having a substring qualifier. For example, if A$ = "BASIC", assigning "KETBALL" to A$(4:5) results in "BASKETBALL", and subsequently assigning "FOO" to A$(1:5) results in "FOOTBALL".

If the assignment of a string to a string variable causes an overflow of that variable, which means that more characters have been assigned to the variable than are allowed, an error message is displayed and program execution terminates.

Examples of string assignment statements:

LET A$ = "ABC"	! String constant ABC is assigned to A$.
C$(I) = B$	! Ith element of array C$ gets contents of B$.
A$,B$ = "NEGATIVE"& "DISCRIMINANT"	! A$ and B$ get the value of the string ! expression.
D$(2,3)(1:6) = "CHANGE"	! Bytes 1 through 6 of element (2,3) of array ! D$ get the value of the string expression.

STRING EXPRESSIONS

A string expression is any string constant, string variable, string function reference, or concatenation of these.

Syntax:

G18014

<string-constant> and <string-variable> are defined in this section under String Constants and String Variables, respectively. <string-function-ref> includes the intrinsic string functions as defined under Intrinsic String Functions in this section, and the user-defined string functions defined in Section 8 under User-Defined Functions. The ampersand character (&) signifies concatenation.

Semantics:

Concatenation is the joining of the end of one string to the beginning of another.

If the result of a string operation is longer than 255 characters, a fatal error occurs.

Examples of string expressions:

```
        "SOONER OR"      ! String constants are string expressions.


        X$(1,3)          ! String variables are string  expressions.


        A$ & B$          ! Following text gives explanation.

        B$ & A$          ! Following text gives explanation.

        A2$ & B$ & "223" ! Concatenation of two string variables
                         ! with a string constant.
```

The third example, A$ & B$, shows the use of the concatenate operator (&). Assume that A$ contains the constant "COME " and that B$ contains the constant "IN". A$ & B$ gives "COME IN".

For the fourth example, B$ & A$, assume that A$ and B$ contain the same values as before. B$ & A$ gives "INCOME ".

Example:

```
A$  =  "FIRST ---"
B$  =  "SECOND ---"
C$  =  A$ & B$ & "THIRD"
PRINT C$
```

Execution of this example gives the following output:

FIRST ---SECOND ---THIRD

INTRINSIC STRING AND STRING-RELATED FUNCTIONS

Predefined functions are supplied as part of the IBASIC system for the evaluation of commonly used string-valued functions and numeric-valued functions whose arguments (parameters passed to a function) are strings. The general syntax for the intrinsic string functions follows.

Syntax:
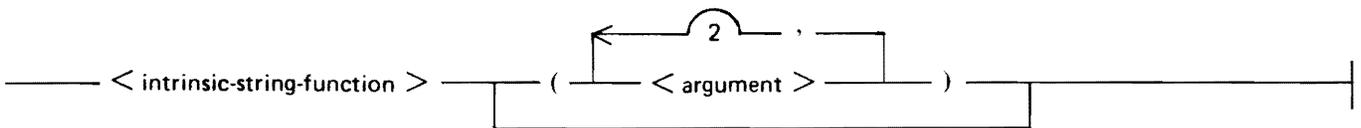
G18015

The number of <argument>s depends on the function.

Semantics:

A description of each of the <intrinsic-string-function>s follows. In each of the descriptions, M represents an index, that is, the rounded integer value of some numeric expression; X represents a numeric expression, and A$ and B$ represent string expressions.

The result of evaluating a string function is a character string which replaces the <string-function-ref> in the string expression.

CHR$(M)

The 1-character string consisting of the character occupying ordinal position M in the collating sequence for the declared character set. (Refer to Tables E-1 and E-2 for the possible collating sequences). M must be greater than zero and less than the number of characters in the declared character set.

Example:

```
OPTION COLLATE STANDARD
PRINT "THE 53RD CHARACTER IS";CHR$(53)
PRINT "THE 65TH CHARACTER IS";CHR$(65)
```

Execution of this partial program causes the following to be displayed.

```
THE 53RD CHARACTER IS 5
THE 65TH CHARACTER IS A
```

DATE$

The date in the string representation YY/MM/DD.

Example:

    PRINT DATE$

If the date was May 17, 1980, this example would give the following output:

    80/05/17

LEN(A$)

The number of characters in the value associated with A$.

Example:

```
READ A$,B$,C$,D$
PRINT "A$  =  ";A$;" LENGTH OF A$ =";LEN(A$)
PRINT "B$  =  ";B$;" LENGTH OF B$ =";LEN(B$)
PRINT "C$  =  ";C$;" LENGTH OF C$ =";LEN(C$)
PRINT "D$  =  ";D$;" LENGTH OF D$ =";LEN(D$)
DATA ABC, DEFGH,IJKLMNOPQRST,U V W X Y Z     !DATA statement is
                                             !described in
                                             !Section 9.
```

Execution of the above example causes the following to be displayed.

```
A$  =  ABC LENGTH OF A$  =  3
B$  =  DEFGH LENGTH OF B$  =  5
C$  =  IJKLMNOPQRST LENGTH OF C$  =  12
D$  =  U V W X Y Z LENGTH OF D$  =  11
```

LWRC$(A$)

The lower-case equivalent of the value of A$.

Example:

```
A$  =  "ABCDEF123"
PRINT LWRC$(A$)
```

Execution of this example gives the following output:

    abcdef123

ORD(A$)

The ordinal position of the character associated with A$ in the collating sequence of the declared character set, where the first member of the character set is in ordinal position zero. The acceptable values of A$ are the single character graphics of the character set and the 2- and 3-character mnemonics of the character set. The acceptable values for the character sets are shown in Tables E-1 and E-2. If the character associated with A$ is not an acceptable value, a fatal error occurs.

Example:

```
OPTION COLLATE STANDARD
PRINT "THE ORDINAL POSITION OF BS IS";ORD("BS")
PRINT "THE ORDINAL POSITION OF A IS";ORD("A")
PRINT "THE ORDINAL POSITION OF 5 IS";ORD("5")
PRINT "THE ORDINAL POSITION OF SOH IS";ORD("SOH")
```

Execution of the above example produces the following output:

```
THE ORDINAL POSITION OF BS IS 8
THE ORDINAL POSITION OF A IS 65
THE ORDINAL POSITION OF 5 IS 53
THE ORDINAL POSITION OF SOH IS 1
```

POS(A$,B$)

The character position in A$ where B$ occurs. The leftmost character in A$ is position 1. If B$ does not occur within A$, then POS(A$,B$) is 0. POS(A$,"") is 1 unless A$ is the null string, in which case POS(A$,"") is 0.

Example:

```
PRINT "CD OCCURS IN ABCDE AT POSITION"; POS("ABCDE","CD")
A$ = "FOUR SCORE AND SEVEN YEARS AGO..."
PRINT "R OCCURS IN A$ AT POSITION";POS(A$,"R")
```

Execution of this example causes the following to be output.

```
CD OCCURS IN ABCDE AT POSITION 3
R OCCURS IN A$ AT POSITION 4
```

POS(A$,B$,M)

Same as the previous POS function except that M-1 positions are skipped before the scan of A$ begins; that is, scanning begins with the Mth character. The value of the function is $M' - 1 + POS(A\$(M:LEN(A\$)),B\$)$, where M' equals MAX(1,MIN(M,LEN(A$))), and A$(M:LEN(A$)) signifies the Mth character position in A$ through the end.

Example:

```
LET A$ = "GRANDSTANDING"
PRINT POS(A$,"AN",1)
PRINT POS(A$,"AN",4)
PRINT POS(A$,"AN",9)
```

Execution of this example produces the following output:

```
3
8
0
```

STR$(X)

The character string which is the numeric representation of the value associated with X. No leading or trailing spaces are included in the character string.

Example:

    PRINT STR$(123.5)
    PRINT STR$(-3.14)

Execution of this example gives the following output:

    123.5
    -3.14

TIME$

The time of day in 24-hour notation. For example, the value of TIME$ at 3:15 PM is 15:15:00.

UPRC$(A$)

The upper-case equivalent of the value of A$.

Example:

    A$ = "abcdef123"
    PRINT UPRC(A$)

Execution of this example gives the following output:

    ABCDEF123

VAL(A$)

Performs the inverse of STR$; has the value of the number associated with A$ if the string associated with A$ is a number. Leading and trailing spaces in the string are ignored. If the evaluation of the number results in a value which causes an underflow, the value returned is zero. If an overflow occurs, the largest possible value is supplied. In either case execution continues.

Examples:

    PRINT VAL(" 123.5 ") * 10
    PRINT VAL("-3.14") * 10
    PRINT VAL("2.E-99")

Execution of these statements gives the following output:

    1235
    -31.4
    error 15 - numeric underflow
    0

STRING DECLARATIONS

String declarations may be used to specify the maximum number of characters allowed in a string variable and/or to specify the character set to be used in a program unit. There are two statements used in string declarations: the DIM statement and the OPTION statement.

DIM Statement String Size Declaration

The DIM statement may be used to declare a maximum size for a string variable. The necessary syntax follows. Other uses of the DIM statement are described in Section 6.

Syntax:

```
           ┌───────────── , ─────────┐
────── DIM ─┴─ < string-variable > * < integer > ─┴──────────────────────┤
G18016
```
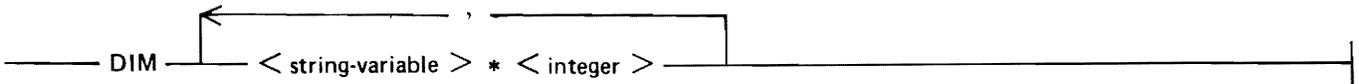
< string-variable > is defined under Variables in this section. < integer > is any integer from 0 to 255 inclusive, and specifies the maximum number of characters allowed in < string-variable >. < string-variable > can appear only once in a size declaration in a program.

Examples of size declarations:

DIM A$*7   ! Max size for A$ is 7.

DIM B1$*80, Z$*10   ! Max size for B1$ and Z$ are 80
                      ! and 10, respectively.

OPTION Statement For Strings

The OPTION statement may be used to specify which character set is used in a program unit. (Refer to Section 6 for descriptions of other uses of the OPTION statement.) The syntax for the OPTION statement when used with strings follows.

Syntax:

```
────── OPTION ────── COLLATE ──┬── NATIVE ────┬──────────────────────────┤
                               └── STANDARD ──┘
G18017
```

Semantics:

The COLLATE option identifies the collating sequence to be used within a program unit for comparing strings. OPTION COLLATE NATIVE specifies that the native collating sequence on the computer be used. For the B 1000 systems the native collating sequence is EBCDIC. Table E-2 contains the EBCDIC character set. OPTION COLLATE STANDARD specifies that the collating sequence ISO International Reference Version be used. Table E-1 contains the ISO character set. The default character set is the STANDARD character set.

SECTION 6
ARRAYS

Arrays are indexed collections of numbers or strings. The indices, referred to as subscripts in this manual, are used to reference one of the elements of the array. Array elements can be manipulated one at a time or an entire array can be manipulated by MAT statements. The keyword MAT derives from the word matrix. A matrix is a 2-dimensional array, but is used in this manual to denote both 1- and 2-dimensional arrays.

ARRAY DECLARATIONS

Array declarations may be used to specify the dimension bound(s) of an array. There are two statements used to declare array dimension bounds: the DIM statement and the OPTION statement.

DIM Statement Array Size Declaration

The DIM statement is used to specify the upper-dimension bound(s) of the subscript(s) of an array and/or to limit the maximum length of a string variable. The use of the DIM statement with string variables is described under DIM Statement String Size Declaration. The syntax for the DIM statement when used with arrays follows.

Syntax:
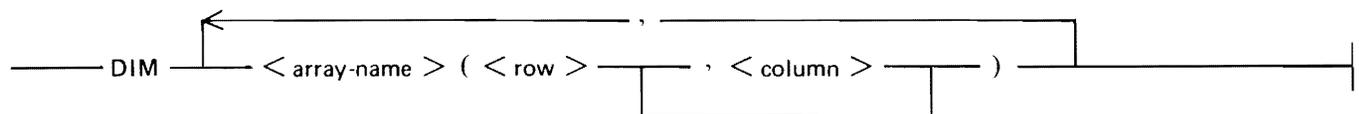
G18018

< array-name > follows the same formation rules as a simple variable name; that is, a letter, optionally followed by a digit, followed by a dollar sign ($) if the array is to contain string data. < row > and < column > must be nonnegative integers. If both < row > and < column > are present, they must be greater than or equal to the base for arrays in the program. Refer to the OPTION statement in this section for more information on the base for arrays.

The declaration for an array, if present, must occur in a lower-numbered line than any reference to that array or to one of its elements. An array may be dimensioned only once in a DIM statement in each program.

Semantics:

Each array declaration occurring in a DIM statement declares the array named to be either 1- or 2-dimensional, according to whether one or two < integer >s are specified within the parentheses following the array name. These < integer >s specify the upper bounds of the array, the maximum values that subscripts for the array may have. There may be a maximum of two subscripts for any array. The first subscript represents the rows, the second represents the columns.

If the declaration for a string array in a DIM statement contains a string size declaration as described under DIM Statement String Size Declaration in Section 5, the maximum length of any character string associated with any element of that string array is the specified value.

Arrays that are not declared in any DIM statement are declared implicitly to be 1- or 2-dimensional according to their use in the program: if one subscript appears in an array reference, the array is 1-dimensional; if two subscripts appear, the array is 2-dimensional. If no subscripts appear in the array reference, the undeclared array is assumed to be 2-dimensional instead of 1-dimensional. The upper-dimension bound on implicitly-declared arrays is 10. The lower-dimension on arrays is 0 unless the BASE option in an OPTION statement has declared it to be 1. Refer to the OPTION Statement for Arrays in this section for more information on the BASE option.

Examples of DIM statements:

OPTION BASE 1	! Lower-dimension bound is 1.
DIM A(6), B(9,10)	! A is 1-dimensional with 6 elements. ! B is 2-dimensional with 9*10 elements. ! Refer to Figure 6-1. Element ! (8,2) is marked.
DIM A$(4,4), B$(100)*5	! A$ is 2-dimensional with 4*4 ! elements. B$ is 1-dimensional ! with 100 elements. Maximum ! element size is 5.
DIM P$(10,10)*10, C(50,2)	! String array P$ is 2-dimensional ! with 10*10 elements. Maximum ! element size is 10. Numeric ! array C is 2-dimensional with ! 50*2 elements.

G18019

Element (8,2) is marked.

Figure 6-1. Representation of a 9 by 10 Array (OPTION BASE 1)

OPTION Statement For Arrays

The OPTION statement may be used to change the value of the lower-dimension bounds for arrays in a program. The syntax follows.

Syntax:

```
———— OPTION ———— BASE ——┬—— 0 ——┐——————————————————————————————————————┤
                         └—— 1 ——┘
```

G18020

Semantics:

If an OPTION statement specifies that the lower bound for array subscripts is 1, then no DIM statement in the program can specify an upper bound of 0.

The OPTION statement with a BASE option, if present, must occur in a lower-numbered line than any DIM statement or any reference to an array in the program. Only one BASE option may occur in a program. If no OPTION BASE occurs in the program, the base defaults to zero.

Examples of OPTION statements:

OPTION BASE 1	!Lower dimension bound for arrays is 1.
OPTION COLLATE NATIVE, BASE 1	!Collating sequence used in string comparisons ! is EBCDIC. Lower-dimension bound is 1.

NUMERIC ARRAY MANIPULATION

BASIC provides several statements to manipulate an entire array, rather than just one array element at a time. The statements available for numeric array manipulation are explained in this subsection. These statements follow normal rules of matrix algebra.

In all numeric array manipulation statements, overflow and underflow are reported to the user and program execution continues. With overflow, the value $7.237E + 75$ is substituted. With underflow, zero is substituted.

MAT Addition Statement

The purpose of the MAT addition statement is to add two numeric arrays and to assign the sum to a third array. The syntax follows.

Syntax:

```
———— MAT < array-name1 > = < array-name2 > + < array-name3 > ——————————————————————┤
```
G18021

<array-name1>, <array-name2>, and <array-name3> follow the same formation rules as a simple variable name.

Semantics:

<array-name1> is assigned the sum of <array-name2> and <array-name3>. If <array-name1> does not have the same dimensions as <array-name2> and <array-name3>, it may be redimensioned. Rules for redimensioning are described under the MAT assignment statement in this section. <array-name2> and <array-name3> must have the same dimensions. <array-name1> and <array-name2> may be the same array.

Examples of the use of the MAT addition statement:

```
OPTION BASE 1
DIM A(3,3) B(3,3), C(3,3)
MAT READ A,B
DATA 1,2,3
DATA 4,5,6
DATA 7,8,9
DATA 1,4,7
DATA 2,5,8
DATA 3,6,9
PRINT "ARRAY A IS"
MAT PRINT A; MAT PRINT statement - Section 9
PRINT
PRINT "ARRAY B IS"
MAT PRINT B;
PRINT
PRINT "ARRAY C IS"
MAT C = A + B
MAT PRINT C;
```

Execution of the above example causes the following to be displayed.

```
ARRAY A IS
  1  2  3
  4  5  6
  7  8  9

ARRAY B IS
  1  4  7
  2  5  8
  3  6  9

ARRAY C IS
  2  6  10
  6  10  14
  10  14  18
```

MAT Assignment Statement

The purpose of the MAT assignment statement is to move the elements of one array to the elements of another array.

Syntax:

```
———————— MAT  < array-name1 >  =  < array-name2 > ——————————————————————————————————|
G18022
```

<array-name1> and <array-name2> follow the same naming conventions as simple numeric variables.

Semantics:

<array-name1> and <array-name2> must have the same number of dimensions but not necessarily the same upper bounds on those dimensions. If the upper-dimension bounds are different, <array-name1> may be redimensioned to match <array-name2>. This is known as dynamic redimensioning. Dynamic redimensioning takes place only if (1) the original total number of elements in <array-name1> is greater than or equal to the total number of elements in <array-name2> and (2) the number of dimensions of <array-name1> and <array-name2> are the same. Otherwise, a fatal error occurs.

When a numeric array is redimensioned dynamically, the current upper bound for each subscript is changed to match the size of its new value and the current lower bound for each subscript stays the same.

Example of a numeric array assignment statement with redimensioning:

```
DIM B(4,4)
MAT READ A(2,2)
DATA 1,2,3,4
MAT B = A
MAT PRINT B;
```

Execution of the example above causes the following to be displayed.

```
1  2
3  4
```

MAT CON Statement

The MAT CON statement initializes all of the data elements of an array to the numeric constant 1 or to the value of a numeric expression. The MAT CON statement may also be used to redimension an array. The syntax of the MAT CON statement follows.

Syntax:

```
──────── MAT < array-name > = ──┬──┬──── ( < numeric-expression > ) ──── * ──┬──────────────────→
                                │  └──────────────────────────────────────┘ │
                                │
>──── CON ──┬──┬──── ( < subscript > ──┬──── · < subscript > ───┬── ) ──┬──────────────────┤
            │  │                       └───────────────────────┘       │
            │  └───────────────────────────────────────────────────────┘
```
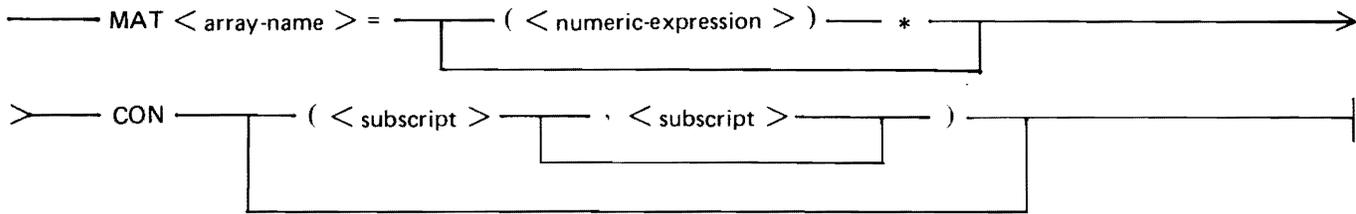
G18023

<array-name> follows the same naming conventions as simple numeric variables. <numeric-expression> is described under Numeric Expressions in Section 4.

Semantics:

If present, the <numeric-expression> is evaluated and used in place of the numeric constant 1 to initialize each element of the array.

The <subscript>s must be greater than or equal to the lower-dimension bound for arrays in the program unit where the MAT CON statement occurs. If one or both of the <subscript>s are present, a redimension is implied. Arrays are redimensioned in the same manner as described for the MAT assignment statement in this section.

Example of the use of the MAT CON statement:

```
OPTION BASE 1
DIM A(3,5)
MAT A = CON
MAT PRINT A;
PRINT
MAT A = (2) * CON(3,3)
MAT PRINT A;
```

Execution of this example gives the following output:

```
1  1  1  1  1
1  1  1  1  1
1  1  1  1  1

2  2  2
2  2  2
2  2  2
```

DOT Function

The DOT function produces the dot product of two 1-dimensional arrays.

Syntax:

```
──────── DOT ( < array-name > · < array-name > ) ────────────────────────────────────┤
```
G18024

<array-name>s follow the same formation rules as a simple numeric variable and must be singly subscripted numeric arrays.

Semantics:

The dot product is the sum of the products of each of the corresponding elements of the arrays.

Example of the use of the DOT function:

```
OPTION BASE 1
DIM A(2), B(2)
DATA 2,3,4,3
MAT READ A, B
PRINT DOT(A,B)
```

Execution of this example produces the following output:

```
17
```

MAT IDN Statement

The MAT IDN statement zeros a square array and places the integer 1 or the value of a numeric expression in each element on the main diagonal.

Syntax:
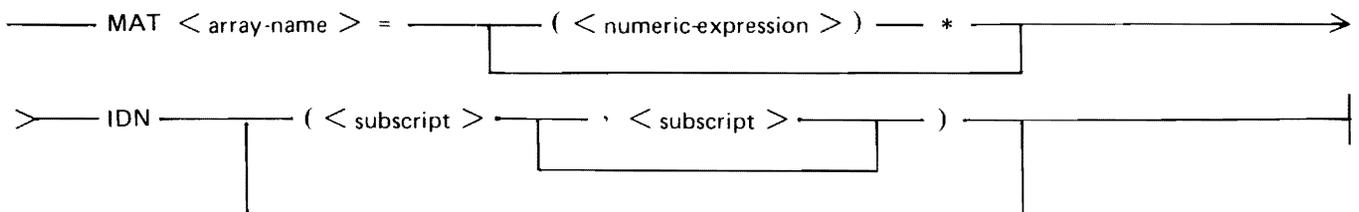
G18025

<array-name> follows the same naming conventions as simple numeric variables. <array-name> must be a square array or must be redimensioned to be a square array. In a square array, the upper-dimension bounds are equal to each other. <numeric-expression> is described under Numeric Expressions in Section 4.

Semantics:

<numeric-expression>, if present, is evaluated and used as the value for each element of the array instead of the integer 1. The <subscript>s must be greater than or equal to the lower-dimension bound for arrays in the program unit where the MAT IDN statement occurs. The subscripts, if present, imply a redimension of <array-name>.

Examples of MAT IDN statements:

```
OPTION BASE 1
DIM X(3,3)
MAT X=IDN
MAT Y=(10)*IDN(4,4)
MAT PRINT X; Y;
```

Execution of this example causes the following to be displayed.

```
1  0  0
0  1  0
0  0  1

10  0   0   0
 0  10  0   0
 0  0   10  0
 0  0   0   10
```

MAT INV Function

The MAT INV function produces the inverse of a square array. The syntax of the MAT INV function follows.

Syntax:

```
—— MAT —— < array-name1> —— = —— INV —— (< array-name2>) —|
```

< array-name2 > must be a square array. Both < array-name1 > and < array-name2 > must follow the same formation rules as used for simple numeric variables.

Example of the MAT INV function:

    MAT A = INV(B)

Assume that array B contains the following values:

```
1  2
3  7
```

Execution of this example results in array A receiving the following values:

```
 7  -2
-3   1
```

MAT Multiplication Statement

The purpose of the MAT multiplication statement is to multiply two numeric arrays and to assign the product to a third array.

Syntax:

```
——— MAT < array-name1 > = < array-name2 > * < array-name3 > ———————————————|
```
G18026

< array-name1 >, < array-name2 >, and < array-name3 > follow the same formation rules as a simple variable name.

Semantics:

<array-name1> is assigned the product of <array-name2> and <array-name3>. If <array-name1> does not have the row dimension of <array-name2> and the column dimension of <array-name3>, it may be redimensioned. Rules for redimensioning are described under the MAT assignment statement in this section. The column dimension of <array-name2> must be the same as the row dimension of <array-name3>. <array-name1>, <array-name2>, and <array-name3> may be the same array.

When two arrays are multiplied, the dot product of the first row of the first array and of each column of the second array forms the first row in the answer; the dot product of the second row of the first array and each column of the second array forms the second row in the answer, and so on.

Example of the MAT multiplication statement:

```
OPTION BASE 1
DIM C(2,2)
MAT READ A(2,3), B(3,2)
DATA 1,2,3,4,5,6
DATA 1,2,3,4,5,6
MAT C = A * B
PRINT "MAT A ="
MAT PRINT A;
PRINT
PRINT "MAT B ="
MAT PRINT B;
PRINT
PRINT "MAT C ="
MAT PRINT C;
```

Execution of this example causes the following to be displayed.

```
MAT A =
 1  2  3
 4  5  6

MAT B =
 1  2
 3  4
 5  6

MAT C =
 22  28
 49  64
```

MAT Scalar Multiplication Statement

The MAT scalar multiplication statement allows each element of an array to be multiplied by any scalar number.

Syntax:

```
────── MAT < array-name > = ──┬──< sign >──┬── ( < numeric-expression > ) * < array-name > ──────┤
                              └────────────┘
```

G18027

< array-name > follows the same naming conventions as simple numeric variables. The two < array-name >s may name the same array. < numeric-expression > is described under Numeric Expressions in Section 4.

Example of the use of MAT scalar multiplication statements:

```
OPTION BASE 1
DIM A(3,3),B(3,3)
MAT READ A
DATA 1,2,3,4,5,6,7,8,9
PRINT "ARRAY A IS"
MAT PRINT A;
PRINT
PRINT "ARRAY B IS"
MAT B = - (2) * A
MAT PRINT B;
```

Execution of this example gives the following output:

```
ARRAY A IS
 1  2  3
 4  5  6
 7  8  9

ARRAY B IS
-2 -4 -6
-8 -10 -12
-14 -16 -18
```

MAT Subtraction Statement

The purpose of the MAT subtraction statement is to subtract one array from another array and to assign the difference to a third array.

Syntax:

```
────── MAT < array-name1 > = < array-name2 > < minus-sign > < array-name3 > ───────────────┤
```
G18028

<array-name1>, <array-name2>, and <array-name3> follow the same formation rules as a simple variable name.

Semantics:

<array-name1> is assigned the difference of <array-name2> and <array-name3>. If <array-name1> does not have the same dimensions as <array-name2> and <array-name3>, it may be redimensioned. Rules for redimensioning are described under the MAT assignment statement in this section. <array-name2> and <array-name3> must have the same dimensions. <array-name1> and <array-name2> may name the same array.

Example of the use of the MAT subtraction statement:

```
OPTION BASE 1
DIM A(3,3), B(3,3), C(3,3)
MAT READ A,B
DATA 1,2,3,4,5,6,7,8,9,2,4,6,8,1,3,5,7,9
PRINT "ARRAY A IS"
MAT PRINT A;
PRINT
PRINT "ARRAY B IS"
MAT PRINT B;
PRINT
PRINT "ARRAY C IS"
MAT C = A - B
MAT PRINT C;
```

Execution of this example gives the following output:

```
ARRAY A IS
   1   2   3
   4   5   6
   7   8   9

ARRAY B IS
   2   4   6
   8   1   3
   5   7   9

ARRAY C IS
  -1  -2  -3
  -4   4   3
   2   1   0
```

MAT TRN Function

The MAT TRN function transposes its argument. The syntax of the MAT TRN function follows.

Syntax:

```
—— MAT —— < array-name1> —— = —— TRN —— (< array-name2>) ——|
```

Both <array-name1> and <array-name2> must follow the same formation rules as a simple numeric variable.

Example of the MAT TRN function:

MAT A = TRN(X)

Assume that array X contains the following:

```
7    12    5   -1
1     2    9    3
2    -8    6    4
```

Execution of this example results in array A receiving the following values:

```
  7    1    2
 12    2   -8
  5    9    6
 -1    3    4
```

MAT ZER Statement

The MAT ZER statement initializes the elements of the specified numeric array to the numeric constant 0, and may optionally be used to redimension the array.

Syntax:

G18029

<array-name> follows the same naming conventions as simple numeric variables. <numeric-expression> is described in Section 4.

Semantics:

The <subscript>s must be greater than or equal to the lower-dimension bound for arrays in the program unit where the MAT ZER statement occurs. If one or both of the <subscript>s are present, a redimension is implied. Arrays are redimensioned in the same manner as described for the MAT assignment statement in this subsection. <numeric-expression> has no effect on the execution of this statement.

Examples of the MAT ZER statement:

```
OPTION BASE 1
DIM A(2,10)
MAT A = ZER
MAT PRINT A;
PRINT
MAT A = ZER(4,2)
MAT PRINT A;
```

Execution of this example gives the following output:

```
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 0
   0 0
```

STRING ARRAY MANIPULATION

As with numeric arrays, an entire string array may be operated upon, rather than just one element. The statements available for string array manipulation are described in this subsection.

MAT Assignment Statement

The purpose of the MAT assignment statement is to move the elements of one array to the elements of another array.

Syntax:

```
————————MAT < array-name1 > = < array-name2 > ————————————————————————————————————|
G18022
```

< array-name1 > and < array-name2 > follow the same naming conventions as simple string variables.

Semantics:

< array-name1 > and < array-name2 > must have the same number of dimensions but not necessarily the same upper bounds on those dimensions. If the upper-dimension bounds are different, < array-name1 > is redimensioned to match < array-name2 >. This takes place only if (1) the original total number of elements in < array-name1 > is greater than or equal to the total number of elements in < array-name2 > and (2) the number of dimensions of < array-name1 > and < array-name2 > are equal; otherwise, a fatal error occurs.

When a string array is redimensioned dynamically, the current upper bounds for its subscripts are changed to match the size of its new value and the current lower bounds stay the same.

Example of the use of a string array assignment statement:

```
OPTION BASE 1
DIM Z$(3,4)
DATA AARDVARK,NEXT STRING,123456789,FOURTH,ELEPHANT,TACK
DATA COZUMEL, WRITE RING,"NO CABO.","NO! NO QUEPO."
DATA "TU TAMPOCO?",END
MAT READ Z$
MAT Y$ = Z$
MAT PRINT Y$
PRINT "----*----1----*----2----*----3----*----4----*----5----*"
```

Execution of this example gives the following output:

```
AARDVARK       NEXT STRING    123456789    FOURTH
ELEPHANT       TACK           COZUMEL      WRITE RING
NO.CABO.       NO! NO QUEPO.  TU TAMPOCO?  END
----*----1----*----2----*----3----*----4----*----5----*
```

MAT NUL$ Statement

The MAT NUL$ statement assigns the null string to each element of a string array. MAT NUL$ may also be used to redimension an array.

Syntax:

G18030

<array-name> must follow the same naming conventions as simple string variables.

Semantics:

The <subscript>s must be greater than or equal to the lower-dimension bound for arrays in the program unit where the MAT NUL$ statement occurs. If one or both of the <subscript>s are present, a redimension is implied. Arrays are redimensioned in the same manner as described for the MAT assignment statement in this subsection.

Examples of the MAT NUL$ statement:

```
MAT A$ = NUL$        ! Each element in A$ gets the null
                     ! string.

MAT B$ = NUL$(5,6)   ! Each element is B$ gets the null
                     ! string and the array is redimensioned
                     ! to a 5 by 6 array.
```

SECTION 7
CONTROL STRUCTURES

Normally, the executable statements in a BASIC program are executed in line number sequence; that is, after one statement is executed, the statement immediately following it is executed. Control statements are used to alter the normal flow of a program. They may transfer the control to another part of the program, terminate program execution, or control iterative processes. The control structures available in Burroughs B 1000 BASIC and the expressions that are used in several of these structures are described in this section.

RELATIONAL EXPRESSIONS

Relational expressions enable the values of expressions to be compared in order to influence the flow of control in a program unit.

Syntax:
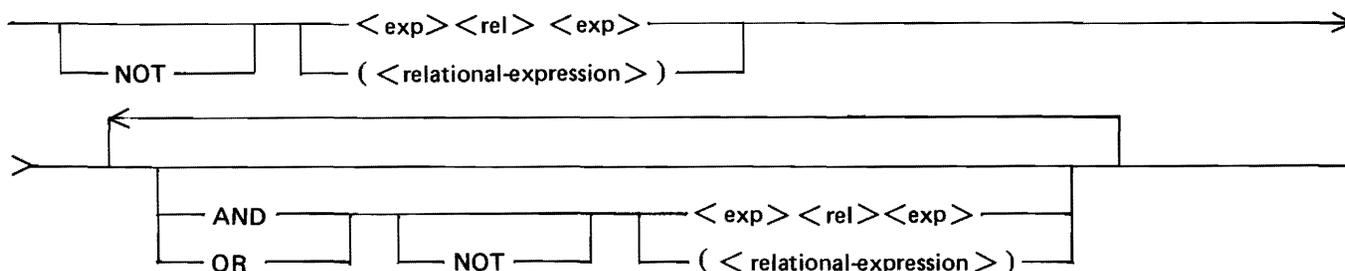
G18031

<exp> is either a string or a numeric expression as described in sections 4 and 5, respectively. <rel> is a relational symbol as listed in table 7-1. The appearance of <relational-expression> within parentheses indicates that a relational expression can be used as an operand in a larger relational expression. This also allows for the nesting of parentheses in a relational expression.

Table 7-1 lists the valid relational symbols.

Table 7-1. Relational Symbols

Symbol(s)	Meaning
=	equal to
< > or > <	not equal to
<	less than
>	greater than
< = or = <	less than or equal to
> = or = >	greater than or equal to

Semantics:

Two numeric expressions are considered equal only if the expressions have the same value. Two string expressions are considered equal only if the values of the two expressions have the same length and contain identical sequences of characters.

In the evaluation of string relational expressions, the relation "less than" means "earlier in the collating sequence than," and "greater than" means "later in the collating sequence than." If two strings of different lengths occur in a relational expression and one is an initial leftmost segment of the other, the shorter string is less than the other. Otherwise, the relationship between two strings of unequal length is determined by the contents of the shorter string and the leftmost portion of the longer string which is the same length as the shorter string.

The following portion of the relational expression syntax is considered a relational term.

Syntax:
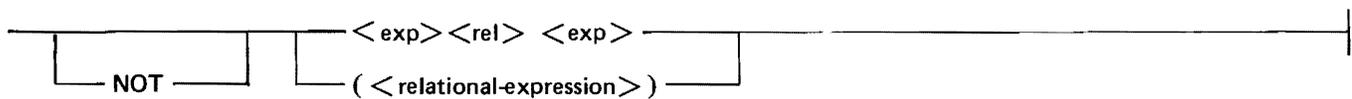
A relational expression involving relational terms connected with the AND operator (conjunction) is TRUE if the value of each relational term is TRUE. A relational expression involving relational terms connected with the OR operator (disjunction) is TRUE if at least one of the relational terms is TRUE. The relational terms of conjunctions and disjunctions are evaluated in sequence from left to right until the truth or falsity of the entire relational expression is determined. A relational term beginning with the NOT operator is TRUE if, and only if, the following portion of the relational term is FALSE.

In the absence of parentheses, the NOT operator is evaluated first, then the AND operator, and finally the OR operator.

Examples of relational expressions:

A < B	!A is less than B.
A > = C	!A is greater than or equal to C.
A < = X AND X < = B	!A is greater than or equal to X and X is less than or !equal to B.
A$ = B$ OR A$ = ""	!A$ equals B$ or A$ equals null string. A<B AND !B<C AND C<D A is less than B and B is less than C !and C is less than D.
I < = 10 AND A(I) < > 0	!I is less than or equal to 10 and A(I) is not equal to 0. !If I is greater than 10, the

X < > 0 OR (C$ = "MULTIPLY" AND NOT D$ = "SUM")

7-2

CONTROL STATEMENTS

Control statements permit the interruption of the normal sequence of execution of statements and cause execution to continue at a specified line, rather than at the line with the next line number in line number sequence.

GOTO Statement

The GOTO statement causes an unconditional transfer of control.

Syntax:

———— GOTO < line-number > ———————————————————————————————————————|
G18032

<line-number> is described under Lines in Section 3. Execution of a GOTO statement causes program execution to continue at the line with the specified <line-number>.

Semantics:

<line-number> must not cause a jump into a FOR NEXT loop (refer to FOR NEXT structure in this section) or a user-defined function.

Examples of GOTO statements:

    GOTO 100
    GO TO 5550

GOSUB and RETURN Statements

The GOSUB and RETURN statements allow for subroutine calls.

Syntax for GOSUB:

———— GOSUB < line-number > ———————————————————————————————————————|
G18033

Syntax for RETURN:

———— RETURN ———|
G18034

<line-number> is described under Lines in Section 3.

Semantics:

Execution of a GOSUB statement causes program execution to continue at the line with the specified <line-number>. <line-number> must not cause a jump into a FOR NEXT loop or a used-defined function.

Execution of a RETURN statement causes execution to continue at the line immediately following the GOSUB statement. GOSUB and RETURN statements may be nested; that is, one set of statements may be contained within another set.

For every RETURN statement executed there must be at least one GOSUB statement for which no RETURN has been executed.

Example of GOSUB and RETURN statements:

```
100 GOSUB 5160
110 REM -- THE RETURN STATEMENT TRANSFERS CONTROL HERE

5160 REM -- THE GOSUB STATEMENT TRANSFERS CONTROL HERE
       .
       .
       .
5300 RETURN
```

ON GOTO Statement

The ON GOTO statement allows control to be transferred to any one of a group of line numbers.

Syntax:
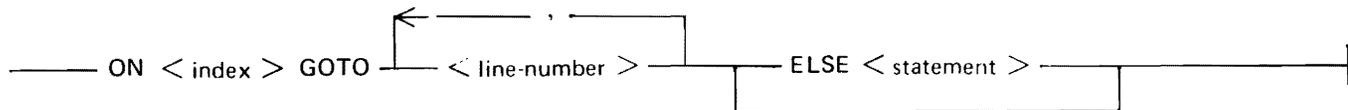
G18035

<index> is a numeric expression as described in Section 4. <index> is rounded to obtain an integer: INT(< index > + 0.5). <line-number> is described under Lines in Section 3. <statement>, if present, can be any BASIC statement except the following:

```
IF
FOR or NEXT
DIM
OPTION
DATA
IMAGE
DEF or FNEND
END
REM
ON
```

Semantics:

When an ON GOTO statement is executed, the <index> is evaluated and used as an index into the list of <line-numbers>. The list of <line-number>s is numbered from left to right, starting with the integer 1. If an ELSE occurs, the <statement> following the ELSE is executed if the value of <index> is less than 1 or greater than the number of <line-number>s in the list. If there is no ELSE clause (ELSE part) and <index> is less than 1 or greater than the number of <line-number>s in the list, a fatal error occurs.

Examples of ON GOTO statements:

```
ON L+1 GOTO 400, 400, 500
ON X GOTO 100, 200, 150, 9999 ELSE LET A = 1
```

In the first statement, if L = 0 or L = 1, line 400 is the next line executed. If L = 2, line 500 is executed next. Any other value for L causes a fatal error.

In the second statement, if X = 1, statement 100 is executed next. If X = 2, statement 200 is executed next. If X = 3, statement 150 is executed next. If X = 4, statement 9999 is executed next. If X is none of these values, variable A is assigned the value 1 and execution continues with the statement following the ON GOTO statement.

ON GOSUB and RETURN Statements

The ON GOSUB and RETURN statements allow control to be transferred to and from any one of a group of subroutines.

Syntax:
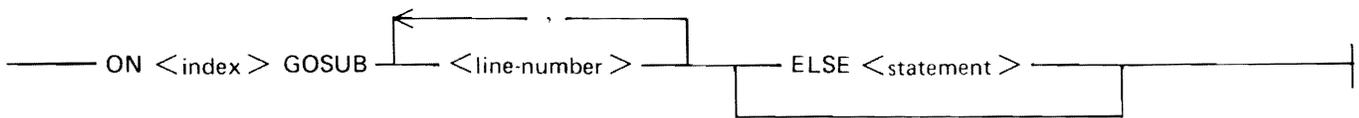
G18036

The syntax for the RETURN statement is the same as listed under GOSUB and RETURN Statements in this section.

<index> is a numeric expression as described in Section 4. <index> is rounded to obtain an integer: INT(< index > + 0.5). <line-number> is described under Lines in Section 3. <statement>, if present, can be any BASIC statement except the following:

    IF
    FOR or NEXT
    DIM
    OPTION
    DATA
    IMAGE
    DEF or FNEND
    END
    REM
    ON

Semantics:

When an ON GOSUB statement is executed, the <index> is evaluated and used as an index into the list of <line-numbers>. The list of <line-number>s is numbered from left to right, starting with the integer 1. If an ELSE occurs, the <statement> following the ELSE is executed if the value of <index> is less than 1 or greater than the number of <line-number>s in the list. If there is no ELSE clause and <index> is less than 1 or greater than the number of <line-number>s in the list, a fatal error occurs.

Examples of ON GOSUB and RETURN statements:

    ON A+7 GOSUB 1000, 2000, 7000, 4000
    ON F1-2 GOSUB 4360, 4460, 4660 ELSE PRINT F1

In the first statement, if A = −6, line 1000 is executed next. If A = −5, line 2000 is executed next. If A = −4, line 7000 is executed next, and if A = −3, line 4000 is executed next. Any other value for A causes a fatal error.

In the second statement, if F1 = 3, line 4360 is executed next. If F1 = 4, line 4460 is executed next. If F1 = 5, line 4660 is executed next. If F1 is none of these values, variable F1 is displayed and execution continues with the statement following the ON GOSUB statement.

LOOP STRUCTURES

Loop structures provide for the repeated execution of a sequence of statements. There are two loop structures available in B 1000 IBASIC: the FOR NEXT structure and the DO LOOP structure.

FOR NEXT Structure

The FOR NEXT structure provides for the construction of counter-controlled loops.

Syntax for the FOR statement:

```
───── FOR <control-var> = <initial-value> TO <limit> ──┬── STEP <increment> ──┬─────────┤
                                                        └──────────────────────┘
```
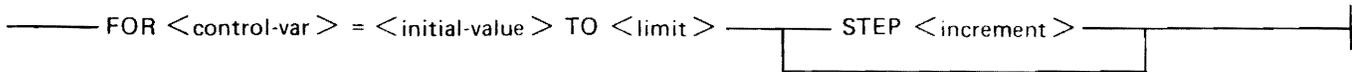
G18037

Syntax for the NEXT statement:

```
───── NEXT <control-var> ──────────────────────────────────────────────────────┤
```
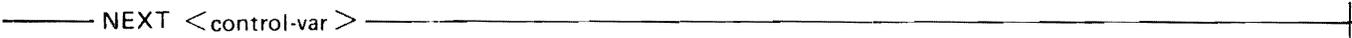
G18038

The FOR and NEXT statements must occur in corresponding pairs. <control-var> is a simple numeric variable, and must be the same variable in corresponding FOR and NEXT statements. <initial-value>, <limit>, and <increment> are numeric expressions. If the STEP clause is omitted, <increment> defaults to 1.

There may be any number of BASIC statements between corresponding FOR and NEXT statements. These statements, plus the FOR and NEXT statements, comprise a FOR block. A FOR block may contain any BASIC statement except an END statement.

Semantics:

The FOR NEXT structure allows a group of statements to be executed a specified number of times. The action of the FOR NEXT structure can be defined in terms of other BASIC statements as follows.

FOR NEXT structure:

```
10 FOR C = I1 TO L STEP I2
   (BASIC statements)
100 NEXT C
```

Equivalent BASIC statements:

```
10 LET X1  =  L
20 LET X2  =  I2
30 LET C  =  I1
40 IF (C – X1) * SGN(X2)  >  0 THEN GOTO 100
        (BASIC statements)
80      LET C  =  C  +  X2
90 GOTO 40
100 REM -- REMAINDER OF PROGRAM UNIT
```

Within the body of the FOR NEXT structure the value of <control-var> may be changed. Any such change may affect the number of times that the body is executed. The numeric expressions <initial-value>, <limit>, and <increment> may also be changed in the body if simple numeric variables are used for these expressions, but changes to these variables do not affect the number of times that the body is executed.

FOR NEXT structures may be nested, but nested FOR NEXT structures must use different <control-var>s. When nesting is used, the innermost structure must be completely terminated before any of the outer structures are terminated.

A line number within the body of a FOR NEXT structure cannot be referred to outside of that structure by a GOTO, GOSUB, ON, or IF statement.

Examples of FOR NEXT structures:

```
FOR I  =  1 TO 10
    LET A(I)  =  I
NEXT I

FOR C7  =  A TO B STEP  – 1
    C$(C7)  =  D$
    PRINT C$(C7)
NEXT C7
```

Example of the use of nested FOR NEXT structures:

```
OPEN #1: "INFILE", INTERNAL, INPUT OPEN statement  –  Section 9
DIM A(100,100)
FOR I  =  1 TO 100
    FOR J  =  1 TO 100
        INPUT #1: A(I,J)
    NEXT J
NEXT I
```

The first FOR NEXT structure initializes elements one through ten of array A to the values 1 through 10.

The second FOR NEXT structure is similar to the first in that an array is being initialized, but in this example, the initial and limiting values are variables instead of constants as in the first example. This example also shows the use of the STEP clause with a negative increment.

The example of the nested FOR NEXT structures shows how a 2-dimensional array, A, can be easily loaded from a disk file. The description of statements used with disk files is in Section 9.

DO LOOP STRUCTURE

The DO LOOP structure provides for the repeated execution of a sequence of statements with arbitrary exit conditions.

Syntax for the DO statement:

```
DO ─┬────────────────────────────────────────────────────────────┤
    ├── WHILE ──┬── <relational-expression> ──┘
    └── UNTIL ──┘
```
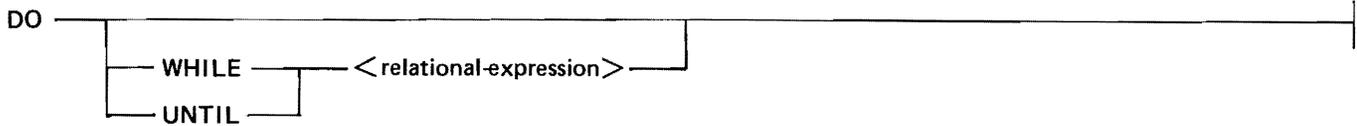
Syntax for the LOOP statement:

```
LOOP ─┬──────────────────────────────────────────────────────────┤
      ├── WHILE ──┬── <relational-expression> ──┘
      └── UNTIL ──┘
```
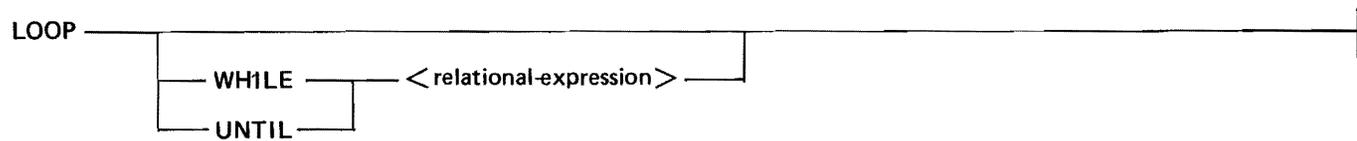
The DO and LOOP statements must occur in corresponding pairs. <relational-expression> is any relational expression as described in this section.

Any number of BASIC statements can appear between the corresponding pairs of DO and LOOP statements, including any number of EXIT statements. The EXIT statement provides a method for exiting the loop structure on a specified exit condition.

Syntax for the EXIT statement:

```
EXIT IF  <relational-expression> ─────────────────────────────────┤
```

<relational-expression> is any relational expression as described in this section.

Semantics:

The DO LOOP structure provides for repeated execution of a group of BASIC statements until an exit condition requires an exit of the loop structure. An exit condition requires an exit from the loop in any one of the following cases:(1) the value of the <relational-expression> following the WHILE keyword is FALSE, (2) the value of the <relational-expression> following the UNTIL keyword is TRUE, or (3) the value of the <relational-expression> following the EXIT IF keywords is TRUE.

If the execution of a program reaches a DO statement, the <relational-expression> in that statement, if present, is evaluated. If there is no <relational-expression> or if the <relational-expression> does not require an exit from the loop, execution proceeds to the next line. If the <relational-expression> requires an exit from the loop, execution continues at the line following the associated LOOP statement.

If the execution of a program reaches an EXIT statement, the <relational-expression> in that statement is evaluated. If the <relational-expression> does not require an exit from the loop, execution proceeds to the next line. If the <relational-expression> requires an exit from the loop, execution continues at the line following the associated LOOP statement.

If the execution of a program reaches a LOOP statement, the <relational-expression> in that statement, if present, is evaluated. If there is no <relational-expression> or if the <relational-expression> does not require an exit from the loop, execution proceeds to the associated DO statement. If the <relational-expression> requires an exit from the loop, execution continues at the line following the associated LOOP statement.

Examples of DO LOOP structures:

```
DO UNTIL I = 72
   A$(I:I) = "*"
   I = I + 1
LOOP

DO
    INPUT PROMPT "ENTER INTEGER BETWEEN 1 AND 100": X
    EXIT IF 1 < = X AND X < 101 AND X = INT(X)
    PRINT "TRY AGAIN."
LOOP
```

DECISION STRUCTURES

Decision structures permit the conditional execution of statements. There are three decision structures available in B 1000 IBASIC: 1) the IF statement, 2) the block IF structure, and 3) the SELECT CASE structure. These decision structures are described in the following paragraphs.

IF Statement

The IF statement permits conditional transfer of control or conditional execution of a statement. The syntax and semantics of the IF statement follow.

Syntax:

G18039

<rel-exp> is a relational expression as described under Relational Expressions in this section. <line-no1> and <line-no2> represent line numbers which refer to lines in the same program unit. <stmt1> and <stmt2> are any BASIC statements except the following:

IF
FOR or NEXT
DIM
OPTION
DATA
IMAGE
DEF or FNEND
END
REM
ON

Semantics:

When the IF statement is executed, < rel-exp> is evaluated. If the expression is TRUE, < stmt1> is executed or control is transferred to < line-no1>. If the expression is FALSE, < stmt2> is executed or control is transferred to < line-no2>.

Examples of IF statements:

```
IF A < B THEN 100
IF A$ < > B$ THEN 250
IF A > 1 AND A < 2 THEN 100 ELSE 200
IF X = > Y2 THEN GOSUB 900 ELSE GOSUB 2000
IF X$ < > "NO" OR X$ = "STOP" THEN LET A = 1
IF A$(3:7) = "CAT" THEN 1000 ELSE PRINT "ERROR #105"
   IF X < > 0 OR (C$ = "MULTIPLY" AND NOT D$ = "SUM") THEN 330
```

BLOCK IF Structure

The block IF structure permits the conditional execution of a sequence of statements. The syntax and semantics of the block IF structure follow.
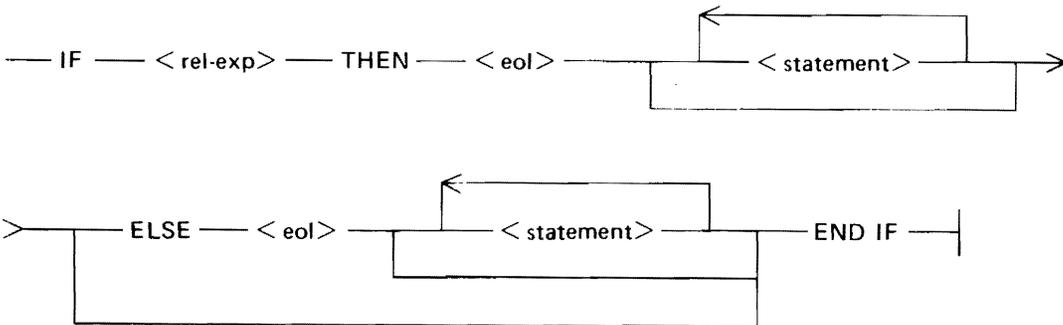
Syntax:

< rel-exp> is a relational expression as described under Relational Expressions in this section. < eol> represents end-of-line and signifies that any subsequent items must begin on a new line. < statement> is any BASIC statement. If < statement> is a BASIC statement that must be accompanied by another BASIC statement in order to be complete, then the accompanying statement must occur before the ELSE statement, if present, or before the END IF statement if there is no ELSE statement.

Line numbers in a statement outside the block IF statement must not refer to a statement within the block IF group.

NOTE
This construct requires several lines for entry. Each new line must begin with a line number and only one statement can occur on a line. For example, END IF must be entered on a separate line from the preceding statement and must have its own line number.

Semantics:

If the value of <rel-exp> in the block IF statement is TRUE, the block of statements following the THEN is executed. If the value of <rel-exp> is FALSE, the block of statements following the ELSE, if present, is executed. After the appropriate THEN block or ELSE block has been executed, execution continues at the line following the END IF statement.

Example of the block IF structure:

```
IF X = INT(X) THEN          !X is tested for an integer value.
    PRINT X; "IS AN INTEGER."    !If X is not an integer, that is,
    A = A + 1               !a real value, the statements
ELSE                        !following ELSE are executed;
    PRINT X; "IS NOT AN     !otherwise,the statements follow-
    INTEGER."               ! ing THEN are executed.
    A = A - 1
END IF
```

Select Case Structure

The SELECT CASE structure permits the conditional execution of any one of a number of alternative sequences of lines. The syntax and semantics of the SELECT CASE structure follow.

Syntax:
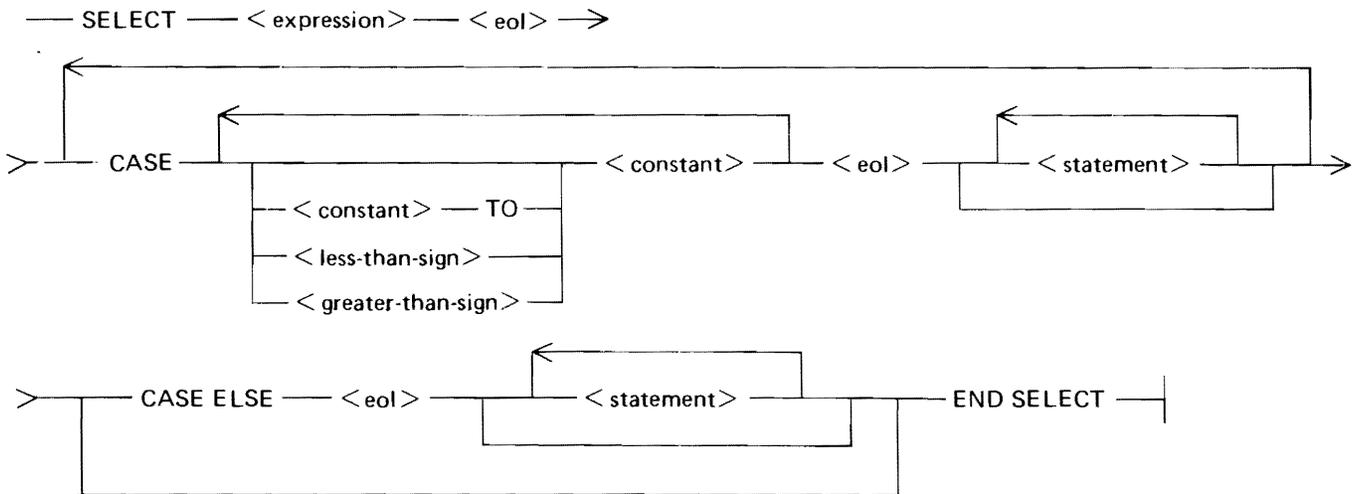
<expression> is either a numeric expression as described under Numeric Expressions in Section 4, or a string expression as described under String Expressions in Section 5. <eol> represents end-of-line and signifies that any subsequent items must begin on a new line. <constant> is either a numeric constant as described under Numeric Constants in Section 4 or a string constant as described under String Constants in Section 5. <statement> is any BASIC statement. If <statement> is a BASIC statement that must be accompanied by another BASIC statement in order to be complete, then the accompanying statement must occur before the ELSE statement, if present, or before the END IF statement if there is no ELSE statement.

Line numbers in a statement outside the SELECT CASE structure must not refer to a statement within the structure.

NOTE

This construct requires several lines for entry. Each new line must begin with a line number and only one statement can occur on a line. For example, END SELECT must be entered on a separate line from the preceding statement and must have its own line number.

Semantics:

When a SELECT CASE structure is executed, <expression> is evaluated and compared with each of the case items in the order in which they occur. A match between <expression> and a case item occurs according to the following table:

Match Criteria

Case Item	<expression> Value
<constant>	Equal to <constant>
<constant> TO <constant>	Greater than or equal to first <constant> and less than or equal to second <constant>
> <constant>	Greater than <constant>
< <constant>	Less than <constant>

If a match is found, the block of <statement>s (case block) immediately following the CASE statement where the match occurred and immediately preceding the next CASE statement or the END SELECT statement is executed. If no case item is matched, the case block headed by CASE ELSE, if it occurs, is executed. If no case item is matched and there is no CASE ELSE clause, a fatal error occurs.

Example of a SELECT CASE structure:

```
SELECT A$
CASE "A" TO "Z", "a" TO "z"
      PRINT A$; " STARTS WITH A LETTER."
CASE "0" TO "9"
      PRINT A$; " STARTS WITH A DIGIT."
      PRINT "THE DIGIT IS "; A$(1:1)
CASE ELSE
      PRINT A$; " DOESN'T START WITH A LETTER OR A DIGIT."
END SELECT
```

SECTION 8
PROGRAM PARTITIONING

Burroughs B 1000 BASIC provides two facilities to partition programs. The first facility enables the user to define functions. These user-defined functions are used in numeric and string expressions in the same manner as the intrinsic numeric and string functions. The second facility, the CHAIN statement, enables separate programs to be executed sequentially without user intervention.

USER-DEFINED FUNCTIONS

There are two types of user-defined functions: single-statement functions and multiple-statement functions. Each type may be either a numeric or a string function.

Single-Statement Functions

A single-statement function is a user-defined function that requires only one statement for its definition, the DEF statement. The function definition specifies the method of evaluating the user-defined function based on the values of the parameters, if any. The syntax of the DEF statement for a single-statement function definition follows.

Syntax:
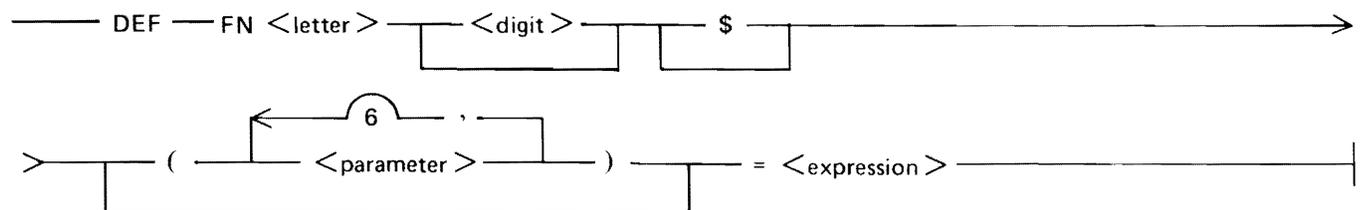
G18040

<letter> is any English alphabet letter from A to Z. <digit> is any decimal digit. Spaces may not occur between FN and <letter>, between <letter> and <digit>, nor between <digit> and $. The entities FN, <letter>, <digit>, and $, are referred to as the function name. <parameter> is a simple variable. <expression> is either a string or numeric expression. <expression> may not reference the function being defined; that is, recursion is not allowed in single-statement functions.

Semantics:

When a user-defined function is referenced, that is, when its name occurs in an expression, any arguments in the function reference are evaluated and their values are assigned to the parameters which appear in the function definition (parameters are passed by value to functions). After the arguments are passed, the expression associated with the function is evaluated and its value is assigned as the value of the function.

A <parameter> is recognized only within the function definition in which it appears, that is, parameters are local to the function; it is distinct from any variable with the same name outside the function definition. All other variables in a function definition are recognized in the entire program, that is, they are global to the program.

A function is executed only when its name is referenced. If a function definition statement is reached in some other fashion, the function is not executed, and execution proceeds to the next line.

Examples of single-statement function definitions:

DEF FNP = 3.14159   ! FNP is defined to be an
                           ! approximation of PI.

DEF FNA(X) = A * X + B

DEF FNC(A,B) = A * B + 1

DEF FNA$(S$,T$) = S$ & T$

The second example defines FNA as a numeric function with one parameter. The value of the function is the product of A and the parameter, X, added to B.

The third example defines FNC as a numeric function with two parameters. The value of the function is the product of the two parameters plus 1.

The fourth example defines FNA$ as a string function with two parameters. The value of the function is the concatenation of the two parameters.

Multiple-Statement Functions

A multiple-statement function is a user-defined function that requires more than one statement for its definition. The function definition specifies the method of evaluating the user-defined function based on the value(s) of the parameter(s), if any. The two statements required for this definition are the DEF statement for multiple-statement functions and the FNEND or END DEF statements. To avoid confusion, only the one-word name FNEND is used to refer to the END DEF statement. The syntax and semantics of these statements follows.

Syntax for the DEF statement:
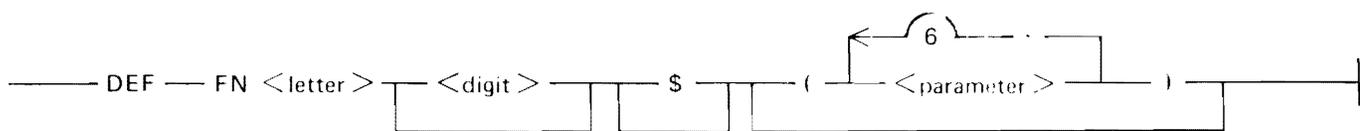
G18041

Syntax for the FNEND statement:

G18042

<letter> is any English alphabet letter from A to Z. <digit> is any decimal digit. Spaces are not allowed between FN and <letter>, between <letter> and <digit>, or between <digit> and $. The entities FN <letter>, <digit>, and $ are referred to as the function name. <parameter> is a simple variable.

Between the DEF statement and the FNEND statement, any BASIC statement may occur except another DEF statement or an END statement. These intermediate statements specify what actions the function performs. FOR NEXT statements must be entirely contained within the function. Recursive function calls are allowed.

Semantics:

A user-defined function is referenced by using its name in an expression. The arguments in the function reference, if any, are evaluated and their values assigned to the parameters which appear in the function definition, that is, parameters are passed by value to functions. After the arguments are passed, the expression associated with the function is evaluated and its value is assigned as the value of the function.

A <parameter> is local to the function definition in which it appears; it is distinct from any variable with the same name outside the function definition. All other variables in a function definition are global to the program unit in which the function occurs.

A function is executed only when its name is referenced. If a function DEF statement is reached in some other fashion, the function is not executed; execution proceeds to the line following the FNEND statement.

A control statement must not transfer control to a line within a multiple-statement function definition from outside the definition, or to a line outside a multiple-statement function definition from a line within it.

Examples of multiple-statement function definitions:

```
100 DEF  FNA(A$)
110      LET  FNA  =  1
120      IF A$  =  "YES"  THEN  140
130      LET  FNA  =  2
140 FNEND
```

```
100 DEF  FNB1$(T)
110      LET  FNB1$  =  "YES"
120      IF T  =  1 THEN  140
130      LET  FNB1$  =  "NO"
140 FNEND
```

Both of these examples show the use of the LET statement with a multiple-statement function. The LET statement is used to assign a value to the function. If no LET statement occurs in a multiple-statement function, the value of the function is either zero or the null string depending on the type of the function. The assignment statement for multiple-statement functions is fully described in the following subsection.

These examples are very similar to one another: the first assigns a numeric value to the function according to the value of a string variable, A$. The second does the converse of the first in that a string value is assigned to the function according to the value of a numeric variable.

Assignment Statement For Multiple-Statement Functions

The assignment statement for multiple-statement functions resembles the assignment statement for numeric and string variables as described in Sections 3 and 4, respectively. However, there is one difference: the assignment statement, as used with multiple-statement functions, is used to assign a value to a function name rather than to a variable.

Syntax:

```
──┬──── LET ──┬──── <function-name> = <expression> ──────────────────────┤
  └───────────┘
```

G18043

<function-name> must be a multiple-statement function. <expression> is any numeric or string expression. The assignment statement for multiple-statement functions must occur within a multiple-statement function.

For examples of the LET statement, refer to the previous subsection entitled Multiple-Statement Functions.

CHAIN STATEMENT

The CHAIN statement allows separate BASIC programs to be run serially without programmer intervention.

Syntax:

```
──────── CHAIN  <program-designator> ───────────────────────────────────┤
```
G18044

<program-designator> is a string expression which specifies the name of the next program to be run. <program-designator> must take the following form:

```
──────── <program-name> ──┬──── ON <pack-name> ──┬──────── ──────────┤
                          └──────────────────────┘
```

G18045

<program-name> must conform to the rules for MCP file names as described under Syntax Definitions in Section 11. <pack-name> is described in the same subsection.

Semantics:

When <program-designator> is evaluated, <program-name> is the name of a BASIC source program residing on disk. The ON option specifies the pack upon which the file resides. If ON is not specified, the default pack is assumed. The default pack is the pack associated with the currently logged-on usercode, or it is the system disk if no usercode is used.

Upon execution of a CHAIN statement, all files that are open (have an assigned channel number) in the currently executing program, are closed (disassociated from the channel number) and must be explicitly opened in the following program if they are to be used in that program. Variables in the program designated by <program-name> are independent of variables of the same name in the program

containing the CHAIN statement; that is, all variables in <program-name> are initialized to zero or to the null string, depending on their type.

Execution of a CHAIN statement causes termination of the program containing the CHAIN statement. The CHAIN statement is not executed if the program containing the CHAIN statement has not been saved. Refer to the SAVE command in Section 11.

Examples of CHAIN statements:

```
CHAIN "PROG2"

CHAIN A$                          !Chain to the program whose name
                                  !is contained in A$.

CHAIN "BASIC/PROG1 ON             !BASIC/PROG1 is on the
PACK1"                            ! pack named PACK1.
```

SECTION 9
INPUT/OUTPUT

Input and output facilities provide for the interaction of a BASIC program with collections of data. Data may be obtained by a program from statements within the program, from a terminal, or from an external file. Output data may be directed to a terminal, a line printer, or an external file. This section describes the statements available in Burroughs B 1000 BASIC for input from and output to these sources and destinations. The section is divided into three major subsections: Program-internal Input, Terminal I/O, and File I/O Statements.

PROGRAM-INTERNAL INPUT

Program-internal data is data that is obtained by a program from statements within the program. There are three statements associated with program-internal data: the DATA statement, the READ statement, and the RESTORE statement.

DATA Statement

The DATA statement supplies data to a READ statement.

Syntax:

G18046

<datum> is either a numeric constant, a string constant, or a string constant without the enclosing quotation marks. More than one DATA statement may appear in a program.

Semantics:

Data from all DATA statements in a program is logically grouped into one data block and is read from that block in the sequence in which the DATA statements appear in the program.

If the execution of a program reaches a line containing a DATA statement, execution proceeds to the next line with no other effect.

The DATA statement is always used in conjunction with the statement.

Examples of DATA statements:

```
DATA 5,12,50,1,8,734
DATA 3.14159, PI, 5E-30, ","
DATA COMMAS CANNOT OCCUR IN UNQUOTED STRINGS.
```

READ Statement

The READ statement reads the data from the DATA statement(s).

Syntax:

```
                 ┌──────── , ────────┐
────── READ ─────┤──── <variable> ───┘──────────────────────────────────┤
G18047
```

<variable> can be any numeric or string variable.

Semantics:

Execution of a READ statement causes variables in the READ statement to be assigned values from the DATA statements in the program unit. If there are more <variable>s in the READ statement than there are data items in the DATA statement, a fatal error occurs.

The type of a <datum> in the DATA statements must correspond to the type of the variable to which it is to be assigned. Numeric variables require as data unquoted strings which are numeric constants, and string variables require quoted strings or unquoted strings as data. An unquoted string that is a valid numeric constant may be assigned by a READ statement to a string variable or to a numeric variable. If an attempt is made to assign a quoted string constant to a numeric variable or to assign an unquoted string constant to a numeric variable, a fatal error results.

Variables in the list of <variable>s for the READ statement are assigned values from left to right. Thus, any variables that appear as subscripts in the list of <variable>s are evaluated after values are assigned to the variables preceding (to the left of) the subscripted variable. Thus, a variable, I for example, may appear in a READ statement as a simple numeric variable, and then to the right of its first appearance, as the subscript for a subscripted variable (for example, READ I,A$(I)).

If the assignment of a numeric <datum> causes an underflow, the value of the <datum> is replaced by the value 0. If an overflow occurs, the <datum> is replaced by the largest machine value. In both of these cases the condition is reported to the user and execution continues. If assignment of a string <datum> to a string variable results in a string overflow, a fatal error results.

Examples of READ and DATA statements:

```
READ A,B,C
READ Z$,Y$,X$
READ D(A), E1(B), F$(C), I
DATA 5,0,9,25,"THIS IS A QUOTED STRING."
DATA THIS IS AN UNQUOTED STRING.,1E+50,7,JOE SMITH,1
```

Execution of these statements has the same effect as the following assignment statements:

```
LET A = 5
LET B = 0
LET C = 9
LET Z$ = "25"
LET Y$ = "THIS IS A QUOTED STRING."
LET X$ = "THIS IS AN UNQUOTED STRING."
LET D(A) = 1E+50
LET E1(B) = 7
LET F$(C) = "JOE SMITH"
LET I = 1
```

RESTORE Statement

The RESTORE statement allows the data in DATA statements to be reread.

Syntax:

```
———— RESTORE ————————————————————————————————————————————————————————|
G18048
```

Semantics:

Execution of a RESTORE statement causes subsequent READ statements to take data starting from the beginning of the block of data formed from all of the DATA statements in the program.

Example of READ, DATA, and RESTORE statements:

```
READ I,J,K,L
DATA 1,2,10,100
RESTORE
READ M,N,O
```

Execution of these statements is equivalent to the following assignment statements:

```
LET I = 1
LET J = 2
LET K = 10
LET L = 100
LET M = 1
LET N = 2
LET O = 10
```

TERMINAL I/O

Terminal input and output statements provide for user interaction with a program by allowing variables to be assigned values supplied from a terminal. The various statements used in terminal I/O are described in this section.

Terminal Input

There are two statements associated with terminal input: the INPUT statement and the LINPUT statement.

INPUT Statement

The INPUT statement allows the user to supply data from a terminal to variables within a program. The syntax for the INPUT statement and for the user's reply follow.

Syntax for the INPUT statement:

```
                                        ┌───── , ─────┐
——— INPUT ─┬── PROMPT <string-expression> : ─┬┴─ <variable> ─┴┬─── , ───┬───————|
           └──────────────────────────────────┘               └─────────┘
G18049
```

Syntax for the INPUT reply:
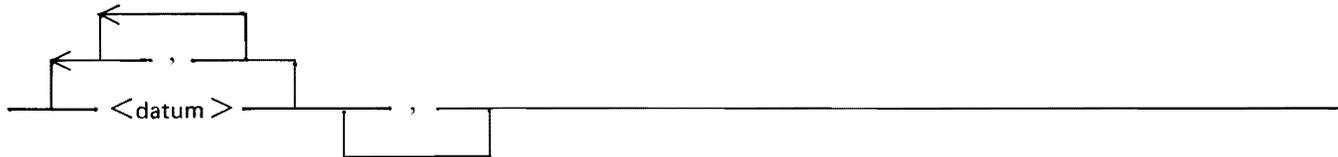
G18050

<string-expression> is any string expression as described in Section 5. <variable> is any numeric or string variable as described in Sections 4 and 5, respectively. The <datum> is either a numeric constant, a string constant, or an unquoted string. An unquoted string consists of characters from Table E-1 in the following ordinal positions: 32, 43, 45, 46, 48 through 57, 65 through 90, and 97 through 122. An unquoted string must be delimited by a comma (,) character if data follows it.

Semantics:

Execution of an INPUT statement causes program execution to be suspended until a valid reply is supplied. The user of a program is informed of the need to supply data by the output of a prompt. If the PROMPT option is not specified in the INPUT statement, the prompt is a question mark (?) followed by a space. If PROMPT is specified, the prompt is the <string-expression>. A null PROMPT ("") is allowed. A null PROMPT allows the input of data from a remote terminal in forms mode.

Each <datum> from the user's reply must correspond to the type of the variable to which it is to be assigned. Numeric constants must correspond to numeric variables, and string constants or string constants not enclosed in quotation marks must correspond to string variables. Each <datum> must also be within the allowable range of values for that <datum>, and there must be an adequate number of data items for the list of <variable>s. No assignment of data is made until the preceding criteria are met. If these criteria are not met, IBASIC requests that the user resupply the data.

After this validation process is completed and valid data have been supplied, the variables in the INPUT statement are assigned values from the INPUT reply in the order in which they occur.

If an overflow occurs on a <datum>, whether numeric or string, IBASIC requests that the input be resupplied. If an underflow occurs on a numeric <datum>, its value is replaced by the value zero; execution then continues.

If neither the INPUT statement nor the corresponding reply contains a final comma (,) character, the number of data items in the reply must equal the number of variables in the INPUT statement.

If an INPUT statement contains a final comma (,) character, the number of data in the INPUT reply may exceed the number of variables requiring values. The remaining data in an INPUT reply are retained to serve as the next requested INPUT reply or LINPUT reply. Any such excess data are discarded upon execution of a PRINT statement.

A comma (,) character at the end of an INPUT reply signifies that more data will be supplied. After the values contained in the INPUT reply are assigned to variables in the INPUT statement, a prompt is reissued. Execution of the program remains suspended until each <variable> in the <variable> list has been supplied with a value.

Subscripts in the list of <variables> are evaluated after values are assigned to the variables preceding (to the left of) them in the list.

9-4

Examples of INPUT statements:

    INPUT X
    INPUT X, A$, Y(2)
    INPUT PROMPT "What is your name? ": N$(30:50)
    INPUT X, Y,
    INPUT PROMPT "": C(3)

Examples of corresponding INPUT replies:

    2
    25 "ABOVE" 0.2
    JOHN DOE
    1,2,3,4,5,6
    7000

LINE INPUT (LINPUT) Statement

The LINE INPUT statement, referred to as LINPUT throughout the rest of this manual, enables an entire line of input, including embedded spaces, commas, and quotation marks, to be assigned as the value of a string variable. The syntax and semantics of the LINPUT statement and for the reply to the LINPUT statement follow.

Syntax for the LINPUT statement:
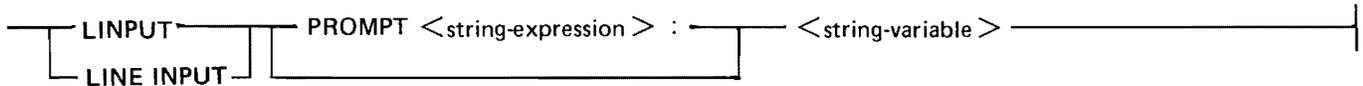
G18051

Syntax for the LINPUT reply:

G18052

<string-expression> is any string expression as described under String Expressions in Section 5. <string-variable> is any string variable as described under String Variables in Section 5. <character> is any character.

Semantics:

Execution of a LINPUT statement causes program execution to be suspended until a valid reply is supplied. The user of a program is informed of the need to supply data by the output of a prompt. If the PROMPT option is not specified in the LINPUT statement, the prompt is a question mark (?) character followed by a space. If PROMPT is specified, the prompt is a <string-expression>. A null PROMPT ("") is allowed. A null PROMPT allows the input of data from a remote terminal in forms mode. After the LINPUT reply is supplied, the string of <character>s is assigned to the <string-variable>.

Examples of LINPUT statements:

```
LINPUT A$
LINPUT Z$(50:75)
LINPUT PROMPT "": A$, B$          !Prompt is the null string
```

Examples of LINPUT replies:

NOW IS THE TIME FOR ALL GOOD MEN TO COME TO THE AID OF THEIR PARTY.
"#$%&'(=-__.,?/<>+;:*@
Any valid character, including commas, may occur in a LINPUT reply.
THIS   IS   A   LINPUT   REPLY.

Terminal Output

There are four statements associated with terminal output: the PRINT statement, the PRINT USING
and IMAGE statements, and the MARGIN statement.

PRINT Statement

The PRINT statement is used to generate output on a terminal.

Syntax:
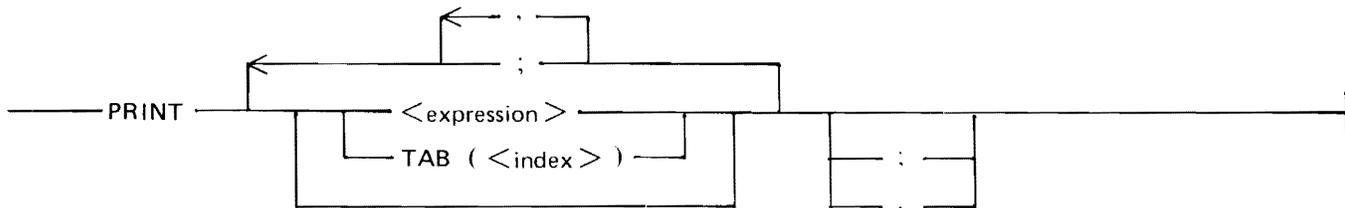
G18053

<expression> is any numeric or string expression. <index> is a numeric expression which, when
evaluated, is the columnar position on the terminal where the next <expression> is to be displayed.
Columnar position is the number of print positions from the leftmost print position.

Semantics:

The execution of a PRINT statement generates a string of characters for transmission to a terminal.
This string of characters is determined by the successive evaluation of each item in the PRINT state-
ment as well as the type of separator used to delimit the items. Refer to Print Separators and TABs
in this section.

Examples of PRINT statements:

```
PRINT X
PRINT X, Y
PRINT X, Y, Z,
PRINT ,,,X
PRINT
PRINT "X EQUALS", 10
PRINT X; (Y*Z)/2
PRINT TAB(10); A$; "IS DONE."
```

The rules used for displaying the items in the PRINT statement are described in the four subsections that follow.

Printing Numeric Values

Numeric expressions in a PRINT statement produce a string of characters consisting of a leading space if the number is positive, or a leading minus sign if the number is negative. This leading space or minus sign is followed by the decimal representation of the absolute value of the number. This sequence of characters is terminated by a trailing space. The possible decimal representations of a number are the same as those described for numeric constants in Section 4 and are used as follows:

1. A numeric value that is an integer in the range -16777215 to $+16777215$ is written as an integer, that is, a series of decimal digits without a decimal point or an exponent.
2. A numeric value that cannot be represented precisely as an integer is written as a real number, that is, a series of decimal digits with a decimal point. A real value that can be represented with six or less digits without losing accuracy is written without an exponent.
3. If accuracy would be lost by using only six digits, the value is displayed as a normalized decimal number with the necessary significant digits and with an exponent. For example, $10**(-6)$ is written as .000001 and $10**(-7)$ is written as 1.E-7. Exponents have a maximum of two decimal digits.

The maximum widths for each format follow.

Format	Maximum width
Integer	! 10 places: sign, 8 digits, ! trailing space
Real, no exponent	! 9 places: sign, 6 digits, ! decimal point, ! trailing space
Real, exponent	! 13 places: sign, 6 digits, ! decimal point, ! E, sign of exponent, ! 2-digit exponent, trailing ! space

Examples of numeric output:

1
5000
3.1415
6.283E + 25
0
1.E-7

Printing String Values

String expressions are evaluated to generate the corresponding string of characters.

Print Separators and Tabs

A print separator is either a comma (,) or a semicolon (;) and is used to separate items in a PRINT or OUTPUT statement. These separators are shown in the syntax for the PRINT and OUTPUT statements in this section. TAB is also shown in the syntax for the PRINT and OUTPUT statements. The function of print separators and TABs is to specify what type of spacing is to be used between print items as they are displayed. A print item is an expression or TAB call occurring in a PRINT or OUTPUT statement.

The evaluation of the semicolon separator generates the null string within the output. A null string is a string of zero length.

Example:

```
LET A, B  =  4
PRINT  "A + B = ";  A + B;  "A*B = ";  16
```

Execution of this example causes the following to be displayed.

```
A + B =  8 A*B =  16
```

The output from the evaluation of a comma separator or a TAB depends upon the string of characters already generated by the current or previous PRINT statements.

The use of the comma separator causes the columnar position to be advanced to the end of a predefined print zone. A print zone has a width of 15 print positions. In an 80-character print line there are five full-width print zones and one 5-character partial print zone. The length of the print line and, thus, the number of print zones may be changed by the MARGIN statement. For more information, refer to the description of the MARGIN statement in this section.

The evaluation of the comma print separator depends upon the current columnar position. One of three actions may be taken depending on this position. First, if the columnar position is neither in the last print zone on a line nor beyond the margin, one or more spaces are generated to set the columnar position to the beginning of the next print zone on the line.

Example:

```
LET A,B=4
PRINT "A+B=", A+B, "A*B=", 16
PRINT "----*----1----*----2----*----3----*----4----*----5"
```

Execution of the above example causes the following to be displayed.

```
A+B=            8           A*B=            16
----*----1----*----2----*----3----*----4----*----5
```

Second, if the current columnar position is in the last print zone on a line, an end-of-line character is generated by IBASIC and subsequent displaying continues on the next line. An end-of-line is a NUL, CR, LF, or ETX character, or the end of the record.

Example:

```
MARGIN 50     ! The MARGIN statement is described in Section 9.
PRINT 1,2,3,4,5,6,7
PRINT "----*----1----*----2----*----3----*----4----*----5"
```

Execution of this example gives the following output:

```
1             2             3             4
5             6             7
----*----1----*----2----*----3----*----4----*----5
```

Third, if the current columnar position is beyond the margin, as it would be if evaluation of the last print item exactly filled the line, an end-of-line character is generated and subsequent displaying begins in the first print zone on the new line.

Example:

```
MARGIN 50
PRINT 1,2,3,"FOUR",5,6,7
PRINT "----*----1----*----2----*----3----*----4----*----5"
```

Execution of this example gives the following output:

```
1             2             3             FOUR
5             6             7
----*----1----*----2----*----3----*----4----*----5
```

The TAB call sets the columnar position of the current line to the specified value. The current line is the string of characters generated by PRINT and OUTPUT statements since the last end-of-line character was generated. The syntax for the TAB call is shown in the PRINT statement syntax and also in the OUTPUT statement syntax.

When TAB is used in a PRINT statement, <index> is evaluated and rounded to an integer, n. If n is less than 1, an error message is displayed, n is replaced by 1, and execution continues. If n is greater than margin m, n is reduced by an integral multiple of m so that it is within the range $1 <= n <= m$; that is, n is set equal to $n - m * INT((n-1)/m)$.

If the current columnar position of the current line is less than or equal to n, spaces are generated, if necessary, to set the columnar position to n. If the current columnar position of the current line is greater than n, an end-of-line character is generated followed by $n-1$ spaces to set the columnar position of the new current line to n.

If the value of a TAB expression is so large that significance is lost, a nonfatal error occurs. A nonfatal error is an error which causes an error message to be printed and allows execution to continue.

Example:

```
MARGIN 50
PRINT A; TAB(20); -1; TAB(40); B; TAB(65); C
PRINT "----*----1----*----2----*----3----*----4----*----5"
```

Execution of this example gives the following output:

```
0                       -1                      0
                  0
----*----1----*----2----*----3----*----4----*----5
```

End-of-Line Conditions

Under certain conditions an end-of-line character is generated by IBASIC before subsequent action to a print line. These conditions are described next.

Whenever the columnar position is greater than the integer 1 and the evaluation of the next print item would cause that position to exceed the margin by more than one position, an end-of-line character is generated prior to the characters generated by that print item.

During the evaluation of a print item whose length is greater than the margin length, if the generation of a character would cause the columnar position to exceed the margin by more than one position, an end-of-line character is generated before that character is displayed, resetting the columnar position to 1 on the following line.

An end-of-line character is generated when evaluation of a list of print items is completed, if that list does not end with a print separator. Otherwise, the rules for the print separator prevail.

Formatted Output

The PRINT USING statement is used to control the format of output by specifying an image to which that output must conform. The following two subsections describe the PRINT USING statement and the images used with it.

PRINT USING Statement

The PRINT USING statement uses a user-created image to format the print items on the print line.

Syntax:
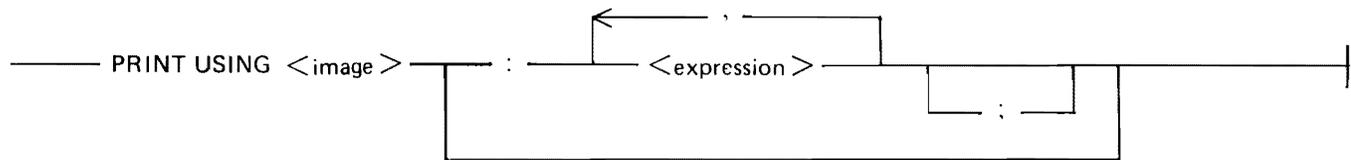
G18054

is either a string expression or a line number which references a separate IMAGE statement. <expression> is either a numeric or a string expression. The items following the colon (:) are referred to as the output list.

Semantics:

The execution of a PRINT USING statement generates a string of characters for transmission to a terminal. This string, which is generated from the list of <expression>s, is formatted according to the <image>. Possible values for the <image> are described next under Images.

Examples of PRINT USING statements:

> PRINT USING 100: A, B   Image is located at line 100.
>
> PRINT USING A$: D, E, F;   Image is located in string A$.

Images

Images are used in conjunction with PRINT USING statements. An <image> is either a line number which refers to an IMAGE statement containing a format string, or a string expression whose value is the format string. The syntax for a string expression used as an image is the same as for string expressions as described in Section 5. The syntax for a line number used as an image is the same as for line numbers as described under Statement Lines in Section 3. The syntax for the IMAGE statement follows. The value for a string expression used as an image is also contained in the following syntax diagram, except that the word IMAGE and the colon (:) are excluded from the string expression.

Syntax:
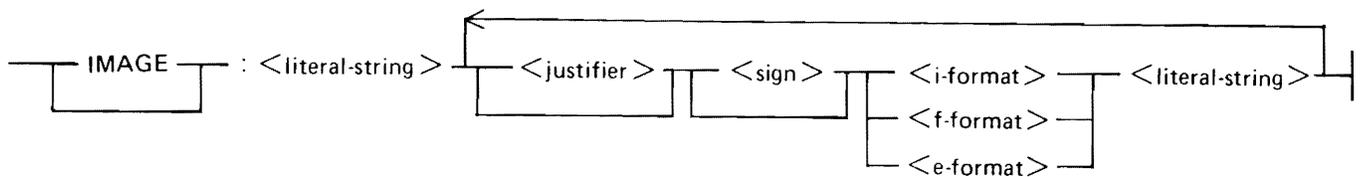
```
──┬─ IMAGE ─┬─ : ─ <literal-string> ─┬─┬─ <justifier> ─┬─┬─ <sign> ─┬─┬─ <i-format> ─┬─ <literal-string> ─┘
  └─────────┘                        │ └───────────────┘ └──────────┘ ├─ <f-format> ─┤
                                     │                                └─ <e-format> ─┘
G18055
```

<literal-string> is made up of any sequence of characters that may be used in a string constant except the following:

Character	Name
>	Greater than
<	Less than
#	Number sign
+	Plus sign
−	Minus sign
.	Period
^	Circumflex accent
"	Quotation mark

Even though these characters cannot be used in a <literal-string>, the same effect may be obtained by putting them in quoted strings in the output list.

<justifier> is either a greater than (>) or a less than (<) character. <sign> is a plus (+) or a minus (−) character.

The format items, <i-format>, <f-format>, and <e-format>, are described in detail under Formatted Numeric Output in this section. All of the items following the colon in an image, which may include several format items and <literal-string>s, are referred to as the format string. Any spaces following the colon are part of the format string.

Semantics:

When the execution of a program encounters a line containing an IMAGE statement, execution proceeds to the next line with no other effect.

When a PRINT USING statement is executed, the associated format string is scanned. Any <literal-string>s are displayed exactly as they occur in the format string. Any format items generate an output field whose length equals the number of characters in the format item (including the <justifier>, <sign>, number sign (#), period (.), and <circumflex-accent>s). The contents of the output field depend upon the corresponding <expression> in the PRINT USING statement. <expression>s are displayed in the sequence in which they occur, according to the format item currently being scanned.

The <expression>s are displayed in the manner described in the following three main subsections: Formatted Numeric Output, Formatted String Output, and End-of-Line Conditions.

If a PRINT USING statement contains an output list, but there is no format item in the associated format string, a fatal error occurs.

If the output from an <expression> in a PRINT USING statement is longer than its corresponding format item, the current line is terminated by an end-of-line character, the evaluated <expression> is displayed unformatted, and displaying continues according to the format. Refer to the second of the following five examples.

Examples of PRINT USING and IMAGE statements:

Assume X has the value 342 and Y has the value 42.021.

```
30 PRINT USING 40: X, Y
40 IMAGE:RATE OF LOSS #### EQUALS ####.## POUNDS
```

The output is the following:

```
RATE OF LOSS  342 EQUALS    42.02 POUNDS
```

Example of format item overflow:

```
5 PRINT USING "OVERFLOW FORMAT ITEM #STARTS NEW LINE":34564
```

The output is the following:

```
OVERFLOW FORMAT ITEM
34564 STARTS NEW LINE
```

Assume A, B, and C have the value 1.

```
10 LET A$ = "<##### ####.#### ####.#### ^^^^"
20 PRINT USING A$: A, B, C
```

The output is the following:

```
1        1.0000 1000.0000E-03
```

Use of the left justifier.

```
60 PRINT USING 70: "ONE", "TWO"
70 :Z<####<####Z
```

These two statements give the following output:

```
ZONE  TWO  Z
```

Use of both justifiers.

```
110 IMAGE :> -##< -##
120 PRINT USING 110: -2, -2
```

These two statements give the following output
(the quotation marks are not displayed):

```
"  -2-2  "
```

Formatted Numeric Output

There are three steps in displaying formatted numbers: generating the value, generating the sign, and justifying the value.

The value is generated first. Numeric values are generated by being rounded and represented according to the format used. The three possible formats are the i-format, the f-format, and the e-format.

i-format

The i-format consists of a series of contiguous number signs (#). For the i-format, the corresponding value is rounded to the nearest integer and is represented using implicit-point, unscaled notation, with no superfluous leading zeros. Implicit-point notation means that the decimal point is not present. Unscaled notation means that there is no exponent part.

Syntax:
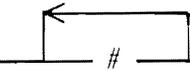
G18056

Example of an i-format:

```
###
```

f-format

The f-format consists of a series of contiguous number signs (#) and an explicit decimal point (.). The decimal point can occur at any point within, before, or after the string of number signs. For the f-format, the corresponding value is represented by explicit-point unscaled notation. Moreover, the representation is rounded or extended according to the number of number signs (#) following the decimal point in the format item. Zeros are not generated to the left of the decimal point unless the number is less than 1 and there is at least one number sign (#) to the left of the decimal point in the format item. In that case, a zero is generated immediately before the decimal point.

Syntax:
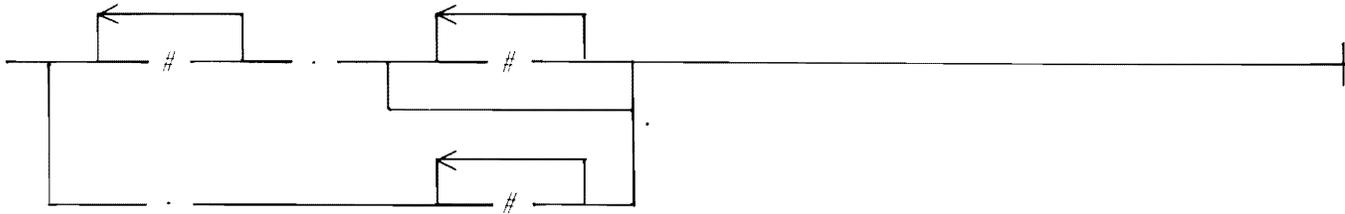
G18057

Examples of f-formats:

```
###.
##.##
.#####
```

e-format

The e-format consists of the i-format or the f-format followed by three or more circumflex accents (∧). For the e-format, the corresponding value is represented by explicit- or implicit-point scaled notation, with as many digits to the left of the decimal point or within the integer as there are number signs (#) to the left of the decimal point in the format item. The representation is rounded or extended according to the number of number signs following the decimal point in the format item. A value of zero generates a single zero in the integer to the left of the decimal point. The number of <circumflex-accent>s in an e-format determines the number of characters in the exponent. The first of these characters is the letter E, the next is a mandatory sign, and the remaining characters represent the value of the exponent, with leading zeros added to ensure that the exponent has the proper length. If the exponent is zero, the mandatory sign is positive; the exponent of zero is zero.

Syntax:
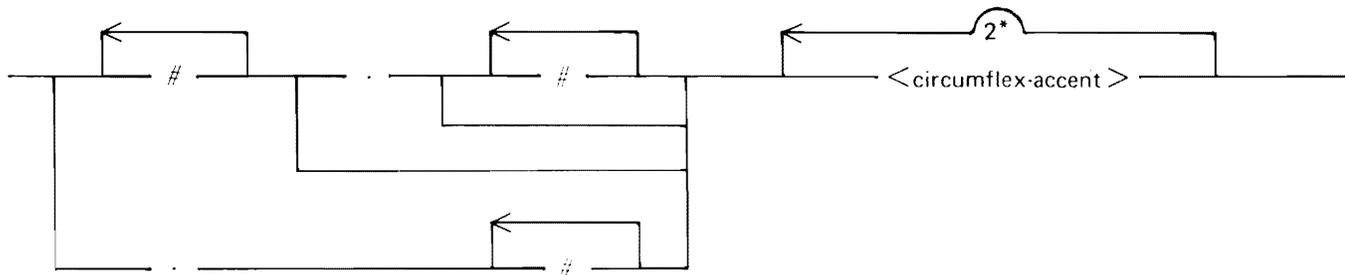
G18058

Examples of e-formats:

```
###∧∧∧
.####∧∧∧∧∧
#.∧∧∧
#.######∧∧∧∧
```

The second step in displaying a formatted number is generating the sign. A leading sign or space is always generated with each number to be displayed according to the following rules.

1. If the rounded value of a number is negative, a minus sign (–) is generated regardless of the sign in the format item.
2. If the rounded value is nonnegative and the format item contains a plus sign (+), a plus sign is generated.
3. If the rounded value is nonnegative and the format item contains a minus sign, a space is generated.
4. If the rounded value is nonnegative and the format item contains no sign, no leading space or sign is generated.

The third step taken by IBASIC before displaying a numeric value is to justify the value (extend it with spaces), if necessary, so that its length equals that of the format item. Spaces are added on the left, unless a format item begins with a less than sign (<), in which case the spaces are added on the right.

Formatted String Output

A string value may be written using any type of format item. The string is extended by spaces so that its length equals that of the format item. These spaces are added on the left for right justification if the format item begins with a greater than sign (>). They are added on the right for left justification if the format item begins with a less than sign. Otherwise, they are added equally on either side for centering. If the number of spaces required in the last case is odd, the extra space is added on the right.

If the string value is longer than its corresponding format item, the current print line is terminated by an end-of-line character, the string value is displayed unformatted, and displaying continues according to the format.

Examples of formatted string output:

```
PRINT USING "<########################## LITERAL STRING":"THIS IS A"    :
PRINT USING "##############.############ LITERAL STRING":"THIS IS A"    :
PRINT USING ">########################## LITERAL STRING":"THIS IS A"    :
```

Execution of these examples gives the following output:

```
THIS IS A                      LITERAL STRING
            THIS IS A           LITERAL STRING
              THIS IS A LITERAL STRING
```

End-of-Line Conditions

The characters generated by each <literal-string> and each value under the control of a format item are transmitted in the same manner as described under Terminal Output in this section. In particular, if the generation of characters for any <literal-string> or value would cause the columnar position of a nonempty line to exceed the margin by more than one, an end-of-line character is generated before the characters of the <literal-string> or value. Furthermore, an end-of-line character is generated each time the columnar position of the current line exceeds a nonzero margin.

Example:

```
MARGIN 50
PRINT 1,2,3,"OVERLAP"
PRINT "----*----1----*----2----*----3----*----4----*----5"
PRINT 1,2,3,4,5,6,7,8,9,10
PRINT "----*----1----*----2----*----3----*----4----*----5"
```

Execution of this example gives the following output:

```
1               2               3
OVERLAP
----*----1----*----2----*----3----*----4----*----5
1               2               3               4
5               6               7               8
9               10
----*----1----*----2----*----3----*----4----*----5
```

If the number of values to be written exceeds the number of format items in the format string, an end-of-line character is generated and the format string is reused for the remaining expressions. If format items remain in the format string after all values have been written, any succeeding <literal-string> is written, and generation of characters is terminated beginning at the first unused format item.

Example:

```
A$ = "### ### END OF FORMAT ITEMS"
PRINT USING A$:1,-1,2,-2,3,-3,4,-4,5
```

Execution of this example gives the following output:

```
1  -1 END OF FORMAT ITEMS
2  -2 END OF FORMAT ITEMS
3  -3 END OF FORMAT ITEMS
4  -4 END OF FORMAT ITEMS
5
```

Finally, an end-of-line character is generated after all other character generation is completed, unless the output list ends with a semicolon, in which case no end-of-line character is generated.

MARGIN Statement

The MARGIN statement provides programmatic control over the length of lines produced by the PRINT or PRINT USING statement.

Syntax:

```
———— MARGIN <margin-value> ——————————————————————————————————|
```
G18059

<margin-value> is a numeric expression.

Semantics:

The MARGIN statement resets the maximum number of bytes that the PRINT and PRINT USING statements can generate in a print line. When <margin-value> is evaluated, it is rounded to the nearest integer and then assigned as the new margin. The new margin takes effect immediately; thus, if a partial line is awaiting completion, the new margin is used when a subsequent PRINT or PRINT USING statement is executed.

A margin setting of 0 restores the margin to the default value for the terminal to which the user is attached.

Examples of MARGIN statements:

```
MARGIN 20
PRINT "MARGIN LENGTH - 20"
PRINT "THIS SENTENCE GOES BEYOND THE MARGIN."
PRINT "THIS STAYS WITHIN."
MARGIN 50
PRINT "NOW THE MARGIN IS 50.
PRINT "----*----1----*----2----*----3----*----4----*----5"
```

Execution of these examples gives the following output:

```
MARGIN LENGTH - 20
THIS SENTENCE GOES B
EYOND THE MARGIN.
THIS STAYS WITHIN.
NOW THE MARGIN IS 50.
----*----1----*----2----*----3----*----4----*----5
```

ARRAY I/O

Array I/O statements enable entire arrays to be read or written.

Array Input

Two statements that enable an entire array to be input are the MAT READ statement and the MAT INPUT statement. The MAT READ statement enables an array to be initialized from internal data. The MAT INPUT statement enables an array to be input from a terminal. These two statements are described in the following paragraphs.

MAT READ Statement

Execution of a MAT READ statement causes arrays to be assigned values from the data sequence created by DATA statements.

Syntax:
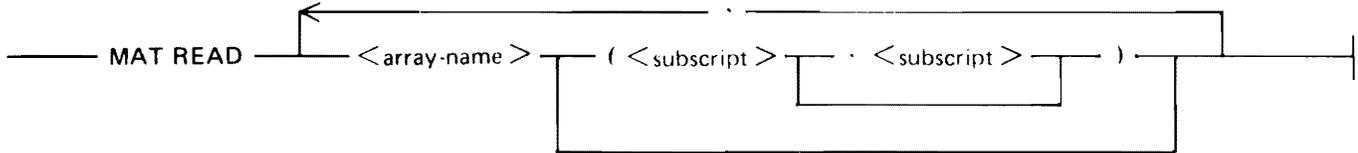
G18060

<array-name> follows the same naming conventions as simple variables. <subscript> is a numeric expression. This numeric expression must evaluate to a number which is greater than or equal to the lower-dimension bound for arrays in the program.

Semantics:

If one or both of the <subscript>s are present, the array is redimensioned before values are assigned to it. Arrays are redimensioned in the manner described for the MAT Assignment Statement in Section 6. <subscript>s are evaluated after values are assigned to the arrays preceding (to the left of) them in the array list.

Elements in an array are assigned values from the DATA statements in a row-by-row fashion. Refer to DATA Statement in this section for more information on DATA statements. The example in this subsection shows row-by-row assignment.

The type of each datum in the data sequence must correspond to the type of the array element to which it is to be assigned. Numeric variables require numeric constants as data and string variables require quoted strings or unquoted strings as data. An unquoted string which is a valid numeric representation may be assigned to a string variable or to a numeric variable by a MAT READ statement. If data types do not match, a fatal error occurs.

If the array list requires more data than are present in the remainder of the data sequence, a fatal error occurs.

Example of the use of a MAT READ statement:

```
OPTION BASE 1
DIM A(10,10)
DATA 1,2,3,4,5,6, 7,8,9,10,11,12
MAT READ A(3,4)
MAT PRINT A;
```

Execution of this example gives the following output:

```
1    2    3    4
5    6    7    8
9   10   11   12
```

MAT INPUT Statement

The MAT INPUT statement permits entire arrays to be input from the terminal. The syntax and semantics of the MAT INPUT statement follow.

Syntax:
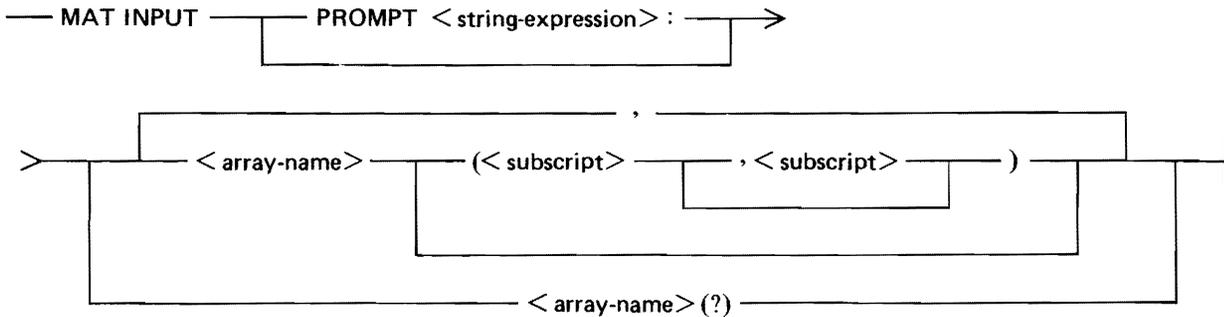
<string-expression> is any string expression as described under String Expressions in Section 5.

<array-name> follows the same naming conventions as used for simple variables. <subscript> is a numeric expression. This numeric expression must evaluate to a number which is greater than or equal to the lower-dimension bound for arrays in the program. <array-name> (?) specifies a variable-length vector.

Semantics:

Execution of a MAT INPUT statement suspends the program until a valid input reply, as specified in subsequent paragraphs, has been supplied. After a valid input reply is received, the MAT INPUT statement causes all arrays in the statement except variable-length vectors to be assigned values from the input reply, in the order in which they appear in the statement. Values are assigned to all elements in each array row by row.

The program user receives a prompt when data is required for a MAT INPUT statement. If the PROMPT option is not specified in the MAT INPUT statement, the prompt is a question mark (?) character followed by a space. If PROMPT is specified, the prompt is the <string-expression>.

If one or both of the <subscript>s are present, the array is redimensioned before values are assigned to it, but after values are assigned to the arrays preceding it in the array list. Arrays are redimensioned in the manner described for the MAT Assignment Statement in Section 6.

The type of each data element in the input reply must correspond to the type of the array element to which it is to be assigned.

The entire response to a request for array input does not have to be supplied with a single input reply. If an input reply contains a final comma (,) character, then a further input reply is requested to obtain more data.

If the evaluation of a numeric data element causes an underflow, the value of the element is replaced by zero.

If <array-name> (?), a variable-length vector, appears in the MAT INPUT statement, then as many data as are present in the input reply (or sequence of input replies up to and including the first which does not end with a comma (,) character) are supplied as input for that vector. The vector is then

redimensioned dynamically by setting the upper bound of the vector equal to the subscript of the element receiving the last data element.

With variable-length vectors, the value of the function NUM is the number of data supplied to the last array that receives data by means of a MAT INPUT statement.

Example of the use of MAT INPUT statements:

```
OPTION BASE 1
DIM A$(4),B(3,4)
MAT INPUT PROMPT "ENTER 4 GRADES": A$
MAT INPUT B(3,3),C(?)
MAT PRINT A$;B;C;
```

Assume the following input is entered from a remote terminal:

```
C,B,A,B
1,2,3,4,5,6,7,8,9
-1,-2,-3,-4
```

Execution of this example results in the following output:

```
CBAB

1    2    3
4    5    6
7    8    9

-1   -2   -3   -4
```

Array Output

One statement is provided to enable entire arrays to be written to the terminal. This statement is the MAT PRINT statement.

MAT PRINT Statement

The MAT PRINT statement displays an entire array on the terminal in row order.

Syntax:
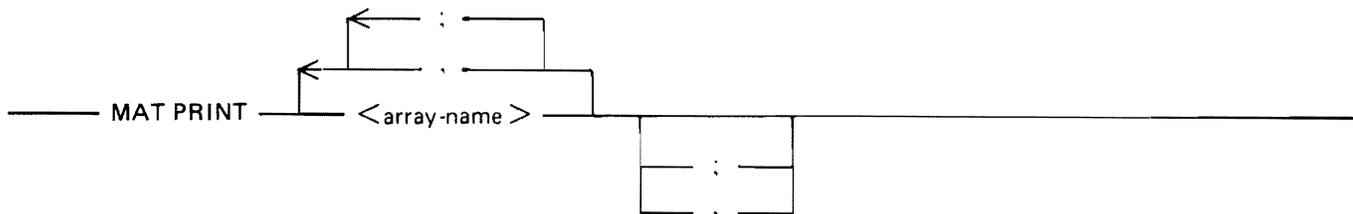
G18061

<array-name> follows the same naming conventions as simple variables.

Semantics:

Execution of a MAT PRINT statement causes the values of all elements in all arrays in the list of
<array-name>s to be displayed on the terminal. The characters generated for transmission to the ter-
minal by the displaying of one array in the list of <array-name>s are those that would be generated
if the individual elements in an array had been listed, row by row, in the list of <expression>s of
a PRINT statement. Each array element displayed with a MAT PRINT statement is separated from
the following element by spaces according to the separator (comma or semicolon) that follows the <ar-
ray-name> in the list of <array-name>s, or by a comma separator if the last separator is not
specified.

An end-of-line character is generated prior to any characters generated by a MAT PRINT statement
if the current line of output is nonempty. An end-of-line character is also generated between the output
for successive arrays in the list of <array-name>s.

Example of the use of the MAT PRINT statement:

```
OPTION BASE 1
MAT READ A$(3,3)
MAT PRINT A$; A$,
DATA ONE, "2", THREE, "4", FIVE, "6", SEVEN, "8", NINE
PRINT "----*----1----*----2----*----3----*----4----*----5"
```

Execution of this example produces the following output:

```
ONE2THREE
4FIVE6
SEVEN8NINE

ONE             2               THREE
4               FIVE            6
SEVEN           8               NINE
----*----1----*----2----*----3----*----4----*----5
```

Example of MAT PRINT with different separators:

```
OPTION BASE 1
MAT READ A (4,4)
MAT PRINT A
PRINT "----*----1----*----2----*----3----*----4----*----5"
MAT PRINT A,
PRINT "----*----1----*----2----*----3----*----4----*----5"
MAT PRINT A;
DATA 11,12,13,14,21,22,23,24,31,32,33,34,41,42,43,44
```

Execution of this example causes the following to be displayed.

```
11                12                13                14
21                22                23                24
31                32                33                34
41                42                43                44
----*----1----*----2----*----3----*----4----*----5
11                12                13                14
21                22                23                24
31                32                33                34
41                42                43                44
----*----1----*----2----*----3----*----4----*----5
11  12  13  14
21  22  23  24
31  32  33  34
41  42  43  44
```

FILE I/O STATEMENTS

An external file is a collection of data stored on disk. BASIC has the capability of manipulating external files. The statements and concepts necessary to use external files are described in this subsection.

File Access

The OPEN and CLOSE statements are provided to enable access to external files. A particular BASIC environment may access, for input only, any external disk file maintained by MCPII which may be accessed under the currently logged on usercode. A particular BASIC environment may create or change only files which have a family name of the currently logged on usercode. Thus, the rules for accessing a file for input only are more flexible than those for files which are to be created or changed. If usercodes are not used, the scope of access of the BASIC environment is limited to public files.

OPEN Statement

The OPEN statement makes an external file accessible to a program by establishing the connection between the physical file (on disk) and the channel number within the program.

Syntax:
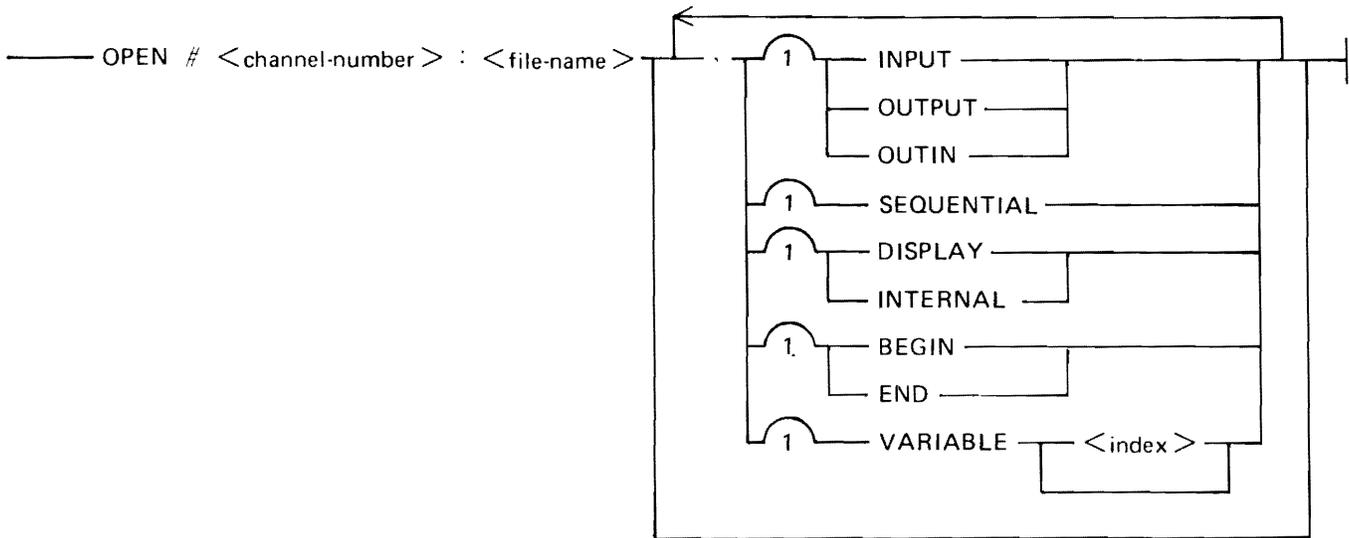
G18062

\<channel-number\> is a numeric expression which must evaluate to an integer in the range 0 to 255.
\<index\> is a numeric expression which gives the maximum allowable length for a record of a file.
\<file-name\> is a string expression which, when evaluated, is the name of a disk file. The value of
\<file-name\> can take the following form.

Syntax for \<file-name\>:
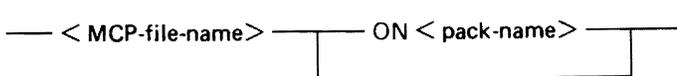
Examples of values for \<file-name\> follow.

```
"FILE1"
"FILE1 ON PACKB"
"(IBASIC)/DATA ON PACKA"
"MULTIID/FILEID ON MYPACK"
```

Semantics:

Through use of the OPEN statement, disk files can be assigned to a channel, and can then be accessed
in a program by referencing the assigned channel. If the file being accessed already exists, that file
is assigned to the channel. If the file does not exist, a new file is created and assigned to the channel.
A maximum of 16 files can be opened at one time in a BASIC program.

At the beginning of program execution, all channels except channel zero are inactive, that is, no file
is assigned to them. Channel zero is always open during the execution of a program. The file associated
with channel zero is the terminal from which IBASIC is executed. This file is the source of data for
INPUT statements and the destination of output from PRINT statements. The appearance of channel
zero in an OPEN statement is ignored. The appearance of a nonzero channel in an OPEN statement
that is already active causes a fatal error.

The keywords that can be listed after the <file-name> are called the file attributes. File attributes specify logical characteristics of a file and the manner in which a file is to be accessed by a program. If the file to be opened does not match the file attributes, the file is not opened and a fatal error occurs. The file attributes that can be specified are access mode, file organization, file type, file pointer position, and record type.

The access mode specifies the manner in which data in the file are accessed. The possible modes of access are INPUT, OUTPUT, and OUTIN, as can be seen from the syntax diagram. If INPUT is specified, it is only possible to read from the file. If OUTPUT is specified, it is only possible to write to the file. If OUTIN is specified, it is possible to read from and to write to the file. If no access mode is specified, the file is opened OUTIN.

The access mode of the terminal, channel zero, is OUTIN.

File organization is the logical organization of records within a file. Currently, there is only one organization possible: SEQUENTIAL. A sequential file is a linearly ordered sequence of records, accessible in sequential order. If no file organization is specified, the file organization is SEQUENTIAL.

The file organization of the terminal, channel zero, is SEQUENTIAL.

File type specifies the format of data in a record of a file. Two types are available: DISPLAY format and INTERNAL format. In a DISPLAY format file, each record is a string of characters. The only file organization provided for DISPLAY format files is SEQUENTIAL. In an INTERNAL format file, each record contains a sequence of numeric or string values. An end-of-record delimiter (NUL or physical end of the record) separates records in the INTERNAL format file, but is not part of the record. If no file type is specified, the type of the file is DISPLAY format.

The file type of the terminal, channel zero, is DISPLAY format.

A file pointer position specifies the initial position of the file pointer when the file is opened. This pointer indicates the position in the file that is affected by the next file input or output statement. The possible choices for the file pointer are BEGIN or END. If the pointer position is END, the pointer is positioned at the end of the file, the position immediately following the last record of the file. If the pointer position is BEGIN, the pointer is positioned at the beginning of the first record of the file, which is also the end of the file if it contains no records.

If the pointer position is not specified, the position is assumed to be the beginning of the file if the access mode is INPUT or the end of the file if the access mode is either OUTPUT or OUTIN.

A record type specifies the type and maximum length of records in a file. Currently, the only record type available is VARIABLE. A VARIABLE type record contains records whose lengths may be any value between 0 and <index>. The length of a record in a DISPLAY format file is the number of characters in that record. The length of a record in an INTERNAL format file is either (1) <index> * 5 bytes if <index> is specified following the VARIABLE attribute, (2) the size of the records in an existing file, or (3) 180 bytes if the file does not already exist and no <index> is specified following the VARIABLE attribute. The length of each numeric item in the record is five bytes. The length of each string in the record is the number of characters in the string plus 1.

If no record type is specified, the type of records in the file is VARIABLE. The maximum length of the records in the file is either 180 bytes or the record size of the file if it already exists before execution of the corresponding OPEN statement.

The record type for the terminal, channel zero, is VARIABLE with a record length corresponding to a default value defined by the type of terminal attached. For example, for a TD830 the record length is 80.

NOTE
A record is variable only to IBASIC. A file actually appears as a fixed record length file on the B 1000 system. Also, the end-of-record delimiter generated by IBASIC is not recognized as such by the B 1000 system, so that old data in a record after the end-of-record delimiter are invisible to IBASIC but visible to the B 1000 system.

Examples of OPEN statements:

```
OPEN #1: "MYFILE"
OPEN #1: "MYFILE", BEGIN
OPEN #2: "RESULT", OUTPUT, VARIABLE 132
OPEN #N: A$, SEQUENTIAL, DISPLAY, OUTIN, BEGIN
```

CLOSE Statement

The CLOSE statement closes the file assigned to the specified channel.

Syntax:

——— CLOSE # <channel-number> ————————————————————————————————|

G18063

<channel-number> has the same syntax as the <channel-number> in an OPEN statement: a numeric expression that must evaluate to an integer in the range 0 to 255.

Semantics:

Execution of a CLOSE statement closes the file assigned to the specified channel, causing the channel to become inactive. All files still assigned to channels when execution of a program terminates are closed. It is possible to close a file and then reopen it in the same program.

If an inactive nonzero channel appears in a CLOSE statement, a fatal error occurs.

Examples of CLOSE statements:

```
CLOSE #3
CLOSE #N
```

File I/O Statements

IBASIC provides for input from and output to disk files, and for end-of-file testing on these files through the use of file input and output statements. The statements necessary for the capabilities previously mentioned are described in this section under the headings File Input, File Output, and Exception Statement.

File Input

File input statements enable the user to obtain input from files. Three statements provide this capability: 1) the file INPUT statement, 2) the file LINPUT statement, and 3) the file MAT INPUT statement.

File INPUT Statement

The file INPUT statement allows data to be transferred from a disk file to variables within a program.

Syntax:

```
──────── INPUT #  <channel-number>  :  ──┬── <variable> ──┬──────────────────────────┤
                                         │   ◄──── , ────  │
                                         │                 │
                                         └───── , ─────────┘
```
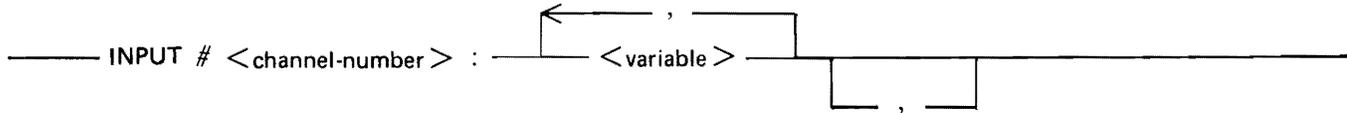
G18064

<channel-number> is a numeric expression which must evaluate to an integer in the range 0 to 255, and specifies the channel through which data transfer takes place. <variable> is any numeric or string variable as described in Sections 4 and 5, respectively. The list of <variable>s specifies the variables within the program that are to receive data.

Semantics:

If channel zero is specified in a file INPUT statement, the statement is executed as if no <channel-number> were specified; it performs the same function as the INPUT statement for terminal I/O. If a nonzero <channel-number> is specified, execution is similar to the INPUT statement for terminal I/O, except that no prompt is transmitted and no error occurs if all values are not supplied in a single record of the file. If an inactive channel is specified, a fatal error occurs.

Each time a value is required from a file, the datum which begins at the current position of the file's pointer is used to supply that value. The type of this datum must correspond to the type of the variable to which it is to be assigned. Overflow and underflow are handled in the same manner as for assignment statements. After the value has been supplied, the file pointer is advanced to the beginning of the next datum in the record, if more data follow; otherwise, the pointer is advanced to the beginning of the next record. If there is insufficient data in a file to satisfy an INPUT request, a fatal error occurs.

If the file pointer is not at the beginning of a record following execution of a file INPUT statement which does not end with a final comma, the pointer is advanced to the beginning of the next record in the file. If the statement does end with a comma, the file pointer remains where it was until subsequent I/O operations take place. A pointer positioned in the middle of a record is advanced to the beginning of the next record if an output statement is executed on that file.

Examples of file INPUT statements:

```
INPUT #1: X
INPUT #N: X, A$, Y(2)
INPUT #N + 1: X,Y,
```

File LINPUT Statement

The file LINPUT statement enables an entire record, including embedded spaces, commas, and quotation marks to be assigned as the value of a string variable. The syntax for the LINPUT statement follows.

Syntax:

—————— LINPUT # <channel-number> : <string-variable> •————————————————————— —————— ———————|

G18065

<channel-number> is a numeric expression which must evaluate to an integer in the range 0 to 255, and specifies the channel through which data transfer takes place. <string-variable> is any string variable as described in Section 5. <string-variable> receives the data as a result of execution of the LINPUT statement.

Semantics:

If channel zero is specified in a file LINPUT statement, the statement is executed as if no <channel-number> were specified; it performs the same function as the LINPUT statement for terminal I/O. If a nonzero channel is specified, execution is similar to the LINPUT statement for terminal I/O, except that no prompt is transmitted and the string of characters starting at the current position of the file's pointer and continuing to the end of the record is assigned as the value of <string-variable>. Following this assignment, the file's pointer is positioned at the beginning of the next record in the file.

If LINPUT is requested from an INTERNAL format file, a fatal error occurs.

Examples of file LINPUT statements:

    LINPUT #1: A$
    LINPUT #N: A$

File MAT INPUT Statement

The file MAT INPUT statement reads data from disk files. The syntax and semantics of the file MAT INPUT statement follow.
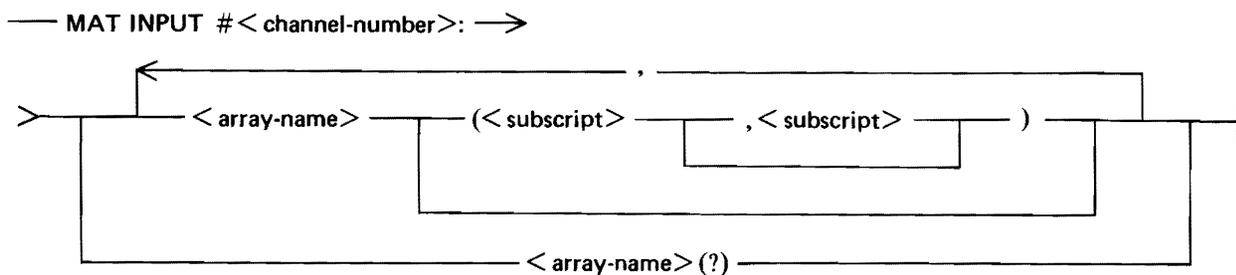
Syntax:

—— MAT INPUT # <channel-number>: ——>

<channel-number> is a numeric expression which must evaluate to an integer in the range 0 to 255, and designates the file to be accessed. <array-name> follows the same naming conventions as used for simple variables. <subscript> is a numeric expression. This numeric expression must evaluate to a number which is greater than or equal to the lower-dimension bound for arrays in the program. <array-name> (?) specifies a variable-length vector.

Semantics:

In a file MAT INPUT statement, values are assigned to the specified arrays from data in the file assigned to the specified channel. Input from channel zero is executed as if no channel were specified.

When input is received from a nonzero channel, no prompts are output. It is not necessary that all the values required by a file MAT INPUT statement be supplied by a single record of the file. Each time a file MAT INPUT requires a value from a file, the data element which begins at the current position of the file pointer is used to supply that value. The type of this data element must correspond to the type of the array element to which it is assigned. After the value has been supplied, the file pointer is advanced to the beginning of the next data element in the record if more data follow in the record, and to the beginning of the next record of the file otherwise.

Following the execution of a file MAT INPUT statement, the file pointer is advanced to the beginning of the next record in the file, unless the statement ends with a final comma (,) character. If the statement does end with a comma (,) character, the file pointer remains where it was until subsequent I/O operations take place. If the file pointer is positioned in the middle of a record, the file pointer is advanced to the beginning of the next record.

Examples of file MAT INPUT statements:

    MAT INPUT #1: A$
    MAT INPUT #K: B(3,3),C(?)
    MAT INPUT #K*2: X,Y(9)

File Output

File output statements write data to files. The two file output statements, the OUTPUT statement and the MAT OUTPUT statement, are described in the following paragraphs.

OUTPUT Statement

The OUTPUT statement allows data to be transferred from variables within a program to a disk file. The syntax and semantics of the OUTPUT statement follow.

Syntax:
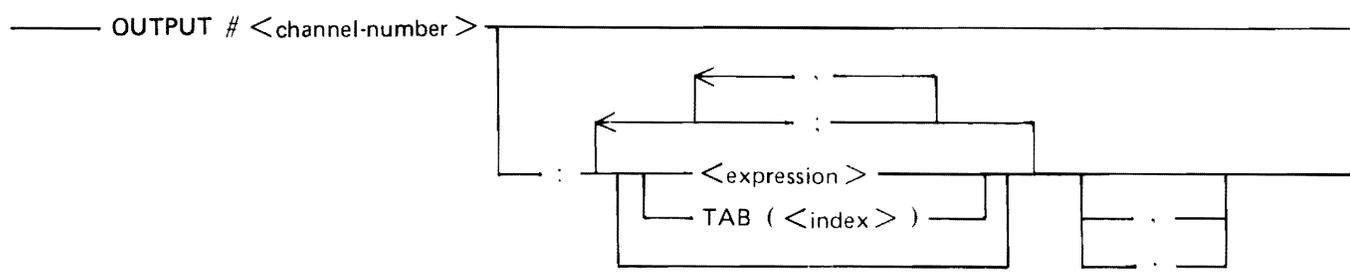
G18066

<channel-number> is a numeric expression which must evaluate to an integer in the range 0 to 255, and specifies the channel through which data transfer takes place. <expression> is any numeric or string expression. The list of <expression>s or TABs specifies the items which are written when the OUTPUT statement is executed. <index> is a numeric expression which, when evaluated, becomes the position within the file where the next <expression> is printed.

Semantics:

The execution of the OUTPUT statement is similar to the execution of the PRINT statement. The following paragraphs explain the differences between the two statements.

If the <channel-number> is nonzero, the output media is disk instead of terminal.

The end-of-line character is the end-of-record for the file.

The margin is the record length and the columnar position is one more than the number of characters generated since the last end-of-record.

The OUTPUT statement may be used with a file whose format is either DISPLAY or INTERNAL.

Output to a file is appended to the file starting at the current position of the file's pointer. Any data previously in the file beyond the file pointer are lost. When an output operation is complete, the file pointer is positioned at the end of the file.

When an OUTPUT statement that ends with a comma or a semicolon is executed, the last record transmitted to the file has no end-of-record. However, if input is requested from a file left in this state, if the file is closed, or if its pointer is reset to the beginning of the file, an end-of-record is appended to the file before the requested operation is performed.

A fatal error occurs if the length of a string written to an INTERNAL format file exceeds the maximum record length of that file.

Examples of file OUTPUT statements:

```
OUTPUT #N
OUTPUT #N: "X EQUALS"; X
OUTPUT #3: TAB(10); A$; "IS DONE."
```

MAT OUTPUT Statement

The MAT OUTPUT statement writes data to disk files. The syntax and semantics of the MAT OUTPUT statement follow.
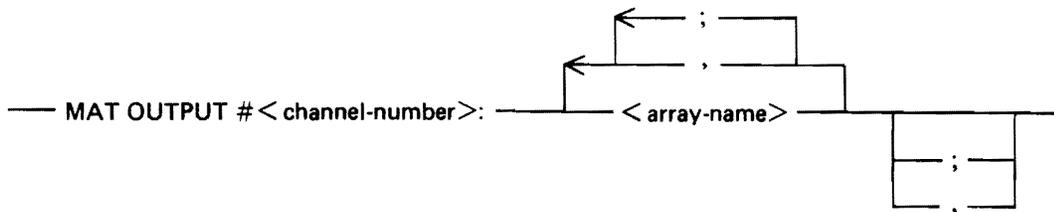
Syntax:

<channel-number> is a numeric expression which must evaluate to an integer in the range 0 to 255, and designates the file to be accessed. <array-name> follows the same naming conventions as used for simple variables.

Semantics:

If the file assigned to the channel specified in the MAT OUTPUT statement is a display format file, execution of that statement writes a string of characters including end-of-record characters to the file. If the output is to go to channel zero, this string is generated exactly as in the execution of a MAT PRINT statement. If the output is to go to another channel, the same rules apply, with the end-of-line, margin, and columnar position being treated as follows. The end-of-line is the end-of-record for the file, the margin is the record length of the file, and the columnar position is one more than the number of characters generated since the last end-of-record character.

Output to a file is appended to the file starting at the current position of the file pointer; any data previously in the file following that position are lost. When output is complete, the file pointer is positioned at the end of the file.

If the file assigned to the channel specified in a MAT OUTPUT statement is an internal format file, execution of that statement generates a sequence of values for transmission to the file, with each value being followed by an end-of-record character. The sequence of values is the same as if output were to a display format file.

Examples of MAT OUTPUT statements:

    MAT OUTPUT #1: A$
    MAT OUTPUT #K: B; C,
    MAT OUTPUT #K*2: X;

Exception Statement

The exception statement allows programmatic action when the end of a file is encountered.

Syntax:

──────── AT EOF # <channel-number> THEN <line-number> ───────────────────────────────┤

G18067

<channel-number> is a numeric expression which must evaluate to an integer in the range 0 to 255, and specifies the channel through which data transfer takes place. <line-number> is a line number which specifies the place where execution continues if an end-of-file condition occurs for the file specified by <channel-number>.

Semantics:

An end-of-file condition occurs when the amount of data remaining in the file is not enough to satisfy a request for input from that file, or when the physical end of a finite-capacity file is reached before output to that file is completed. Execution of an exception statement specifies the line number at which execution continues whenever an end-of-file occurs during a data transfer operation on the specified channel. The statement itself does *not* test for an end-of-file condition existing, nor does it branch, that is, transfer control to a line number other than the one next in sequence, it only specifies the action to be taken when an end-of-file condition is detected. If no exception statement is executed for a given channel prior to the occurrence of an end-of-file condition for that channel, a fatal error results. If more than one AT EOF statement is executed for a <channel-number>, the latest one executed takes precedence if an end-of-file condition occurs.

The effect of the exception statement is nullified if the file assigned to <channel-number> is closed.

Example of the use of an exception statement:

```
100 AT EOF #2 THEN    !Go to line 400 when end-of-file
400                   ! is detected.
200 INPUT #2: A, B$
300 GOTO 200
400 STOP
```

File Control Statements

File control statements are provided to: control the position of a file pointer for an open file, erase the contents of a file, and inquire about certain attributes of a file. There are three statements that accomplish these tasks: 1) the file RESTORE statement, 2) the SCRATCH statement, and 3) the IN-QUIRE statement.

File RESTORE Statement

The file RESTORE statement resets the pointer for a file to the beginning of the file.

Syntax:

```
———— RESTORE # <channel-number> ——————————————————————————————|
G18068
```

<channel-number> is a numeric expression which must evaluate to an integer in the range 0 to 255.

Semantics:

Execution of a file RESTORE statement resets the pointer for the file assigned to the specified channel to the beginning of the file.

Examples of file RESTORE statements:

```
RESTORE #I
RESTORE #15
```

SCRATCH Statement

The SCRATCH statement erases the contents of a file and resets the pointer to the beginning of the file.

Syntax:

```
———— SCRATCH # <channel-number> ——————————————————————————————|
G18069
```

<channel-number> is a numeric expression which must evaluate to an integer in the range 0 to 255.

Semantics:

Execution of a SCRATCH statement erases the contents of the file assigned to the specified channel and resets the pointer for that file to the beginning of the file. If the <channel-number> is zero, no action occurs.

Examples of SCRATCH statements:

    SCRATCH #2
    SCRATCH #Z

INQUIRE Statement

The INQUIRE statement solicits information concerning a file. The syntax and semantics of the INQUIRE statement follow.
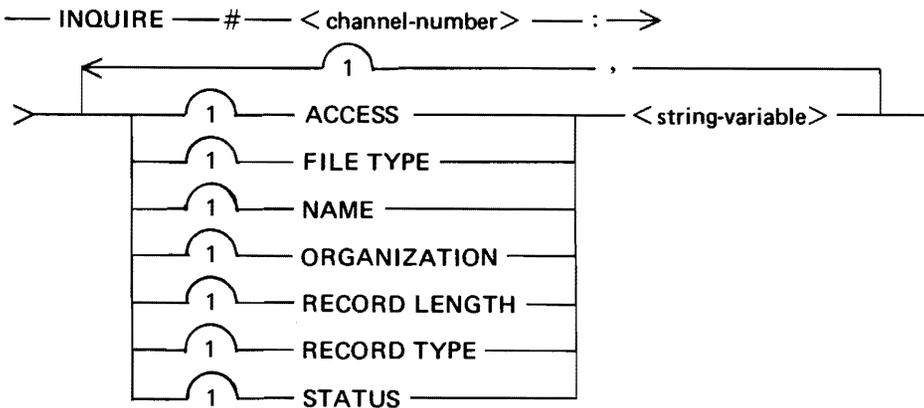
Syntax:

<string-variable> is a string variable as described under String Variables in Section 5. <channel-number> is a numeric expression which must evaluate to an integer in the range 0 to 255, and designates the file to be accessed. The items included in a complete circuit of this syntax diagram are referred to as an inquiry item.

Semantics:

Execution of an INQUIRE statement causes the <string-variable>s in the inquiry items to be assigned values corresponding to the attributes (such as ACCESS, NAME, and STATUS) of the file currently assigned to the specified channel. Each <string-variable> is assigned a value corresponding to the attribute which immediately precedes it. If the channel is inactive, all <string-variable>s are assigned the value NONE. The values corresponding to the attributes are shown in the following table.

Attribute	Values
ACCESS	INPUT, OUTPUT, or OUTIN
FILE TYPE	DISPLAY, or INTERNAL
NAME	Name of the file assigned to the channel
ORGANIZATION	SEQUENTIAL
RECORD LENGTH	STR$(N), N is the maximum length of records in the file
RECORD TYPE	VARIABLE
STATUS	OPEN, INPUT, OUTPUT, SCRATCH, RESTORE, or END OF FILE. Value assigned depends on most recent file operation

Examples of INQUIRE statements:

    INQUIRE #3: NAME N$, ACCESS A$, FILE TYPE T$
    INQUIRE #N: STATUS S$
    INQUIRE #2: FILE TYPE Z$(1:8), STATUS Z$(10:25)

SECTION 10
DEBUGGING AIDS

Three statements are provided in BASIC for the purpose of debugging a program. They are the DEBUG statement, the BREAK statement, and the TRACE statement.

DEBUG STATEMENT

The DEBUG statement either activates or deactivates the debugging facilities in BASIC. The statements available to the user when debugging is active are the BREAK and the TRACE statements. The syntax for the DEBUG statement follows.

Syntax:

```
───── DEBUG ──┬── ON ──┬──────────────────────────────────────────────┤
              └── OFF ──┘
```

G18070

Semantics:

DEBUG ON activates debugging. DEBUG OFF deactivates debugging. The default is DEBUG ON.

Examples of DEBUG statements:

    DEBUG ON
    DEBUG OFF

BREAK STATEMENT

The BREAK statement causes program execution to be temporarily stopped. The syntax for the BREAK statement follows.

Syntax:

```
───── BREAK ──────────────────────────────────────────────────────────┤
```

G18071

Semantics:

When debugging is active and a BREAK statement is executed, program execution is stopped and a message is sent to the terminal. This message informs the user that a break has occurred and gives the line number where execution stopped. Execution may be continued by the execution of a CONTINUE or a STEP command. The STEP and CONTINUE commands are described in Section 11.

If debugging is not active, the BREAK statement has no effect.

TRACE STATEMENT

The TRACE statement causes each source line (a line of source code) of a program to be displayed as it is executed.

Syntax:

```
——— TRACE ———┬— ON ———┬————————————————————————————┥
              └— OFF ——┘
```
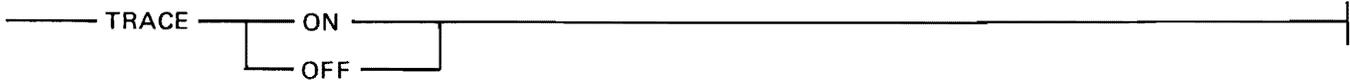
G18072

Semantics:

When debugging is active and a TRACE ON statement is executed, tracing is turned on. When tracing is on, each source line is displayed on the terminal as it is executed. TRACE OFF turns tracing off. If debugging is not active, the TRACE statement has no effect.

NOTE
If TRACE ON is in effect and DEBUG is switched off, tracing continues until DEBUG ON is set and a TRACE OFF statement is executed.

Examples of TRACE statements:

TRACE ON
TRACE OFF

Example of TRACE ON output:

```
90 IF A = B THEN 150
100 IF A < B THEN 130
130 B = B – A
140 GO TO 90
90 IF A = B THEN 150
100 IF A < B THEN 130
130 B = B – A
140 GO TO 90
90 IF A = B THEN 150
150 PRINT "THE GREATEST COMMON DIVISOR OF;C;"AND";D;"IS";A
THE GREATEST COMMON DIVISOR OF 10 AND 5 IS 2
160 PRINT

170 PRINT "DO YOU WANT TO CONTINUE? Y OR N"
```

SECTION 11
SYSTEM COMMANDS AND CAPABILITIES

System and editing commands are provided to allow the user to interact with IBASIC in Command mode.

SYNTAX DEFINITIONS

Several of the system commands described in this section require the use of the constructs explained in this subsection. These constructs are line number range, BASIC file name, pack name, and MCP file name.

Line Number Range

A line number range allows a successive group of lines to be specified in a command.

Syntax:
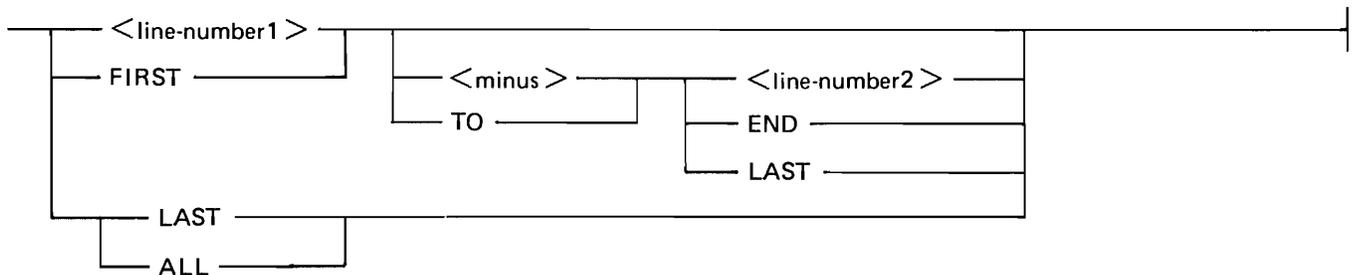
G18073

The line number syntax is diagrammed under Statement Lines in Section 3. <minus> is a minus sign (–).

Semantics:

<line-number2> must be greater than or equal to <line-number1>. Also, the pseudo line number FIRST, which refers to the first line in the program, must refer to a line number smaller than or equal to <line-number2>. The pseudo line number LAST (or END) must refer to a line number greater than or equal to <line-number1>. The special cases of FIRST TO LAST or ALL refer to the entire program.

Examples of valid line number ranges:

```
100-200
FIRST TO
3000
2000 – LAST
FIRST
ALL
FIRST TO 200    ! First lines < = 200.
```

Examples of invalid line number ranges:

```
200-100
3000 TO
FIRST
LAST TO
2000
END  −  100
FIRST  TO  200      ! Where the first line  > 200.
```

BASIC File Name

Syntax:
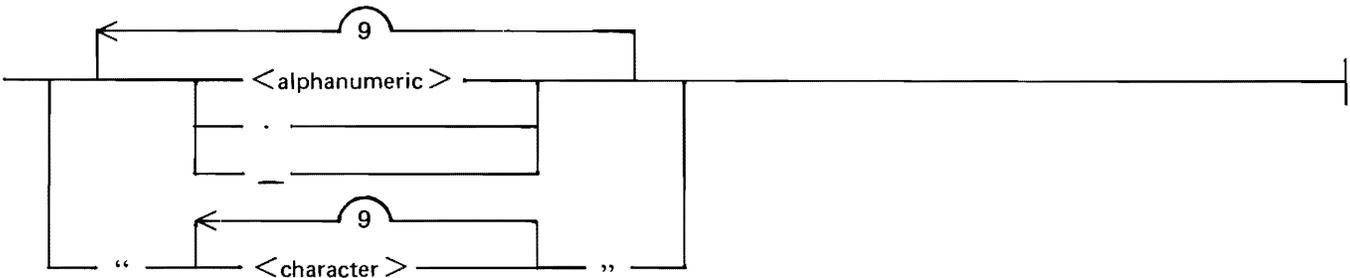
G18074

<character> is any character valid to the MCP. <alphanumeric> is an alphabetic character or decimal digit.

A BASIC file name, that appears inside quotation marks, can contain a quotation mark by using two consecutive quotation marks instead of one (for example, "MY""FILE"). A BASIC file name, that does not use quotation marks, is translated to upper-case characters and must begin with an alphanumeric character.

A BASIC file name cannot begin with an asterisk (*), space ( ), or equal sign (=). The file name can begin with a number sign (#) as long as the name is within quotation marks. The first and last characters of a BASIC file name cannot be a left parenthesis and a right parenthesis, respectively.

Examples of valid BASIC file names:

```
ABCD
1234
P.Q
"!""#$"
Q123ABC
abcd (equivalent ABCD)
"abcd"
c---d
```

Examples of invalid BASIC file names:

    ABCDEFGHIJK
    !"#$
    .ABC
    "qwe
    *ABC
    (ME)
    "(ME)"
    - - - -

Pack Name

Syntax:

```
    ┬── <BASIC-file-name> ─────┬──────────────────────────────────────┤
    └── " <up to 10 blanks> " ─┘
```
G18075

A pack name can contain up to 10 characters according to the same formation rules as described under BASIC File Name in this section. Spaces for the pack name indicate the system pack.

Examples of valid pack names:

    USER.243
    "1&2"
    " "

Examples of invalid pack names:

    (ME)
    = or "="
    "   "

MCP File Name

Syntax:
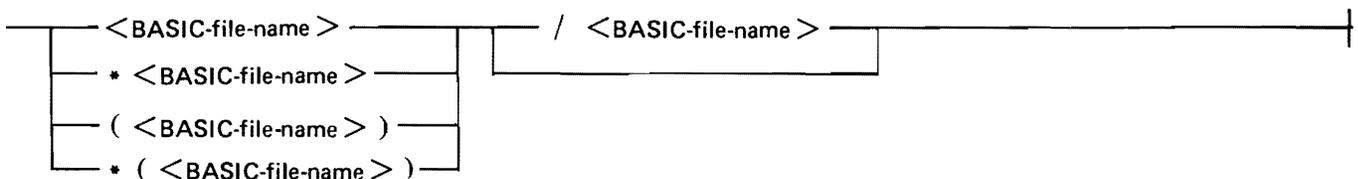
```
    ┬── <BASIC-file-name> ─────┬──┬── / <BASIC-file-name> ──┬──────────┤
    ├── * <BASIC-file-name> ───┤  └─────────────────────────┘
    ├── ( <BASIC-file-name> ) ─┤
    └── * ( <BASIC-file-name> )┘
```
G18076

The use of quotation marks in a <BASIC-file-name> needs further explanation in relation to an MCP file name. If the first BASIC file name uses quotation marks, they must enclose any asterisk, and/or parentheses used.

Examples of valid MCP file names:

```
(IBASIC)/PQR
XYZ
XYZ/P.Q.R
*QWER
*(QWER)/LKJ
"*AS$F"
"*(qwer)"/"zxcv"
X_Y/L.K
```

Examples of invalid MCP file names:

```
"(IBASIC)/PQR"
*"QWER"
X$/Y$
X/=
" ASD"
"(qwe"
=
=/=
```

SYSTEM COMMANDS

System commands can be entered at any time, although their effect may depend upon the state of the BASIC environment. Unless stated to the contrary, the effect of the command is immediate. These commands cannot be preceded by a line number nor can they be imbedded within a BASIC statement or BASIC command. The system commands are listed in alphabetical order.

BYE Command

The BYE command causes the BASIC environment to be cleared and the IBASIC program to go to EOJ, unless the IBASIC program was initiated under SMCS with an initial parameter or NO EOJON-LOGOUT, in which case the IBASIC program waits for another user to sign on (refer to Initial Parameters in appendix C). The state of the BASIC environment is examined and the IBASIC program does not go to EOJ if either of the following conditions is true: (1) the source file is not saved, or (2) the BASIC environment is running.

Syntax:

```
——————— BYE ——————————————————————————————————————————|
```
G18077

Semantics:

The action of the BYE will not occur if either of the following conditions applies: the source file is not saved or the BASIC environment is running.

11-4

CONTINUE Command

The CONTINUE command causes the BASIC environment to continue execution from wherever it last stopped.

Syntax:
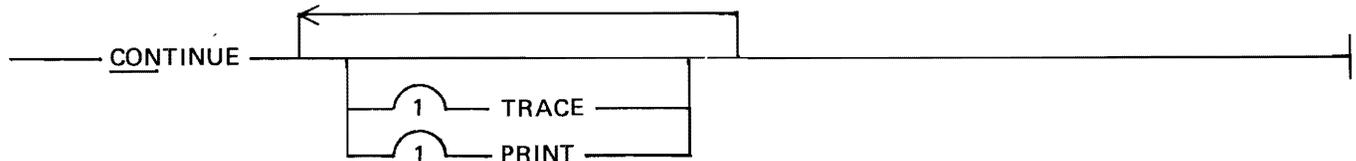
G18078

Semantics:

When a CONTINUE command is entered, the BASIC data environment is not cleared, that is, data variables retain the values they had when the BASIC environment last stopped. A RUN or WALK command must precede the use of the CONTINUE command.

Certain editing functions, for example, deletion of the next statement to be executed, changing the dimensions of an array, or changing a FOR NEXT statement, cause the CONTINUE command to be disallowed until after a RUN or WALK command is executed.

The TRACE option causes the statements to be displayed on the remote device as they are executed.

The PRINT option causes the output from BASIC PRINT statements and the input from BASIC IN-PUT and LINPUT statements to be written to the file LINE and to the remote device. The file LINE is closed on execution of an END statement in the BASIC program.

If both the PRINT and TRACE options are specified, the trace of statements executed is directed to the file LINE interspersed with the BASIC PRINT, INPUT, and LINPUT output, and is not displayed on the remote device.

The PRINT and TRACE options have effect until the program is stopped by a STOP statement, an END statement, or by a fatal error.

The CONTINUE command has effect only when the BASIC environment is in a stopped state.

Examples of CONTINUE statements:

CON	
CON TRACE	! Continue and commence tracing.
CONTINUE	
CONTINUE TRACE	! Continue and commence tracing.
CON PRINT TRACE	! Continue, and commence or continue ! writing to file LINE.
CON PRINT	! Continue, commence tracing, and ! writing to file LINE.

DELETE Command

The DELETE command deletes the specified line numbers from the workfile file.

Syntax:

```
 ┬── DELETE  <line-number-range> ──┬────────────────────────────────────┤
 └── <line-number> ────────────────┘
```

G18079

<line-number> is described under Lines in Section 3. <line-number-range> is described under Line Number Range in this section. <line-number> and <line-number-range> must specify lines that actually exist.

Semantics:

If the deleted line would have been the next line executed, program execution can be initiated only by entering a RUN or WALK command.

The line numbers requested must be present in the workfile. No deletion occurs if any requested line number is not present in the workfile.

If the BASIC environment is running, the DELETE command is ignored.

Examples of DELETE commands:

```
                  DELETE  123              ! Delete line 123 only.
                  DELETE  2000-3000
                  123                      ! Delete line 123 only.
                  DELETE  4000  TO  LAST
                  DEL  FIRST
                  DEL  FIRST  TO  100
                  DEL  FIRST  TO  LAST     ! Delete all program
                                           ! statements.
```

EDIT Command

The EDIT command initiates edit mode. The syntax and semantics of the EDIT command follow.

Syntax:

```
── EDIT ──┬─────────────┬──┬───────────────────────────────┬──┤
          └── ERRORS ────┘  ├──────────────── <line-number> ─┤
                            └── FROM ──┘
```

Semantics:

In edit mode, each line of a program is presented one line at a time for review and possible change. Editing of a program can begin either at the beginning of the program or at a particular point (line number) within the program. Edit mode is terminated when either the end of the program is reached, by explicitly using the pseudo break feature, or by entering a command instead of a statement. If the line number is changed in the process of editing a statement, the next line after the new line is displayed for editing.

A variant of edit mode is the edit-errors mode. In this mode, the next line which contains a syntax error is displayed for editing.

The EDIT command is only valid from a screen type terminal, such as the MT983 or the TD830 series terminal.

Examples of the EDIT command:

    EDIT
    EDIT ERRORS
    EDIT FROM 23450

FILE Command

The FILE command provides the capability to obtain information about a disk file. The syntax and semantics of the FILE command follow.

Syntax:

G18080
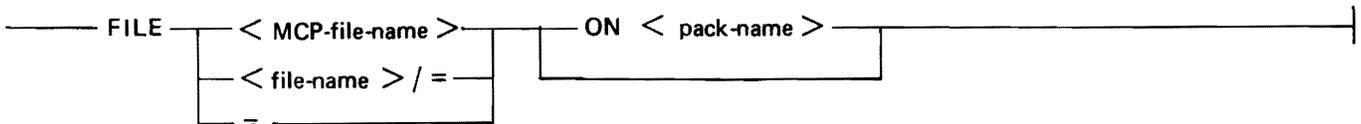
< MCP-file-name > and < pack-name > are described under Syntax Definitions in this section. < file-name > is constructed according to the rules for the family name portion of < MCP-file-name > .

Semantics:

The FILE command scans the disk directories available to the current usercode (refer to Scope of File Access in the Glossary) to determine whether the file exists. If the file exists, information about it is returned. If no < pack-name > is specified, the default pack associated with the current usercode is queried. If the < file-name > / = option is used, a sorted list of files with a family name of < file-name > is returned.

If the " = " option is used, a sorted list of the files in the current directory is returned.

Examples of FILE statements:

```
FILE IBASIC
FILE *XY
FILE X/Y ON P
FILE (PQR)/ZXY ON USER
FILE BLACKJACK ON ""        ! Look for BLACKJACK on
                            ! system pack.
```

FIND Command

The FIND command is equivalent to the XREF command. Refer to the XREF command in this section.

FIX Command

The FIX command enables the user to change portions of one or more lines without re-entering the entire line.

Syntax:

```
——— FIX ┬─────────────────────────┬─ <delim> <text> <delim> <new-text> ┬───────────────┬──|
         └─ <line-number-range> ──┘                                     └─ <delim> ──┘
```

G18081

<line-number-range> is described under Syntax Definitions in this section. <delim> can be any character other than A through Z, 0 through 9, or space. <text> may be any string of characters excluding <delim> and may contain no characters, in which case <new-text> is inserted after the line number of the line scanned, and the rest of the line is shifted right accordingly. <new-text> may be any string of characters excluding <delim>. If the trailing <delim> is omitted, any trailing blanks are not included in <new text>. <new-text> may contain no characters, implying that occurrences of <text> are deleted from the scanned line, and that the rest of the line is shifted to the left accordingly.

Semantics:

The FIX command searches the specified <line-number-range> for the occurrence of the required <text>, replaces each occurrence of <text> within each line scanned with the <new-text>, and displays the new line. If <text> and <new-text> are not the same length, the rest of the line is shifted appropriately.

If the <line-number-range> is omitted, the whole program is scanned.

If a statement is changed, the modified statement is checked for syntax and included in the current file. As a result, the previous line having the line number of the modified statement is overwritten. Any syntax errors that result must be resolved before the program can be run.

If the resulting statement exceeds 256 characters, the excess is truncated. No warning message is given.

Examples of FIX commands:

FIX /3.14159/PI

FIX 100 .X/Y.Y/X

FIX LAST /END/ END   ! Move the END statement
                      ! three characters to the right.

FIX 2230 TO 2500 :::   ! Make lines 2230 through 2500
                       ! comment lines.

GET Command

The GET command allows BASIC source files to be loaded into the BASIC environment.

Syntax:

```
———— GET  <MCP-file-name> ———┬——————————————————————————————————————┤
                             └—— ON <pack-name> ——┘
```

G18082

<MCP-file-name> and <pack-name> are described under the heading Syntax Definitions in this section.

Semantics:

The GET command searches the directory for <MCP-file-name> and proceeds to load the file into the BASIC environment. If any syntax errors are found in the file, suitable error messages are emitted and the file cannot be run until these errors are fixed. If syntactically incorrect statements are listed with the LIST command, they are highlighted: reverse video on TD820 and TD830 terminal types, preceded by an asterisk (*) otherwise.

Only those files within the current usercode scope of access are available. The default pack for the current usercode can be overridden by using the ON option.

Various parameters of the file are checked to make sure that it is a BASIC source file; for example, file type is data, record size is less than or equal to 256 bytes, and number of records is less than the maximum allowed (1979 records). Any record in this file which does not start with a valid line number is not included in the loaded file. The records in the file do not have to be in strict line number sequence. The lines are entered in the workfile just as if they were being entered from the terminal.

Examples of GET commands:

GET BLACKJACK	! Load BLACKJACK from default ! pack.
GET PQR ON ""	! Load PQR from system pack.
GET *MYPROG	! Load MYPROG from system ! pack, bypassing usercode defaults.
GET (HIS)/FILE	! Access another user's file.
GET (MY)/FILE ON OTHER	! Load a file from a specific ! pack.

HELLO Command

The HELLO command allows a user to log on or off.

Syntax:
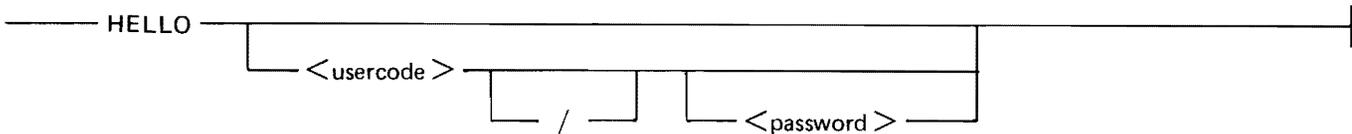
G18083

Formation of a <usercode> follows the same rules as for a <BASIC-file-name> except that the maximum length of a <usercode> is seven characters. Formation of a <password> follows the same rules as for a <BASIC-file-name>. The syntax for a <BASIC-file-name> is described in this section under Syntax Definitions.

Semantics:

If only "HELLO" is entered, a log off function is requested. Log off occurs only if the current BASIC environment permits. Refer to conditions for the BYE command in this section. If the HELLO command is allowed, the current BASIC environment is cleared, and only the HELLO, BYE, and TEACH commands are allowed thereafter.

If <usercode> and, optionally, <password> follow HELLO, an implicit log off of the current usercode is performed, if necessary, followed by a log on of the requested usercode/password pair.

The <usercode> and <password> pair must be in the SYSTEM/USERCODE file if IBASIC was executed under a privileged usercode. (Refer to the B 1700/B 1800 Systems System Software Operation Guide, Volume 2, form number 1108966 for more information on privileged usercodes.) If IBASIC is not running under an MCP usercode, <password> has no meaning and is not allowed (refer to Usercode Considerations in Appendix C).

HELLO is ignored if the IBASIC system is not privileged (refer to Usercode Considerations in Appendix C for more information on a privileged IBASIC system).

Examples of HELLO commands:

HELLO ME/ SECRET	! Log ME on.
HELLO	! Log off, if allowed.
HELLO MYOWN	! Log on, not running under ! the MCP usercode system.

LIST Command

The LIST command causes the requested lines or all of the current program to be listed at the remote terminal.

Syntax:

```
—————— LIST ——┬————————————————————————————————————┬——————————————————————|
              └—— <line-number-range> ——┘  └—— PRINT ——┘
```

G18084

<line-number-range> is described under Syntax Definitions in this section.

Semantics:

With the LIST command, any syntactically incorrect lines are highlighted by means of reverse video (for TD820 and TD830 terminals) or a preceding asterisk (*). If the line which is displayed would be the next line executed as a result of a CONTINUE or STEP command, it is highlighted by means of bright video (for TD830 terminals) or by a preceding greater than sign (>) (for all other terminal types).

If the PRINT option is requested, the list is written to the file LINE and not to the remote device. The file LINE is defined as a printer file.

Examples of LIST commands:

LIST	! List the whole program.
LIST FIRST TO 100	! List up to line 100.
LIST 1234	! List line 1234 only.
LIST PRINT	! List the current file on ! the printer.

MAKE Command

The MAKE command allows a file to be named and created.

Syntax:

──── MAKE ⟨BASIC-file-name⟩ ──┤

G18085

⟨BASIC-file-name⟩ is described under Syntax Definitions in this section.

Semantics:

The MAKE command clears the BASIC environment under the same restrictions as the BYE command and prepares for entry of a file to be called ⟨BASIC-file-name⟩. If ⟨BASIC-file-name⟩ already exists, the MAKE command is ignored.

If no MAKE command has been entered, IBASIC will still accept BASIC statements (line number present) and put them into the current workfile, but the file must be named (TITLE or SAVE AS command) before it can be saved.

Examples of MAKE commands:

```
        MAKE NEWPROG
        MAKE "ODDNAME?"      ! Quotation marks used because of
                             ! special character "?".
```

MERGE Command

The MERGE command allows BASIC source code (BASIC language statements) to be merged into the workfile.

Syntax:

──── MERGE ─┬─────────────────────────┬─ ⟨MCP-file-name⟩ ─┬──────────────────────┤
            └─ ⟨line-number-range⟩ ─┬─┘   └─ ON ⟨pack-name⟩ ─┘
                                    └─ FROM ─┘

G18086

⟨line-number-range⟩, ⟨MCP-file-name⟩, and ⟨pack-name⟩ are described in this section.

Semantics:

The MERGE command searches the directory for ⟨MCP-file-name⟩ and proceeds to load the requested portion of it (or the entire file if ⟨line-number-range⟩ is omitted) into the current BASIC environment. If a line that already exists is merged, the new merged line overwrites the existing line.

The merged file need not necessarily be in ascending line number sequence. The MERGE command searches the whole merged file, and if there are duplicate line numbers to be merged, the physically last in sequence is the one finally merged.

If a syntax error is found in a merged line, a suitable error message is displayed on the terminal, and runs of the file are inhibited until these errors are fixed. If those syntactically incorrect statements are listed with the LIST command, they are highlighted by reverse video on TD820 and TD830 terminal types, or preceded by an asterisk (*), otherwise.

Only those files within the current usercode scope of file access are available. The default pack for the current usercode can be overridden by using the ON option.

Various parameters of the file are checked to make sure that it is a BASIC source file; for example, file type is data, record size is less than or equal to 256 bytes, and number of records is less than the maximum allowed (approximately 2000 records).

Any record in the merged file which does not start with a valid line number is ignored.

Examples of MERGE commands:

MERGE OTHER/FILE   ! Merge the whole of OTHER/
                               ! FILE into the current environment.

MERGE 1000 TO 2000 FROM OTHER/ FILE

MERGE LAST OTHER/FILE ON OTHERPACK

PASSWORD Command

The PASSWORD command changes the password for the current usercode in the MCP usercode file.

Syntax:

——— PASSWORD <old-password> <new-password> <new-password> ———————————————————————|
G18087

Formation of <old-password> and <new-password> follow the same rules as for a <BASIC-file-name>.

Semantics:

The PASSWORD command is only valid if the IBASIC system runs under an MCP usercode. The new password must be entered twice identically to ensure correct and intentional entry.

Example of a PASSWORD command:

PASSWORD SECRET TOPSECRET TOPSECRET

Psuedo Break Feature

The pseudo BREAK feature can be used to prematurely terminate a function.

Syntax:

```
─────── .BRK ───────────────────────────────────────────────────────┤
G18088
```

To terminate a function, a special message of the precise form .BRK with no leading or trailing spaces can be entered, the SPCFY key can be depressed for CRT terminals, that is, display screen terminals, or the BREAK key can be depressed. (The BREAK key is supported for TTY-type devices and has the same effect as .BRK; however, the BREAK key is only effective on these devices during the printing of an output message.)

Semantics:

This pseudo BREAK feature may be used from any supported terminal type to terminate the following commands: LIST, RUN, WALK, CONTINUE, STEP, GET, SAVE, FIX, RENUMBER and a CHAIN statement or command.

In addition, a pseudo BREAK may be used to terminate the automatic recovery of a prior user session, the period of time during which a user is logged on. If a pseudo BREAK is used in this context, that session is thereafter irrecoverable. Refer to Recovery in this section for more information on automatic recovery.

.BRK may also be used to interrupt the execution of an INPUT or LINPUT statement which is currently soliciting a response from the remote terminal. In the case of CRT terminals, .BRK must be typed over the input prompt character in the top left portion of the screen. Interruption of an INPUT or LINPUT statement also implies changing the state of the BASIC environment from running to stopped.

RENAME Command

The RENAME command provides the capability of changing the name of a file.

Syntax:

```
─────── RENAME <BASIC-file-name1> ┬──────────────────────┬─┬──────┬─<BASIC-file-name2> ──────────┤
                                  └─ ON <pack-name> ──┘   └─ AS ─┘
G18089
```

<BASIC-file-name1>, <BASIC-file-name2>, and <pack-name> are described under Syntax Definitions in this section.

Semantics:

The RENAME command checks for the presence and availability of <BASIC-file-name1> on the requested pack, ensures the absence of <BASIC-file-name2>, and causes the MCP to change the name of <BASIC-file-name1> to <BASIC-file-name2>. Only files within the current user's scope of access for files may be changed.

Examples of RENAME commands:

RENAME ORANGES AS LEMONS ! On default pack.

RENAME PROG1 ON "" AS PROG ! Force change on system pack.

RENAME X Y

RENUMBER Command

The RENUMBER command is used to resequence the line numbers in a BASIC source program.

Syntax:

```
                                           ┌─────────────────────────────┐
                                           │                             ▲
──────── RENUMBER ─┬──────────────────────┬─┬───────────────────────────┬──────────────┤
                   └─<line-number-range>──┘ ├──( 1 )── STEP <increment>──┤
                                            └──( 1 )── AT <line-number>──┘
```

G18090

<line-number-range> is described under Syntax Definitions in this section. <line-number> is described under Statement Lines in Section 3. <increment> is an integer constant.

Semantics:

The RENUMBER command is used to resequence the line numbers of, and the references to, all or part of the currently loaded source program. The STEP parameter defines the amount to increment each new line number. The AT parameter defines the starting value for the new numbers. If the STEP parameter is omitted, a value of 10 is assumed. If the AT parameter is omitted, a value of 100 is assumed.

Before the actual renumbering is done, checks are performed to make sure that (1) no overlap of existing statements would occur, (2) the order of execution of statements is not changed, (3) a previously unresolved line number reference would not become implicitly resolved by the renumber process, and (4) the last line number in the program would not exceed 99999.

The RENUMBER command cannot be interrupted by the pseudo BREAK feature. If the system fails during a RENUMBER command, the workfile may be partially renumbered.

Examples of RENUMBER commands:

RENUMBER	! Renumber the whole ! program ! with implicit step 10 and ! starting with a value of 100.
RENUMBER 100 TO 1000	! Renumber the specified part ! of the program and all ! references to that part ! with the default parameters.
RENUMBER 5000 TO LAST AT 5000 STEP 100	! Renumber the ! last part of the program, ! incrementing the line ! numbers by 100.

RUN Command

The RUN command causes a program to be executed.

Syntax:
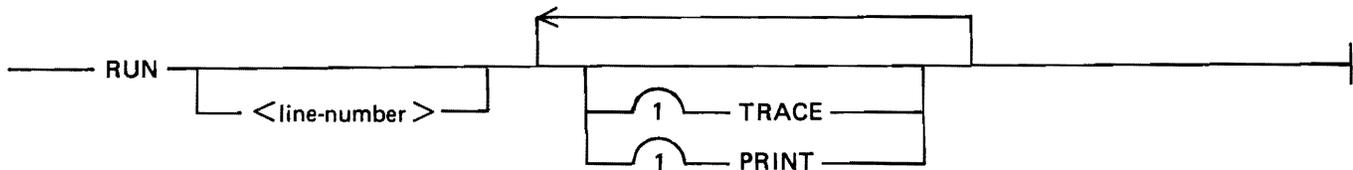
G18091

< line-number > is described under Statement Lines in Section 3.

Semantics:

The RUN command initiates the continuous execution of the statements in the current workfile, starting either from the first statement in the workfile, or from < line-number >, if specified. The BASIC data environment is cleared so that all numeric data items are zero and all string items have the value of the null string (""). The line number, if specified, must be a valid line number within the program. RUN < line-number > causes the program to be executed as if GOTO < line-number > were the first statement of the program.

The TRACE option causes the statements to be displayed on the remote device as they are executed.

The PRINT option causes the output from BASIC PRINT statements and the input from BASIC IN-PUT and LINPUT statements to be written to the file LINE as well as to the remote device. The file LINE is closed on execution of an END statement in the BASIC program.

If both the PRINT and TRACE options are specified, the trace of statements executed is directed to the file LINE interspersed with the BASIC PRINT, INPUT, and LINPUT output. In this instance, the traced statements are not displayed on the remote device.

The PRINT and TRACE options have effect until the program is stopped by a STOP or END statement or by a fatal error.

Examples of RUN commands:

RUN	! Run the program from its first ! executable statement.
RUN TRACE	! Run the program from its first ! executable statement and trace ! execution.
RUN 1234	! Run from line 1234 and clear the ! program's data variables.
RUN PRINT	! Print output to be written to file ! LINE.

SAVE Command

The SAVE command causes the current workfile to be saved on disk.

Syntax:

```
──── SAVE ┬─────────────────────┬──────────────────────┬─────────────────────┬───┤
          └─ AS <BASIC-file-name> ─┘   └─ ON <pack-name> ─┘   └─ FOR CANDE ─┘
G18092
```

<BASIC-file-name> and <pack-name> are described under Syntax Definitions in this section.

Semantics:

The SAVE command writes the current source program to a file on the current usercode default pack with the name of the current file. The pack can be overridden by the ON option and the name can be overridden by the AS option.

If the FOR CANDE option is used, an attempt is made to make the saved file compatible with CANDE BASIC files: leading zeros are appended to the line numbers, if necessary, to make them five characters long. The resulting line is checked for a maximum of 80 characters. If a line exceeds 80 characters, the save is not done, and a message is emitted to identify the line (or lines) which cannot be made compatible.

If the current workfile does not have a name associated with it, either the AS option or the TITLE command must be used to associate a name with the workfile.

If the DEBUG WW initial parameter or the switch SW = W option is used (refer to appendix C), the current file takes the name of the AS file if present, as if a TITLE command had been used.

Examples:

SAVE	! Save the current workfile.
SAVE AS PQR ON P	! Save the current workfile on ! pack P with name PQR.
SAVE FOR CANDE	! Save the current workfile and ! attempt to make the file compatible with ! CANDE.

SCRATCH Command

The SCRATCH command causes a file to be removed or the current environment to be cleared, or causes both.

Syntax:

```
─────── SCRATCH ──┬──────────────────────────────┬───────────────────┤
                  └── <BASIC-file-name> ──┬──────────────────┬──┘
                                          └── ON <pack-name> ──┘
```
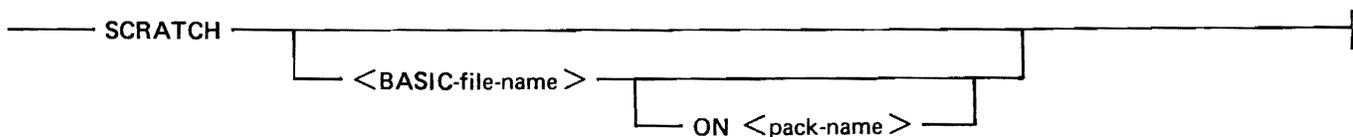
G18093

Semantics:

<BASIC-file-name> and <pack-name> are described under Syntax Definitions in this section.

The SCRATCH command is used to clear the current BASIC code and data environment or to remove a file in the current usercode scope of access from the disk directory. If no file name follows SCRATCH, the clearing of the current file in the BASIC environment is assumed. If the file name is specified, files can be removed from disk, from the default pack, or from an explicit pack if the ON option is used. The scratched file is irrecoverably removed.

Examples of SCRATCH commands:

SCR	! Clear the current workfile.
SCRATCH OLDFILE	! Remove OLDFILE from the ! user's directory.
SCR BADFILE ON P	! Remove BADFILE from the ! user's directory on pack P.

SEQUENCE Command

The SEQUENCE command initiates sequence mode on a screen type terminal. The syntax and semantics of the SEQUENCE command follow.

Syntax:
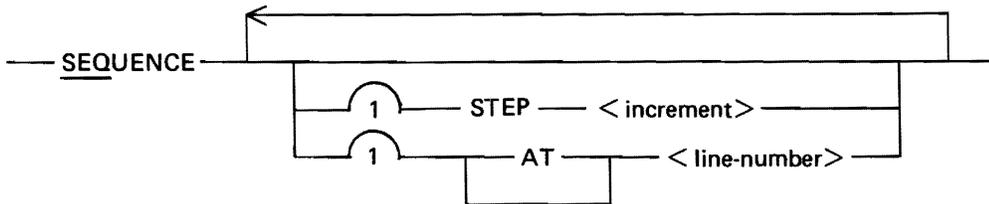
Semantics:

In sequence mode, line numbers are generated in sequence, starting at <line-number> and incrementing by <increment>, at the top of the screen ready for statement entry. If the generated line number corresponds to a line that already exists, the source line is displayed in a fashion similar to edit mode. If, after entry of a line, a line is found between the last entered line and the next line in sequence, that intermediate line is displayed as in edit mode.

Sequence mode is terminated if the generated line number exceeds 99999, if the pseudo break feature is used, or if any command is entered in the normal fashion.

If <line-number> is not specified, a value of 100 is assumed. If STEP <increment> is omitted, a value of 10 is assumed.

Examples of the SEQUENCE command:

    SEQ
    SEQ STEP 100
    SEQUENCE AT 1000
    SEQ 1000 STEP 100

STEP Command

The STEP command causes single stepping of a program. Single stepping is stopping after the execution of a statement.

Syntax:

G18094

Semantics:

The STEP command can be used during the execution of a program when the BASIC environment is stopped. It causes the next statement to be displayed and then executed. After this statement, an implicit BREAK statement is executed. Thus, the STEP command enables the statement-by-statement execution of a BASIC program at the user's discretion.

When a STEP command is entered, the BASIC data environment is not cleared, that is, data variables retain the values they had when the BASIC environment last stopped.

Certain editing functions, such as deletion of the next statement to be executed, changing the dimensions of an array, or changing a FOR NEXT statement cause the STEP command to be disallowed until after a RUN or WALK command is entered.

The STEP command has effect only when the BASIC environment is in a stopped state.

The STEP command may be simulated by using the SPCFY key. Refer to SPCFY Key Use in this section.

TEACH Command

The TEACH command lists the syntax of a command on the user's terminal.

Syntax:

```
————— TEACH ——┬─────────────────┬──————————————————————————————————┤
              └── <command> ──┘
```

G18095

<command> is any system or editing command contained in this section except pseudo BREAK.

Semantics:

The TEACH command causes a syntax diagram of the requested <command> to be displayed on the remote terminal. If no particular <command> is requested, a list of available commands is returned.

Examples:

    TEACH
    TE XREF
    TEACH CONT

TITLE Command

The TITLE command gives a name to the current workfile.

Syntax:

```
————— TITLE  <BASIC-file-name> ——————————————————————————————————————┤
```

G18096

<BASIC-file-name> is described under Syntax Definitions in this section.

Semantics:

The TITLE command changes the name of the BASIC environment. The new name is used as the file name for a SAVE operation, unless explicitly overridden by the SAVE AS syntax.

Example of a TITLE command:

>   TITLE   ! The current workfile is assigned
>   THIS   ! the name THIS.

USER Command

The USER command allows a user to log on.

Syntax:

```
——— USER <usercode> ———┬———————┬——┬———————————————┬—————————————————┤
                       └— / —┘  └— <password> —┘
G18097
```

Formation of <password> follows the same naming conventions as a <BASIC-file-name>.

Semantics:

The USER command is identical in function to the HELLO command, except that USER cannot be used for log off only. The intent of this command is to enable the SMCS auto log on feature, but its use is general.

Example of a USER command:

>   USER ME/ SECRET   ! Logs off the current usercode, if
>   ! one exists, and logs on ME/
>   ! SECRET.

WALK Command

The WALK command causes program execution to be initiated in a single-stepping fashion.

Syntax:

```
——— WALK ———┬———————————————————┬—————————————————————————————————┤
            └— <line-number> —┘
G18098
```

<line-number> is described under Statement Lines in Section 3.

Semantics:

The WALK command initiates the execution of the BASIC environment, clearing the BASIC data environment, and causes execution to be halted after the first statement is executed. The statement executed is displayed on the remote terminal. The STEP or CONTINUE command may be used to continue program execution after this command.

The presence of the optional <line-number> implies a GOTO <line-number> before execution begins. If <line-number> is invalid, a suitable message is returned.

Examples of WALK commands:

WALK   ! Run the first executable statement
                ! of the current workfile.

WALK 1234   ! Run line 1234 as the first executable
                ! statement of the current workfile.

WHAT Command

The WHAT command returns a statement that indicates the status of the current BASIC environment.

Syntax:

```
——— WHAT ———————————————————————————————————————————————|
G18099
```

Example of WHAT command output:

```
YOU ARE (SOONER) AT S8, LSN = 12 (TD832)
THE TIME IS 14:55:33.2 AND THE DATE IS 80 AUG 30
YOUR FILE IS CALLED "IBTEST" AND IS SAVED
AND BEGINS AT LINE 10 AND ENDS AT LINE 230
```

WHERE Command

The WHERE command returns information about the execution of a program.

Syntax:

```
——— WHERE ———┬————————————————————————————————————————|
             ├—— FROM ——┤
             └— CALLED —┘
G18100
```

Semantics:

The WHERE command, with no options, returns a message that contains the next statement to be executed.

If the FROM option is used, the last few (not more than 20) statements and commands executed are displayed.

If the CALLED option is used, the last few (not more then 20) GOSUB and user-defined FN<x> calls are displayed.

Examples of WHERE commands:

    WHERE
    WHERE FROM
    WHERE CALLED

Example of WHERE FROM command output:

    last 20 statements executed :-
    90 IF A = B THEN 150
    100 IF A < B THEN 130
    110 LET A = A - B
    120 GOTO 90
    90 IF A = B THEN 150
    100 IF A < B THEN 130
    110 LET A = A - B
    120 GOTO 90
    90 IF A = B THEN 150
    100 IF A < B THEN 130
    110 LET A = A - B
    120 GOTO 90
    90 IF A = B THEN 150
    100 IF A < B THEN 130
    110 LET A = A - B
    120 GOTO 90
    90 IF A = B THEN 150
    100 IF A < B THEN 130
    BREAK
    WHAT
    you are stopped - ready to continue at line 110

XREF or FIND Command

The XREF and FIND commands are equivalent. To avoid confusion, XREF is used to refer to the command throughout this manual. The syntax and semantics of the XREF command follow.

Syntax:

```
  ┌──XREF──┬┬─── <variable-name> ──────────────┬─────────────────────────────┤
  └──FIND──┘├─── <line-number> ────────────┤   └── <line-number-range> ──┘
            ├─── <line-number-range> ──────┤
            ├─── <user-defined-function> ──┤
            └── <delim>  <string>  <delim> ─┘
```
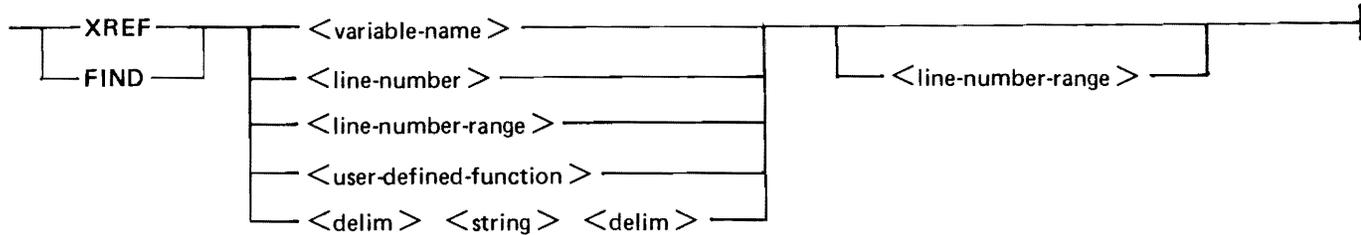
G18101

<variable-name> is any numeric or string variable. <line-number> is described under Statement Lines in Section 3. <line-number-range> is described under Syntax Definitions in this section. <user-defined-function> is any user-defined function as described in Section 8. <delim> is any non-alphabetic character. <string> is any string of characters, excluding <delim>.

Semantics:

The XREF command displays the BASIC statements that reference the requested item(s).

If a <line-number-range> is specified after a requested item, only that part of the BASIC program is searched.

A request for a <variable-name> (A-Z9,A$-Z9$) looks for both scalar and array references that use the given name.

A request for a <line-number> returns only the statements which reference that <line-number>.

A request for a <line-number-range> returns references to items within that range inclusively.

A request for a user FN name (FNA-FNZ9,FNA$-FNZ9$) returns the statements which call, define, or assign that name.

A request for a delimited string initiates a search through the source statement(s) for occurrences of that string as a strictly literal string.

Examples of XREF statements:

XREF A$	! Lists those statement lines ! which reference A$.
XREF 100	! Lists those statement lines ! which reference line 100.
XREF 100-200 2000-3000	! Lists those statement lines, ! between lines 2000 and 3000, ! which reference lines 100-200.
XREF FNX 123-654	! Lists those statement lines ! between 123 and 654 which ! reference FNX.
XREF "X"	! Lists those statement lines in ! which the character X ! occurs.

BASIC COMMANDS

A BASIC command is similar to a BASIC statement in function, except that a BASIC command is executed immediately, is required to be re-entered completely if the user desires to have it executed again, and is entered without a preceding line number.

Many of the BASIC statements described in Sections 3 through 10 of this manual can be used as BASIC commands but, by definition, some BASIC statements have no meaning unless properly accompanied by another statement or statements. A list of BASIC statements that are not allowed as commands follows.

DATA
DEF
DIM
FNEND
FOR
GOSUB
IMAGE or :
NEXT
ON GOSUB
OPTION
User-defined function references

BASIC commands which reference line numbers, for instance, GOTO and ON GOTO, simply change the next statement pointer (a memory location that contains the address of the next statement to be executed). Hence, if the BASIC environment is in a stopped state and a GOTO command is entered, a CONTINUE command causes execution to resume where the next statement pointer points, not necessarily where the environment was stopped. If the BASIC environment is running and a GOTO command is entered, execution continues as if the BASIC GOTO command were the next statement. Hence, the flow of execution of the running program may be changed dynamically.

If the statement following a THEN or ELSE in an IF statement is disallowed in Command mode, then the whole IF statement is disallowed.

Examples of BASIC commands:

```
PRINT A;B;C
BREAK
LET A = 1
PRINT A + B
MAT A = B - C
GOTO 500
```

STOP AND END COMMANDS

The STOP and END statements, when executed as commands, take on a special meaning. If a program is running or is stopped but able to be continued, the STOP or END command function as a STOP or END statement; that is, they terminate the execution of the current program such that continuation is prohibited. If a program is stopped, but not able to be continued (that is, a RUN or WALK statement must be used to cause a program to execute), then a STOP or END command terminates the current session by closing the printer backup file created by the PRINT option on the RUN, CONTINUE, and LIST commands. In this manner, a session begins at the first LIST PRINT, RUN PRINT, or CONTINUE PRINT and ends either when a STOP or END terminates it or when the user logs off.

The printer backup file created has the current usercode as its family name. The file name contains the date and time of the session in the form YYMMDDHHMM, where YY represents the last two digits of the current year, MM represents the month number, DD represents the day number, HH represents the hours using the 24-hour clock, and MM represents the minutes.

BASIC STATEMENT ENTRY

BASIC statements may be entered in any line number order. If a statement with a particular line number is entered more than once, the last entered line is retained, and all previously entered lines with that line number are lost.

The syntax of the statement is checked at the time of entry, and the line is retained even if there is a syntax error. Some BASIC statements rely on corresponding BASIC statements for complete correctness of syntax and function (for example, the FOR statement and the corresponding NEXT statement; the GOTO statement and the object of the GOTO). Any errors relating to this type of statement are detected and suitable error messages are emitted when any attempt is made to execute the program.

Thus, there are two kinds of syntax error message. One is emitted at statement entry time and the other is emitted when an attempt is made to run the program. These kinds of error messages are not mutually exclusive, hence, they may appear interspersed as a result of an attempt to run a program.

A BASIC statement may not exceed 256 characters in length.

Examples of BASIC statements:

```
10 PRINT A,B,C,D
5000 LET A$ = "IN THE " & A$
```

RECOVERY

The IBASIC system is able to recover the current user's BASIC source program if the central system or the IBASIC system fails. At user log on time (HELLO time or at BOJ of IBASIC if auto log on is requested or at sign on time if the SMCSAUTOLOG initial parameter is specified) the IBASIC program checks the system for the presence of a workfile left over from a previous session. If this file is present, IBASIC automatically reloads itself with the contents of this workfile. Only BASIC source code is maintained in this file, so the values of data variables from the previous session are lost.

If the recovery is not wanted, the workfile must be removed before log on, the recovered file must be scratched after log on, or a BREAK command must be entered during recovery. In order to remove the workfile before log on, it is necessary to determine the name of the workfile.

If IBASIC was using the MCP usercode system, the workfile name is <default-pack-name>/ (<usercode>)/ WORKFILE<xx>. The expression <xx> is a unique pair of characters generated from the usercode index in the MCP usercode file. Refer to the Recovery feature in the B 1000 Systems CANDE Reference Manual, form number 1090586.

If IBASIC was not using the MCP usercode system, the workfile name is called <usercode>/(WORKFILE).

SPCFY KEY USE (TD820 AND TD830 TERMINALS ONLY)

Depending on the state of the BASIC environment, the SPCFY key can be used as a shorthand way of typing a particular function.

The SPCFY key can be used in all cases where the pseudo BREAK feature can be used. Refer to Pseudo BREAK Feature in this section.

The SPCFY key can also be used as a shorthand notation for the STEP command or for the BASIC BREAK command, according to the state of the BASIC environment. If the BASIC environment is in a stopped state, ready to continue execution from wherever it was halted, a depression of the SPCFY key will have the same effect as entering the STEP command. If the BASIC environment is running, the effect is the same as when the BASIC BREAK command is entered: the program is halted and can be continued from that point.

To make sure that the SPCFY key does what is expected, it is suggested that the terminal be put in local mode by depressing the LOCAL key before depressing the SPCFY key.

SECTION 12
SPECIAL COMMANDS ('DOT' COMMANDS)

Dot commands are special commands which are primarily intended for debugging the IBASIC system. These commands do not go through the normal process of compilation and execution. Their syntax is simple.

General syntax:

G18102

BACKSPACE < new backspace char >

BACKSPACE changes the character to be used as a backspace character for TTY type terminals only. By default, this character is a reverse solidus (\ ).

CASE

CASE enables or disables the use of lower-case letters in system responses. By default, lower-case is enabled for TD820 and TD830 terminal types and disabled for TTY and TD800 terminal types.

CONTINUOUS

CONTINUOUS changes the setting of continuous or wait mode for consecutive output messages. CANDE does not recognize the change and assumes that the mode is unchanged. Continuous mode means that the terminal is not switched to local mode after receiving a message. Wait mode means that the terminal is switched.

DEBUG

DEBUG sets or resets a debug toggle which enables various compiler trace and dump functions.

DUMP

DUMP causes a dumpfile of IBASIC to be created.

FREEZE

FREEZE inhibits the MCP rollout process; thus, IBASIC is frozen in memory.

FREEZE

FREEZE inhibits the MCP rollout process; thus, IBASIC is frozen in memory.

HELLO

HELLO returns the opening message.

HINTS <STRING>

HINTS prints the contents of a memory area called HINTS, which is useful in the analysis of the IBA-SIC system. This memory area contains the values of several variables pertinent to the system. If <string> is present, it is included in the heading of the printout.

This command should be used if a problem is believed to exist. To initiate a dump to be sent to Burroughs for analysis, enter .DUMP HINTS.

LOCAL

LOCAL sets or resets a toggle which forces the remote terminal to be put in local mode after every command response.

LOG

LOG opens or closes a print file of all input and output messages. This command sets or resets a toggle accordingly.

OL

OL returns the data communication status of the remote station.

OVERLAY

OVERLAY returns the number of data and code overlays IBASIC has performed since BOJ.

PROMPT

PROMPT switches the form of the prompt for user input from a single "#" to the word "ready" as a prompt message and vice versa.

RY

RY, entered from the ODT, changes STATION(READY) to true.

SS <STRING>

SS displays <string> on either the remote terminal or the system console, the opposite of where the message originated. The RMSG system option must be set for system console messages to be displayed.

ST

ST changes STATION(READY) to false.

STATUSLINE

STATUSLINE switches on or off the maintenance of the TD830 status line.

NOTE
Firmware prior to the 2.0 release level in the TD830 does not implement the
STATUSLINE feature, so STATUSLINE must be switched off by entering
.STATUSLINE as the first message to IBASIC.

TIME

TIME returns the elapsed time since the current user logged on and the total amount of cpu time accumulated this session.

APPENDIX A
GLOSSARY OF IBASIC TERMS

active channel
　　A channel that has a file assigned to it.

alphanumeric
　　An alphabetic or a numeric character.

American Standard Code for Information Interchange (ASCII)
　　The standard code, consisting of 7-bit code characters, used for information interchange among data processing systems.

argument
　　An expression used in a function reference to communicate data between the calling program unit and the function.

arithmetic operator
　　A symbol used in a numeric expression to indicate the arithmetic operation to be performed by IBASIC.

array
　　A group of string or numeric values stored under an array name and organized in columns, or in rows and columns.

array element
　　One element of an array.

array name
　　A symbolic name for an array.

ASCII
　　Refer to American Standard Code for Information Interchange.

assign
　　To give a variable a value through use of a READ, INPUT, LINPUT, or assignment statement.

automatic log off
　　The log off action that takes place when a remote terminal is prematurely disconnected from IBASIC.

automatic log on
　　The function of automatically logging a user on to IBASIC without a specific log on action for the specified usercode.

BASIC
　　Beginner's All-Purpose Symbolic Instruction Code.

BASIC command
　　A BASIC statement used in Command mode.

BASIC environment
　　The set of BASIC code and data that are maintained in the workfile by IBASIC.

BASIC file name
    The name for an external file that can be specified within IBASIC.

BASIC statement (also BASIC language statement)
    A group of BASIC keywords and expressions associated with a single line number.

BASIC program
    A sequence of BASIC statements terminated by an END statement.

branch
    Transfer to another line in a program other than the next line in sequence.

break
    Temporary interruption in the execution of a program caused by the execution of a BREAK command or statement.

bridge
    A part of railroad syntax that specifies the number of times a path may or must be traversed.

channel number
    A numeric expression which evaluates to an integer in the range 0 to 255. The channel number specifies the channel through which a file is accessed.

character
    A letter, symbol, digit, or blank.

clause
    Part of some BASIC statements. A clause starts with a key word such as STEP, ELSE, USING, or THEN.

closed file
    A file that is not assigned a channel.

column
    The dimension of an array which represents the vertical arrangement of elements of that array.

columnar position
    The print position that is occupied by the next character transmitted to the current line; print positions are numbered consecutively from the left, starting with position one.

command
    Operating instruction to the system that is executed immediately when entered.

Command AND Edit (CANDE)
    An editor program on the B 1000 systems.

command mode
    The mode of interaction that is in effect when a command (no preceding line number) is entered.

constant
    A nonvariable numeric or string value.

continuous mode
    A type of message transmission that does not leave the terminal in local mode.

control statement
    A statement that can alter the sequence of execution for statement lines by causing the program to branch.

control variable
    A simple numeric variable used in a FOR NEXT loop to count and control the number of iterations of the loop.

conversational mode
    The "talking" mode in which IBASIC and a user interact.

CRT terminal
    Display screen terminal, most likely a cathode ray tube.

current line
    The string of characters (possibly zero) generated by PRINT and OUTPUT statements since the last end-of-line character was generated.

current program
    The program that is currently loaded into the BASIC environment.

data block
    List of constant values to be assigned to variables in a program through DATA and READ statements.

datum
    One item in a logical group of data.

debug
    To find and correct errors in a program.

default
    An attribute or value which is automatically selected by the system when not specified by the user.

default pack
    The pack associated with a specified usercode.

delimit
    To separate items of data with a delimiter.

delimiter
    A character that separates items of data.

digit
    A graphic character that represents an integer, for example, one of the characters 0 to 9.

dimension
    The size of an array.

dot command
    A special IBASIC command preceded by a dot.

dummy variable
    A variable used in the definition for a function that is defined in a program. When the function is used, the values listed as arguments are substituted for the dummy variables in the definition.

EBCDIC
    Refer to Extended Binary-Coded Decimal Interchange Code.

end of line
    The end of the record, or the first occurrence of a NUL, CR, LF, or ETX character.

end of record
    A NUL character or the physical end of the record.

enter
    To submit information to the IBASIC system for processing by pressing the transmit key (XMT key on a TD830).

entry mode
    The mode of interaction that is in effect when a BASIC statement is entered.

error
    A mistake in BASIC syntax, program logic, or system operation.

error number
    A number used by the system to identify an error.

execute
    To perform the operation or task indicated by a statement, program, or command. IBASIC executes a program by executing individual statements in a prescribed order.

explicit-point notation
    A method of representing a decimal number with decimal digits and a decimal point.

expression
    A constant, variable, function reference, or combination of these separated by operators and used to represent numbers or strings.

Extended Binary-Coded Decimal Interchange Code (EBCDIC)
    A character set, consisting of 8-bit coded characters, used for information interchange in data processing systems.

fatal error
    A run-time error which halts execution. An error message is displayed to inform the user of the error.

file name
    The name assigned to an external file.

file pointer
    An indicator of the position in a file that is affected by the next file input or output statement.

floating-point notation
    A method of representing a real number. For example, 0.0001234 is 0.1234E-3, where 0.1234 is the fractional part and E-3 is the exponent.

forms mode
    A format function, available on some remote terminals, which is used to send page information from the central system to the keyboard operator for the purpose of entering data in a particular format.

function
    An algorithm for making a calculation which yields a single value. Functions for some common calculations are provided by IBASIC. Other functions can be defined in programs with DEF statements.

function name
    A symbolic name used to identify a function.

global variable
    A variable which can be referenced from anywhere within a program.

IBASIC
    Refer to Interactive BASIC system.

image
    The format according to which one or more data items are to be printed.

implicit-point notation
    A method of representing a decimal number that contains only decimal digits. The decimal point is assumed to occur to the right of the rightmost digit of the number.

inactive channel
    A channel with no file assigned to it.

index
    A numeric expression which evaluates to an integer and identifies the position of an item of data with respect to some other item of data.

input
    Data supplied for processing through external media.

integer
    A whole number that can be represented exactly, using only decimal digits.

interaction
    The conversational dialogue that takes place between the user and the computer.

Interactive BASIC system
    The compiler, interpreter, message control system (MCS), editor, external intrinsics, and dummy program that comprise IBASIC.

interactive command
    An instruction to IBASIC.

intrinsic numeric function
    A predefined function supplied as part of IBASIC for the evaluation of commonly used numeric functions.

intrinsic string function
    A predefined function supplied as part of IBASIC for the evaluation of commonly used string-valued functions and numeric-valued functions whose arguments are strings.

jump
    Refer to branch.

justifier
    A greater than (>) or less than (<) sign, occurring in an image, which specifies right or left justi-
    fication, respectively.

keyword
    A character string which provides a distinctive identification of a statement or a component of
    a statement.

letter
    An English alphabet character: A through Z or a through z.

line number
    A number used to sequence a statement line. It may contain up to five decimal digits.

line number range
    A syntactic construction that allows a sequential group of line numbers to be specified.

local variable
    A variable that is only understood in a user-defined function. Parameters are the only variables
    that fall into this class.

loop
    A sequence of statements in a program that are executed repeatedly; a repeating path in a railroad
    syntax diagram.

margin
    The number of characters, excluding the end-of-line character, that can be written on one output
    line.

matrix
    A 2-dimensional array.

MCP file name
    A name for an external file that is valid to the MCP.

MCS
    Refer to message control system.

memory
    A place where the system can temporarily store programs and data during processing.

message control system (MCS)
    A program which opens a remote file with the HEADERS option and thereby controls the stations
    in that remote file.

NDL
    Refer to Network Definition Language.

nest
    To imbed a language structure within itself.

Network Controller
    The program generated through compilation of a Network Definition Language source program.
    The Network Controller handles the line discipline for the data communication devices of a system
    and the interface queue between an MCS and the operating system.

Network Definition Language
  A descriptive free-form language for defining and implementing a data communications network.
  The NDL compiler analyzes the input statements and generates a network controller.

next statement pointer
  A memory location that contains the address of the next statement to be executed.

nonfatal error
  A run-time error that does not halt execution of the BASIC program. An error message is displayed to inform the user of the error.

null string
  A string value containing no characters, represented in BASIC by " ".

numeric constant
  A series of decimal digits, occurring with a BASIC program, that denote a numeric value.

numeric expression
  A numeric constant, numeric variable, numeric function reference, or a combination of these separated by arithmetic operators.

numeric function reference
  An intrinsic numeric function or a user-defined numeric function.

numeric overflow
  A condition that occurs when a numeric value exceeds the maximum numeric value allowed.

numeric variable
  A symbolic name used to represent a numeric value which may be changed during program execution.

object of a loop
  An item within a loop of a railroad syntax diagram.

open file
  A file that is assigned to a channel.

operand
  A numeric or string expression used as part of a larger numeric or string expression.

operator
  The symbol used in a numeric, string, or relational expression to indicate the operation to be performed by IBASIC in order to find the value of the expression.

optional item
  An item in a railroad syntax diagram that may be omitted.

order of operations
  The standard sequence in which IBASIC performs operations to find the values of expressions.

ordinal position
  The position of a character in either of the character sets used in BASIC (ASCII or EBCDIC).

output
  The results of a program, written to external media.

overflow
Refer to numeric overflow and/or string overflow.

parameter
A simple variable used in a function to pass data between the calling routine and the function.

password
A word associated with a usercode that allows access to IBASIC.

path
The sequence of execution for statements in a program.

print item
An expression or a TAB call occurring in a PRINT or OUTPUT statement.

print line
A transmission of characters, from PRINT and/or OUTPUT statements, which terminates with an end-of-line character.

print zone
A contiguous set of 15 character positions in an output line which may contain an evaluated PRINT or OUTPUT statement expression.

privileged status
The status that the IBASIC system has if it is executed under a privileged MCP-type usercode or a non-MCP usercode. Privileged status affects the types of files that may be accessed.

program
A sequence of instructions for doing a task on a computer.

program designator
A string expression whose value specifies the name of a program to which chaining is to be performed.

prompt
A message displayed to signal the user to enter input.

quoted string character
Any character in Table E-1 in Appendix E, except those characters in ordinal positions 0 through 31, 34, 64, 91, 92, 96, and 123 through 127. Ordinal position 34, the quotation mark ("), may appear as a quoted string character if it is represented by two adjacent quotation marks.

real number
Decimal number containing a decimal point.

record length
The number of characters between the beginning of a record and the end of the record.

redimension
To change the bounds of an existing array.

relational expression
An expression containing a relational operator and having the value of true or false. Relational expressions are used only in IF statements to cause the program to take one path if the expression is true and another path if the expression is false.

relational operator
    Symbol used in an expression to define a comparison to be made between two numbers or strings.

remark string
    A string of characters occurring in either a tail comment or a REM statement.

required item
    An item in a railroad syntax diagram that may not be omitted.

reverse video
    A method of highlighting a line on a display screen terminal.

row
    The dimension of an array which represents the horizontal arrangement of elements of that array.

run-time
    Occurring during program execution.

scalar
    A quantity characterized by a single numeric or string value.

scaled notation
    A method of representing a number by using a real number raised to a power of 10.

scope of file access
    A particular BASIC environment may access, for input only, any disk file maintained by MCPII which may be accessed under the currently logged on usercode. The BASIC environment may only create or change a file which has the family name of the currently logged on usercode. Thus, the rules for forming a file name, which is for input only, are more flexible than those for files which are to be created.

sign
    A plus (+) or minus (–) sign.

simple variable
    A variable that is not subscripted.

single stepping
    A method of executing a BASIC program in which each BASIC statement is performed in response to a single manual operation.

SMCS
    Refer to Supervisory Message Control System.

source code
    BASIC language statements.

source line
    A line of source code.

statement
    An instruction in a BASIC program occurring on one statement line.

statement line
    A line number followed by a BASIC statement.

string
A series of consecutive characters treated as a group.

string constant
A string of characters enclosed within quotation marks (").

string expression
A string constant, string variable, string function reference, or a concatenation of these.

string function reference
An intrinsic string function or a user-defined string function.

string length
The number of characters represented by a string.

string overflow
A condition occurring when a string variable is assigned more characters than its length allows.

string variable
A symbolic name used to represent a string value which may be changed during program execution.

subscript
An index into a row or a column of an array. A 1-dimensional array has one subscript and a 2-dimensional array has two subscripts.

subscripted variable
A variable with one or two subscripts.

Supervisory Message Control System (SMCS)
The standard message control system available on the B 1000 systems.

symbolic name
A symbol or symbols used to represent a numeric or string variable.

syntax error
An error in the syntax of a command or statement.

system command
See command.

underflow
A condition that occurs when an attempt is made to represent a numeric value smaller than the smallest value representable in BASIC.

unscaled notation
Notation for a number characterized by the absence of an exponent part as occurs in scaled notation.

usercode
A name assigned to a user and used for file security.

user-defined function
A function defined by a user with a DEF statement.

user session
   The period of time during which a user is logged on.

value
   A number or a string represented by a constant, variable, or expression.

variable
   A data name within a program whose value can be changed.

vector
   A 1-dimensional array.

wait mode
   A type of message transmission that leaves the terminal in local mode.

workfile
   The temporary file that IBASIC uses to store user-entered BASIC statements.

zoned format
   Design for output that allocates 15 character positions for each value.

APPENDIX B
IBASIC LOG ON, LOG OFF, AND EXECUTION

EXECUTION UNDER SMCS

IBASIC can be initiated by using either the EXECUTE program control instruction or by using the SIGN ON command of SMCS.

The signal character for IBASIC cannot be the pound sign character (#), the period character (.), or SUB (refer to table E-1 in appendix E).

The current IBASIC session can be terminated by using the SIGN OFF command of SMCS. However, if the SIGN OFF command is used during an IBASIC session initiated with the NO EOJONLOGOUT parameter (refer to Initial Parameters in appendix C), the IBASIC program does not go to EOJ. In this case, the SMCS program allows the same station to initiate another IBASIC session using the SIGN ON command without causing another BOJ. If the SIGN OFF CLEAR command is used to terminate an IBASIC session initiated with the NO EOJONLOGOUT parameter, the IBASIC program remains available to any station using the SIGN ON command without causing another BOJ.

Termination of an IBASIC job occurs if a message is sent to the IBASIC program by way of the SMCS PASS mechanism.

USING THE EXECUTE PROGRAM CONTROL INSTRUCTION

The IBASIC program can be initiated using the EXECUTE program control instruction. In this case, no entry is required in the SMCS jobs file. An accept (AX) message can be used at the beginning of the IBASIC program execution to set the initial parameters for the the IBASIC environment. This will be referred to as an early accept message in the following sections. These parameters are described under Initial Parameters in appendix C.

If the terminal is logged on under an MCP usercode, the IBASIC program is executed under that usercode and the normal usercode considerations apply. If automatic log-on is required and the logged on usercode is privileged, the AUTOLOG parameter must be specified in the early accept message.

USING THE SIGN ON COMMAND

IBASIC can be initiated using the SIGN ON command of SMCS (for example, ON IBASIC). This requires one of two possible entries in the SMCS jobs file: (1) one which causes an EOJ when the user signs off, and (2) one which does not cause an EOJ when the user signs off.

The following entry in the SMCS jobs file causes an automatic log-on and causes an EOJ when the user signs off.

```
column 1  $
column 2  IBASIC LOG-ON NO-SESSION ;
          EX IBASIC
          ME <user site default>
          PR <user site default>
          AX SMCSAUTOLOG <plus any other parameters
              listed under Initial Parameters in
              Appendix C>
```

The following entry in the SMCS jobs file causes an automatic log-on and does not cause the IBASIC program to go to EOJ when the user signs off. The user is logged off and the user station is detached from the IBASIC program. The IBASIC program waits in the mix for another user to sign on. If a user is not logged on, the IBASIC program goes to EOJ upon receiving a BYE or SIGN OFF command.

```
column 1    $
column 2    IBASIC NO-EOF <optional wait limit>
                LOG-ON EXCEPT NO-SESSION;
            EXECUTE IBASIC;
            MEMORY <user site default>;
            PRIORITY <user site default>;
            AX SMCSAUTOLOG NO EOJONLOGOUT <plus any
                other parameters listed under Initial
                Parameters in appendix C>
```

If a limit is desired on the time the IBASIC program waits in the mix with no user, a wait limit can be placed in <optional wait limit>. This field has the following form.

<OPTIONAL WAIT LIMIT> SYNTAX:
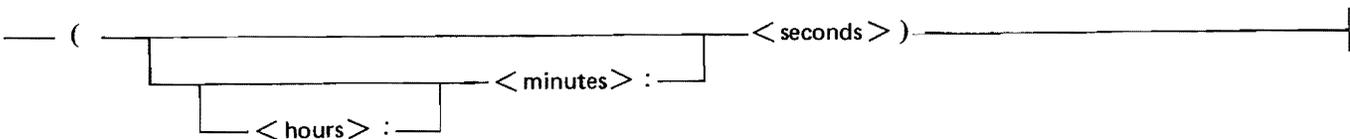
<hours>, <minutes>, and <seconds> are integers. For example, (2:30) specifies a limit of 150 seconds.

EXECUTION UNDER CANDE

The IBASIC program can be initiated through CANDE under the usercode which is logged on to CANDE by using the EXECUTE command. If automatic log-on to this usercode is required, the early accept message must include the AUTOLOG parameter (refer to Initial Parameters in appendix C).

Example:

    EXECUTE *IBASIC;<optional ME parameter>; AX AUTOLOG

                        NOTE
    If IBASIC does a display to the ODT (for instance, as a result of a .SS com-
    mand), the display messages are repeated at the remote terminal and are dis-
    played only at the ODT if the RMSG system option is set.

EXECUTION WITH NO MCS

If no MCS is present, the IBASIC program must be executed from the ODT with the initial parameter of ODT_MODE included in the early accept message (refer to Initial Parameters in appendix C). If automatic log-on to the usercode under which the IBASIC program is executed is desired, the early accept message must also include the AUTOLOG parameter. If other than the default memory is required, a memory clause must be added to the control string.

Example:

USER ME/MINE EXECUTE IBASIC; MEMORY 400000; AX ODT__MODE AUTOLOG

AUTOMATIC LOG-OFF

If the remote terminal is prematurely disconnected from IBASIC before a proper log-off (BYE or HELLO command) occurs, an automatic log-off occurs regardless of the state of the BASIC environment. The remote user must re-establish connection and log on to IBASIC again in the normal manner. The source file that was loaded at the time of the log-off can be recovered, but any data values are lost.

The TERMINATE ERROR mechanism in the network controller implements this feature. Thus, if IBASIC receives a TERMINATE ERROR message from the remote terminal with one or more of the relevant error conditions true, a log-off procedure is initiated. The following data communication errors are relevant:

TIMEOUT
LOSS OF DSR
LOSS OF CARRIER
ADDRESS ERROR
TRANSLATE ERROR
FORMAT ERROR
READ NOT READY

The same procedure is invoked if the SMCS sign OFF command is used before proper log off procedures occur.

It is strongly recommended that the remote user log off IBASIC in the proper manner if conditions allow.

RUN-TIME LIMIT

The amount of Central Processing Unit (CPU) time that a user session can consume without interruption can be specified. If no limit is specified (the default situation), a user session can continue indefinitely. If a limit is specified, a program is temporarily suspended if that limit is exceeded during the session. Program execution can be continued in the normal fashion after this interruption until the limit is reached again. This feature provides a way to stop a program from consuming vast amounts of CPU time if the program is in a loop and is running unattended.

The RUNLIMIT parameter of the early accept message specifies the run-time limit in minutes (refer to Initial Parameters in appendix C). For example, if the IBASIC program is executed with an initial parameter of RUNLIMIT 2, then every time two minutes of CPU time is used for program execution, the user session is interrupted and the user must do something positive, for example, enter a CONTINUE command to continue the session. If the RUNLIMIT parameter has a value of zero, the user session is unlimited.

APPENDIX C
OPERATIONAL CONSIDERATIONS

NETWORK CONTROLLER CONSIDERATIONS

There are some constraints on the generation of the Network Controller. The Interactive BASIC system assumes the validity of the TYPE field specified in the TERMINAL Section of the NDL source. Currently, only the following values for the TYPE field are valid:

Type	Terminal
0	B9350 (TTY)
26	TC4000
41	TD801
42	TD802
43	TD821
44	TD822
45	TD831
46	TD832

Scrolling of input and output lines is supported only for TD820 and TD830 type terminals. To enable scrolling, the standard CANDE request and control sets must be used.

For correct operation of TTY type terminals, the CANDE IOTTY request set must be used.

The following considerations are only relevant if nonstandard request and control sets are used. A knowledge of NDL coding and the use of station TOG and TALLY values is assumed. The special meanings for the following values are assumed.

When TOG[1] is set on an output message, it means that this message is not to be scrolled.

A true value in station TOG[2] in an input message means that the input message was scrolled or that the output message is to be scrolled.

Station TOG[3] is set for all output messages to a teletype, and associated TOG[7] can be set to indicate inhibition of transmission of trailing CR and LF characters for this message.

Station TOG[3] is also set for all messages that are to be scrolled. This method is used to enable multiple line output scrolling which is in the CANDE request sets.

A true value in station TOG[5] means that BREAK was detected in the last output attempt. This must only be true in a 'GOOD RESULTS REPLY' type message.

If station TALLY[0] = 2, a screen type terminal is forced to local after this output message (even in scroll mode).

INTERACTIVE BASIC SYSTEM CONSIDERATIONS

Modifications to the IBASIC environment can be made to suit the individual user. These modifications may be applied through MODIFY MCP syntax or as parameters with the EXECUTE program control instruction.

There is one copy of the IBASIC program for each user of the system. Thus, the following considerations may be applied differently to each user, or globally to all users as required.

INITIAL PARAMETERS

An early accept message can be used to supply a set of initial parameters which help determine the IBASIC environment. The early accept message is specified with the EXECUTE program control instruction. All of the desired parameters must be supplied in a single accept message.

Syntax:

```
EXECUTE IBASIC; AX ────┬──────────────┬──── AUTOLOG ──────────┐
                       └── NO ──┘      ├── CONTINUOUS ──────┤
                                       ├── EOJONLOGOUT ─────┤
                                       ├── LOWERCASE ───────┤
                                       ├── ODT ─────────────┤
                                       ├── RUNNER ──────────┤
                                       ├── SMCSAUTOLOG ─────┤
                                       └── USERCODES ───────┘
                       ├── BACKSPACE  < character > ────────┤
                       ├── DEBUGTYPE ──┬── DEBUG ──────────┐
                       │               ├── LOG ────────────┤
                       │               ├── WW ─────────────┤
                       │               └── QC ─────────────┘
                       ├── RUNLIMIT  < integer > ───────────┤
                       └── WORKFILEAREAS  < integer > ──────┘
```

Restrictions:

The SMCSAUTOLOG and AUTOLOG parameters cannot both be set.

The SMCSAUTOLOG and ODT parameters cannot both be set.

The NO EOJONLOGOUT parameter requires that the SMCSAUTOLOG parameter be set.

Table C-1 contains descriptions of the parameters and arguments shown in the early accept message of the EXECUTE program control instruction.

Table C-1. Early Accept Parameters and Arguments

Parameter	Valid Arguments	Default Setting	Description
AUTOLOG	none	FALSE	Refer to Execution Under SMCS in appendix B.
BACKSPACE	< character >	" \ "	Changes the character used as a backspace character for TTY-type terminals.
CONTINUOUS	none	FALSE	Causes page pauses on terminal output.
DEBUGTYPE	DEBUG	none	Sets a debug option within the compiler.
	LOG	none	Initiates printer log of the input and output operations of a remote user.
	WW	none	Same effect as SWITCH = W
	QC	none	Sends all terminal output messages to the remote file that approved the open for IBASIC. The message type is set to 50 to indicate that the message is from IBASIC and not the remote station.
EOJONLOGOUT	none	TRUE	Refer to Execution Under SMCS in appendix B.
LOWERCASE	none	TRUE	Specifies the type case output.
NO	none	none	The NO keyword resets the parameter that immediately follows it.
ODT	none	FALSE	Refer to ODT Operation in this appendix.
RUNLIMIT	< integer >	0	CPU run-time limit in minutes. Zero means no limit.
RUNNER	none	TRUE	Enables calls to the microinterpreter.
SMCSAUTOLOG	none	FALSE	Refer to Execution Under SMCS in appendix B.
USERCODES	none	TRUE	Indicates whether or not operation is under MCP usercode/password security
WORKFILEAREAS	< integer >	5	Number of areas in the source workfile. Each area can contain 400 statements.

DYNAMIC MEMORY

IBASIC relies heavily on SDL paged array structures. Both s-code and data are maintained in paged arrays; therefore, if the average BASIC program has many statements or uses large amounts of data, IBASIC may require a larger dynamic memory size. Dynamic memory size can be changed only at BOJ time of IBASIC, so either an ME control parameter can be included in the execute command or a particular installation can modify a default value to suit its particular median requirements. Use of the .OVERLAY dot command may help in determining the need for more memory.

HARDWARE REQUIREMENTS

The Interactive BASIC system requires the following minimum hardware:

    B 1000 processor
    128K bytes (dependent on number of users)
    1MB disk per user
    TD820, TD830, TC4000, TTY type terminals

ODT OPERATION

If no remote operation is required (that is, if no data communication system exists or is needed), IBASIC can be operated from the system console. To do this, the IBASIC program must be initiated with the EXECUTE program control instruction and must be given an early accept message specifying the ODT parameter (refer to Initial Parameters in this appendix). The ensuing log-on process is identical to the normal remote log-on process according to the usercode (if any) under which the IBASIC program is logged on. There is no implied limit to the number of users who can use the ODT. Further communication with the IBASIC program is achieved through normal accept messages.

PRIORITY

For quick servicing of a request from a user station, the IBASIC program must be run at a priority (both memory and processor) higher than any batch jobs, as is the case for most remote applications.

SOFTWARE REQUIREMENTS

The Interactive BASIC system requires the following software.

    IBASIC   (compiler)
    IBASIC/INTERP   (interpreter)
    IBASIC/INTRINSICS   (intrinsics)

    MCPII   (Systems software release 9.0 or later)
    NDL   (Systems software release 9.0 or later)

SWITCH VALUES

The following switch value is allowed for debugging purposes.

SW = W
Enables extensions to the ANSI language as explained in appendix F, and forces auto log-on.

USERCODE CONSIDERATIONS

The user has a choice of whether or not to run the IBASIC program under the B 1000 file security system.

If IBASIC is not executed under the B 1000 file security system, the scope of access of the BASIC environment is limited to public files. In this case, the NO USERCODES parameter must be specified in the early accept message (refer to Initial Parameters in this appendix). In this mode, the concept of a password does not exist: a logged on usercode is simply a default family name of a file accessed for input only; and is also a mandatory family name for a file to be created or changed.

If the IBASIC program is executed under a nonprivileged MCP usercode, it automatically logs on that usercode. Log-on of another usercode is disallowed. The IBASIC program must be logged off and re-executed under a different usercode to effect a change of user.

If the IBASIC program is executed under a privileged MCP usercode, automatic log-on to the execution usercode depends on the value of the AUTOLOG parameter. If automatic log-on is not requested, the IBASIC program requests an explicit log-on from the user before continuing. The usercode and password must be defined in the usercode file. Regardless of the value of the AUTOLOG parameter, another user can log on after the current user has logged off without the IBASIC program going to end of job (EOJ).

WORKFILE CONSIDERATIONS

The IBASIC workfile allows up to 1979 BASIC statements in a program by default. This limit (and the disk space required for the IBASIC user) can be modified by using the WORKFILEAREAS parameter in an early accept message (refer to Initial Parameters in this appendix). The default value of this parameter is 5 areas and each area can contain 400 statements. The value specified by the WORKFILEAREAS parameter can be used to increase or decrease the default value. The parameter only affects the creation of a new workfile. If a recovery workfile is used, the maximum size is determined by the size of the recovery workfile alone.

APPENDIX D
SYNTAX SUMMARY

ABS FUNCTION

```
———— ABS ( <numeric-expression> ) ———————————————————————————————————————|
G18103
```

ACOS FUNCTION

```
———— ACOS ( <numeric-expression> ) ——————————————————————————————————————|
G18104
```

ANGLE FUNCTION

```
———— ANGLE ( <numeric-expression> , <numeric-expression> ) ——————————————|
G18105
```

ASIN FUNCTION

```
———— ASIN ( <numeric-expression> ) ——————————————————————————————————————|
G18106
```

ATN FUNCTION

```
———— ATN ( <numeric-expression> ) ———————————————————————————————————————|
G18107
```

.BACKSPACE COMMAND

```
———— . BACKSPACE <new backspace char> ———————————————————————————————————|
G18108
```

BASIC FILE NAME


```
G18074
```
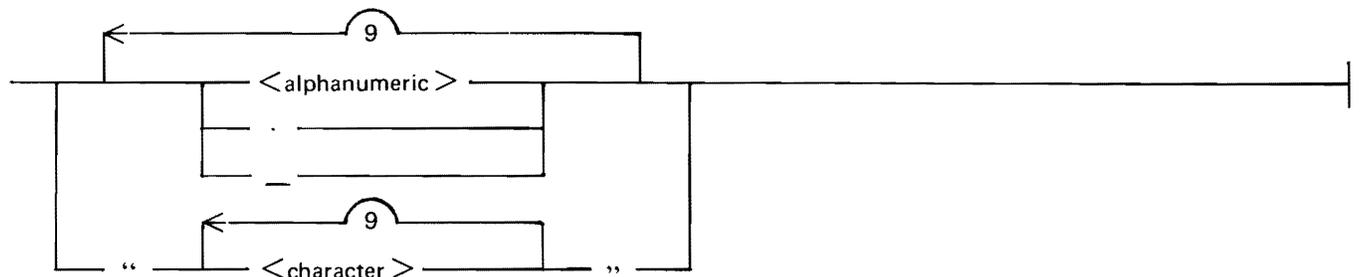
BREAK STATEMENT

──── BREAK ──┤

G18071

.BRK COMMAND, BRK

──── .BRK ───┤

G18088

BYE STATEMENT

──── BYE ──┤

G18077

.CASE COMMAND

──── .CASE ──┤

G18109

CEIL FUNCTION

──── CEIL (<numeric-expression>) ────────────────────────┤

G18110

CHAIN STATEMENT

──── CHAIN <program-designator> ───────────────────────────┤

G18044

CHR$ FUNCTION

──── CHR$ (<numeric-expression>) ────────────────────────┤

G18111

CLOSE STATEMENT

──── CLOSE # <channel-number> ─────────────────────────────┤

G18063

CONTINUE COMMAND

──── CONTINUE ──┬─────────────────────┬────────────────────┤
                │   ┌── 1 ── TRACE ──┐ │
                └───┤   ├─┘
                    └── 1 ── PRINT ──┘

G18078

D-2

.CONTINUOUS COMMAND

```
———— .CONTINUOUS ————————————————————————————————————————|
G18112
```

COS FUNCTION

```
———— COS ( <numeric-expression> ) —————————————————————————————|
G18113
```

COSH FUNCTION

```
———— COSH ( <numeric-expression> ) ————————————————————————————|
G18114
```

COT FUNCTION

```
———— COT ( <numeric-expression> ) —————————————————————————————|
G18115
```

CSC FUNCTION

```
———— CSC ( <numeric-expression> ) —————————————————————————————|
G18116
```

DATA STATEMENT

```
                    ┌←——— , ———┐
———— DATA ————┌———— <datum> ———┘—————————————————————————————|
G18046
```

DATE FUNCTION

```
———— DATE ———————————————————————————————————————————————|
G18117
```

DATE$ FUNCTION

```
———— DATE$ ——————————————————————————————————————————————|
G18118
```

DEBUG STATEMENT

```
———— DEBUG ——┬—— ON ——┬———————————————————————————————|
             └— OFF ——┘
G18070
```

.DEBUG COMMAND

```
———— .DEBUG ————————————————————————————————————————————————|
G18119
```

DEF STATEMENT

```
                                        ┌──── 6 ──── , ──┐
————— DEF ——FN <letter> ┬─<digit>─┬┬─$─┬┬─ ( ─┴─ <parameter> ─┴─ ) ─┬┬─ = <expression> ─┬|
                        └─────────┘└───┘└──────────────────────────┘└───────────────────┘
G18120
```

DEG FUNCTION

```
————— DEG ( <numeric-expression> ) —————————————————————————————————|
G18121
```

DELETE COMMAND

```
——┬── DELETE <line-number-range> ──┬─————————————————————————————————|
  └── <line-number> ───────────────┘
G18079
```

DET FUNCTION

```
—— DET ──┬── ( — <array-name> — ) ──┬—|
         └──────────────────────────┘
```

DIM STATEMENT

```
              ┌────────────────── , ──────────────────────────┐
————— DIM ──┬─┴─ <array-name> ( <row> ─┬── , <column> ─┬─ ) ──┴─————————|
            │                          └───────────────┘
            └─ <string-variable> * <integer> ──┘
G18122
```

DO STATEMENT

```
DO ─┬──————————————————————————————————————————————————————————————|
    ├── WHILE ──┬── <relational-expression> ─┘
    └── UNTIL ──┘
```

DOT FUNCTION

```
——— DOT ( < array-name > · < array-name > ) ————————————————————————————————|
G18024
```

.DUMP COMMAND

```
——— .DUMP ————————————————————————————————————————————————————————————|
G18123
```

EDIT COMMAND

```
— EDIT ——┬—————————————┬——┬—————————————< line-number > ——┬—|
         └— ERRORS —┘    ├——————————————┘
                         └— FROM —┘
```

END STATEMENT

```
——— END ————————————————————————————————————————————————————————————|
G18004
```

EPS FUNCTION

```
——— EPS ————————————————————————————————————————————————————————————|
G18124
```

EXCEPTION STATEMENT

```
——— AT EOF # < channel-number > THEN < line-number > ————————————————|
G18067
```

EXIT STATEMENT

```
EXIT IF < relational-expression > ——————————————————————————————————|
```

EXP FUNCTION

```
——— EXP ( < numeric-expression > ) ——————————————————————————————————|
G18125
```

FILE COMMAND

```
——— FILE ——┬—< MCP-file-name >——┬——┬— ON < pack-name > ——┬—|
           ├—< file-name > / = —┤  └———————————————————————┘
           └— = —————————————————┘
G18080
```

FIX COMMAND

```
——— FIX ┬──────────────────┬── <delim> <text> <delim> <new-text> ┬──────────────────┬──┤
         └── <line-number-range> ──┘                                └── <delim> ──┘
```

G18081

FNEND STATEMENT

```
──┬── FNEND ──┬────────────────────────────────────────────────────────────────┤
  └── END DEF ──┘
```

G18042

FOR STATEMENT

```
——— FOR <control-var> = <initial-value> TO <limit> ─┬── STEP <increment> ──┬──────┤
                                                     └──────────────────────┘
```

G18037

FP FUNCTION

```
——— FP ( <numeric-expression> ) ─────────────────────────────────────────────┤
```

G18126

.FREEZE COMMAND

```
——— .FREEZE ──────────────────────────────────────────────────────────────────┤
```

G18127

GET COMMAND

```
——— GET <MCP-file-name> ─┬──────────────────────┬──────────────────────────────┤
                          └── ON <pack-name> ──┘
```

G18082

GOSUB STATEMENT

```
——— GOSUB <line-number> ──────────────────────────────────────────────────────┤
```

G18033

GOTO STATEMENT

```
——— GOTO <line-number> ───────────────────────────────────────────────────────┤
```

G18032

HELLO COMMAND

```
─────── HELLO ─────┬─────────────────────────────────────────┬──────────┤
                   └── <usercode> ─┬───────────┬─┬──────────────┬─┘
                                   └──── / ────┘ └── <password> ─┘
```
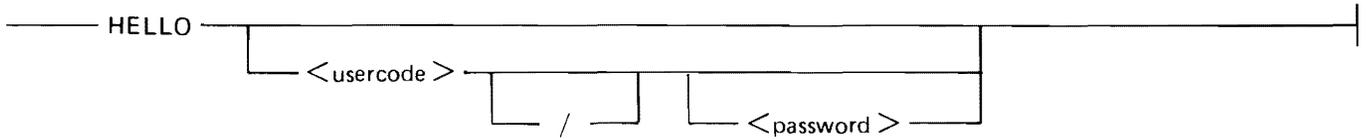
G18083

.HELLO COMMAND

```
─────── .HELLO ──────────────────────────────────────────────────┤
```

G18128

.HINTS COMMAND

```
─────── .HINTS <string> ──────────────────────────────────────────┤
```

G18129

IF STATEMENT

```
─────── IF <rel-exp> THEN ──┬── <line-no1> ──┬─── ELSE ──┬── <line-no2> ──┬──────┤
                            └── <stmt1> ──────┘           └── < stmt2 > ───┘
```

G18039

IF STRUCTURE (BLOCK)

IMAGE STATEMENT

G18055

INF FUNCTION

```
——————— INF ——————————————————————————————————————|
```
G18130

INPUT REPLY

```
——————— INPUT ——┬——PROMPT  <string-expression> : ——┬——<variable>——┬——————|
                 │                                   │              └—— , ——┘
                 └—— # <channel-number> : —————————————┘
```
G18131

INPUT STATEMENT

```
——————— INPUT ——┬——PROMPT  <string-expression> : ——┬——<variable>——┬——————|
                 │                                   │              └—— , ——┘
                 └—— # <channel-number> : —————————————┘
```
G18131

INQUIRE STATEMENT

```
—— INQUIRE —— # —— <channel-number> —— : —→

  > ——┬—— 1 —— ACCESS ————————┬—— <string-variable> ——|
      ├—— 1 —— FILE TYPE ——————┤
      ├—— 1 —— NAME ———————————┤
      ├—— 1 —— ORGANIZATION ———┤
      ├—— 1 —— RECORD LENGTH ——┤
      ├—— 1 —— RECORD TYPE ————┤
      └—— 1 —— STATUS —————————┘
```

INT FUNCTION

```
——————— INT  ( <numeric-expression> ) ————————————————————————|
```
G18132

IP FUNCTION

```
——————— IP  ( <numeric-expression> ) —————————————————————————|
```
G18133
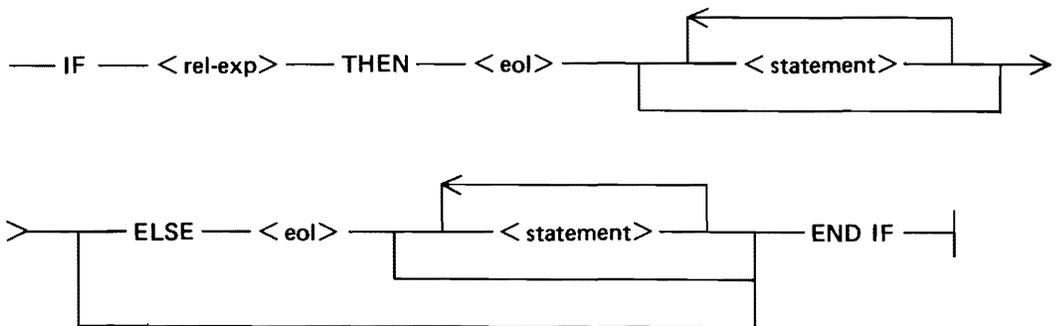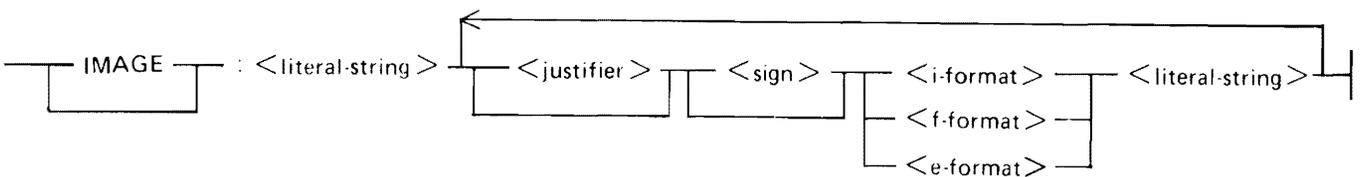
LDIM FUNCTION

——————— LDIM (<array-name> , <numeric-expression>) ———————————————————————

G18134

LEN FUNCTION

——————— LEN (<string-expression>) ———————————————————————————————————

G18135

LINE NUMBER
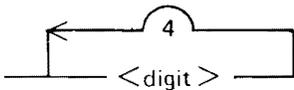
G18136

LINE NUMBER RANGE
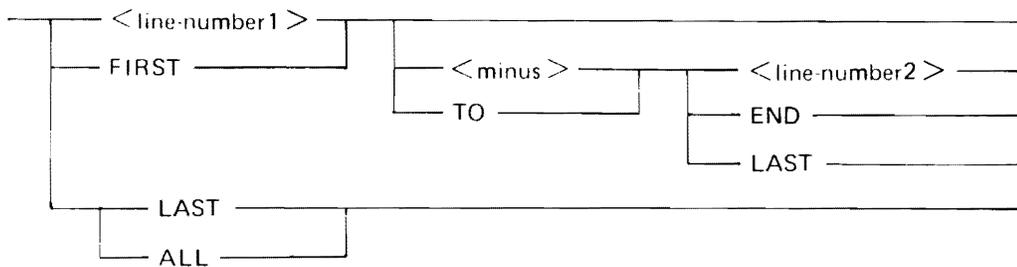
G18073

LINPUT REPLY
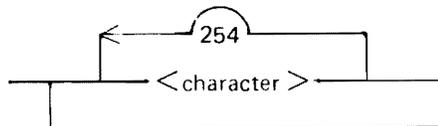
G18052

LINPUT STATEMENT
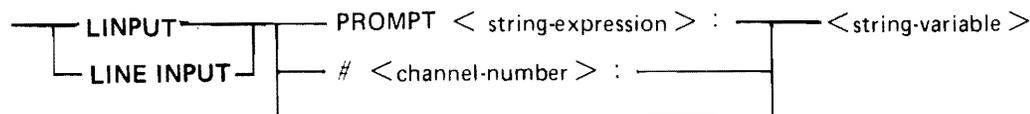
G18051

LIST COMMAND
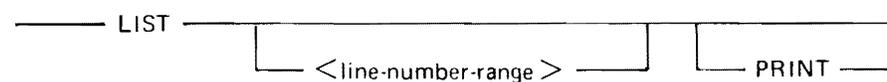
G18084

.LOCAL COMMAND

```
──────── . LOCAL ────────────────────────────────────────────────────────
G18138
```

LOG FUNCTION

```
──────── LOG ( <numeric-expression> ) ───────────────────────────────────
G18139
```

.LOG COMMAND

```
──────── . LOG ───────────────────────────────────────────────────────────
G18140
```

LOG10 FUNCTION

```
──────── LOG10 ( <numeric-expression> ) ─────────────────────────────────
G18141
```

LOG2 FUNCTION

```
──────── LOG2 ( <numeric-expression> ) ──────────────────────────────────
G18142
```

LOOP STATEMENT

```
LOOP ─────────────────────────────────────────────────────────────────────
         ├── WHILE ─────┬── <relational-expression> ──┤
         └── UNTIL ─────┘
```

MAKE COMMAND

```
──────── MAKE <BASIC-file-name> ─────────────────────────────────────────
G18085
```

MARGIN STATEMENT

```
──────── MARGIN <margin-value> ──────────────────────────────────────────
G18059
```

MAT ADDITION STATEMENT

```
──────── MAT < array-name1 > = < array-name2 > + < array-name3 > ────────
G18021
```
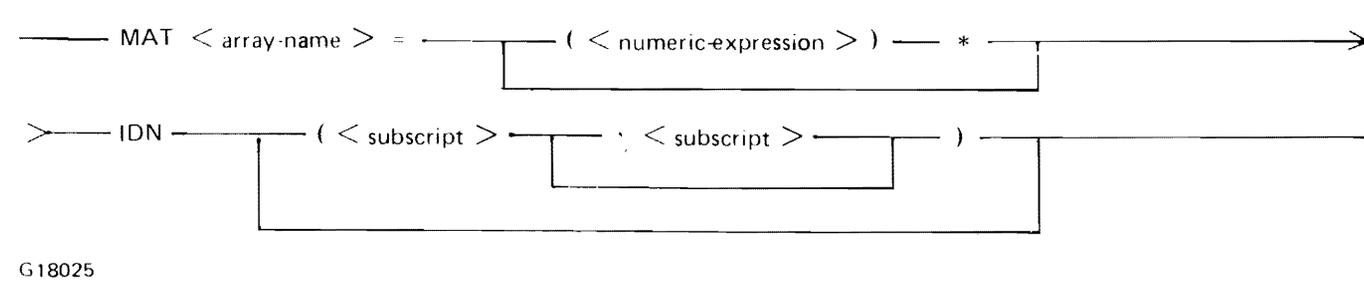
D-10

MAT ASSIGNMENT STATEMENT

————— MAT < array-name1 > = < array-name2 > ————————————————————————

G18022

MAT CON STATEMENT

————— MAT < array-name > = ——┬—— (< numeric-expression >) —— * —————————————→

>—— CON ——┬—— (< subscript > ——┬—— , < subscript > ——┬——) ——————————————

G18023

MAT IDN STATEMENT

————— MAT < array-name > = ——┬—— (< numeric-expression >) —— * —————————————→

>—— IDN ——┬—— (< subscript > ——┬—— , < subscript > ——┬——) ——————————————

G18025

MAT INPUT STATEMENT

—— MAT INPUT ——┬—— PROMPT < string-expression > : ——————→
               └—— # < channel-number > : ——————

>——┬—— < array-name > ——┬—— (< subscript > ——┬—— , < subscript > ——┬——) ——┬——
   └—————————— < array-name > (?) ——————————————

MAT INV FUNCTION

—— MAT —— < array-name1 > —— = —— INV —— (< array-name2 >) ——|

MAT MULTIPLICATION STATEMENT

————— MAT < array-name1 > = < array-name2 > * < array-name3 > ————————————————
G18026
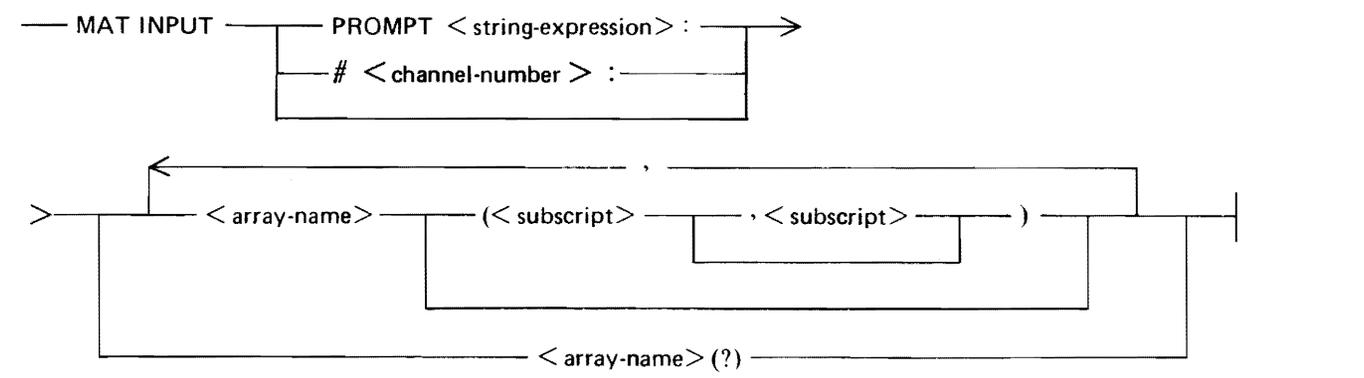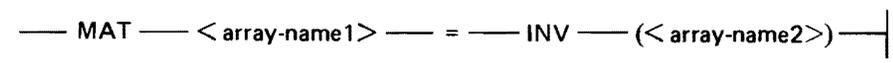
MAT NUL$ STATEMENT
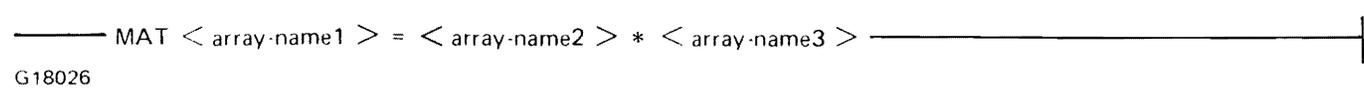
———— MAT < array-name > = NUL$ ——(< subscript > —— , < subscript > ——)
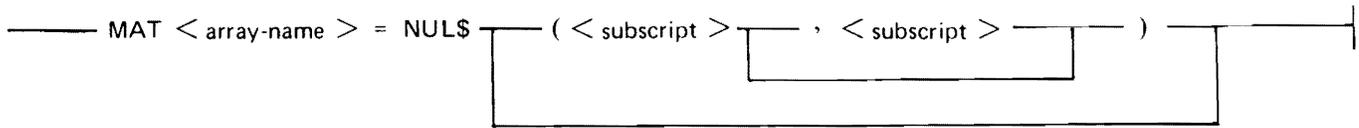
G18030

MAT OUTPUT STATEMENT

—— MAT OUTPUT #< channel-number>: —— < array-name> —— ; , ; ,
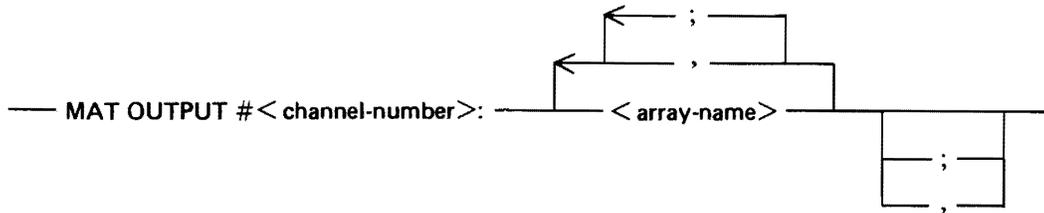
MAT PRINT STATEMENT

———— MAT PRINT —— <array-name > ; , : ,
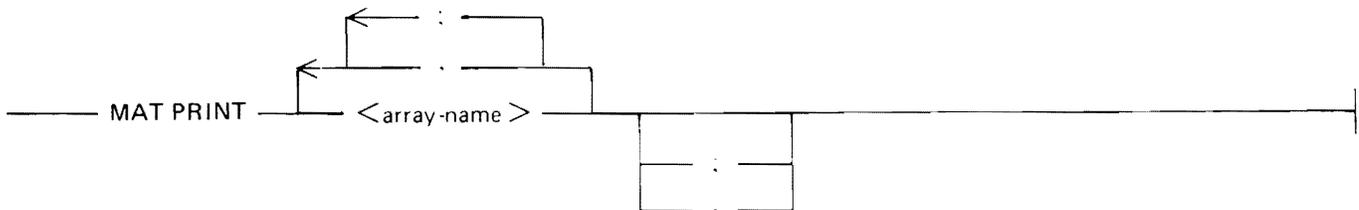
G18061

MAT READ STATEMENT

———— MAT READ —— <array-name > —— (<subscript > —— , <subscript > ——) ——
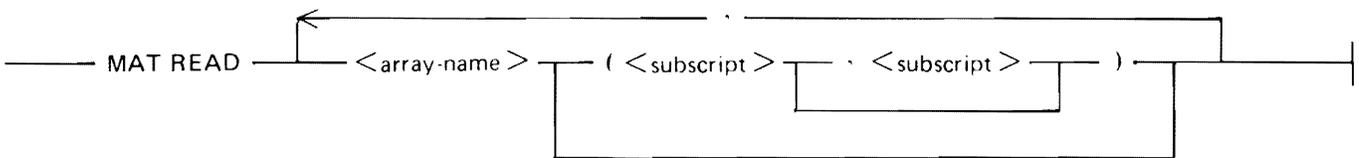
G18060

MAT SCALAR MULTIPLICATION STATEMENT

——— MAT < array-name > = —— < sign > —— (< numeric-expression >) * < array-name > ———
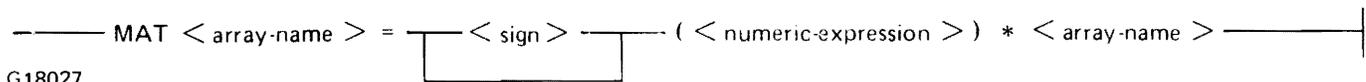
G18027

MAT SUBTRACTION STATEMENT

———— MAT < array-name1 > = < array-name2 > < minus-sign > < array-name3 > ————

G18028

MAT TRN FUNCTION

—— MAT —— < array-name1 > —— = —— TRN —— (< array-name2>) ——|

MAT ZER STATEMENT

——— MAT < array-name > = ——— (< numeric-expression >) —— * ———→

>—— ZER ——— (< subscript > —— , < subscript > ———) ———|

G18029

MAX FUNCTION

——— MAX (< numeric-expression > , < numeric-expression >) ———|

G18143

MCP FILE NAME

—— < BASIC-file-name > ——— / < BASIC-file-name > ———|
—— * < BASIC-file-name > ——
—— (< BASIC-file-name >) ——
—— * (< BASIC-file-name >) ——

G18076

MERGE COMMAND

——— MERGE ——————————— < MCP-file-name > ———|
—— < line-number-range > ——— —— ON < pack-name > ——
—— FROM ——

G18086

MIN FUNCTION

——— MIN (< numeric-expression > , < numeric-expression >) ———|

G18144

MOD FUNCTION

——— MOD (< numeric-expression > , < numeric-expression >) ———|

G18145

MULTIPLE-STATEMENT FUNCTION ASSIGNMENT STATEMENT

LET — <function-name> = <expression>
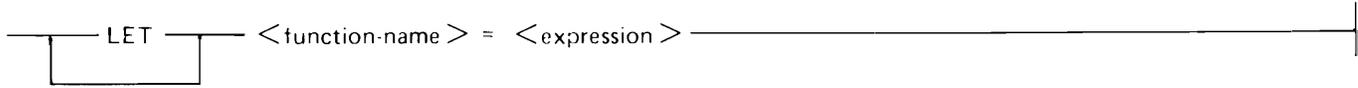
G18043

NEXT STATEMENT

NEXT <control-var>
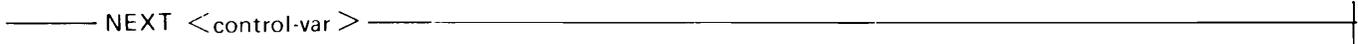
G18038

NUMERIC ASSIGNMENT STATEMENT

LET — <numeric-variable> , — = <numeric-expression>
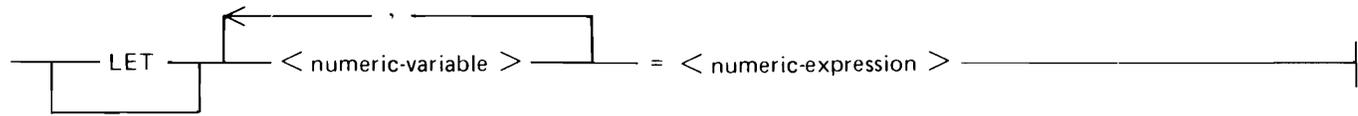
G18007

NUMERIC CONSTANT

<sign> <integer> . <integer> E <sign> <integer>
. <integer>
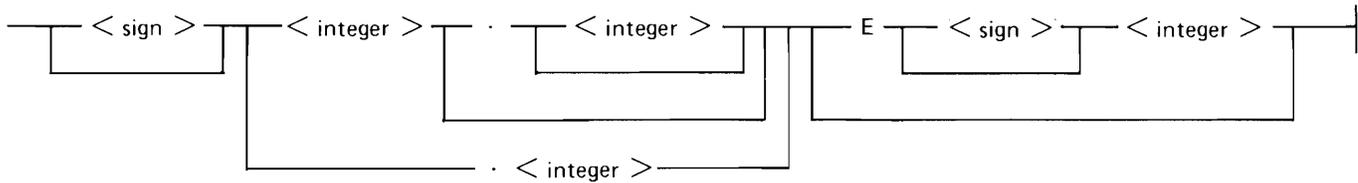
G18005

NUMERIC EXPRESSION

<sign> — <operator>
<numeric-constant>
<numeric-variable>
<numeric-function-ref>
(<numeric-expression>)
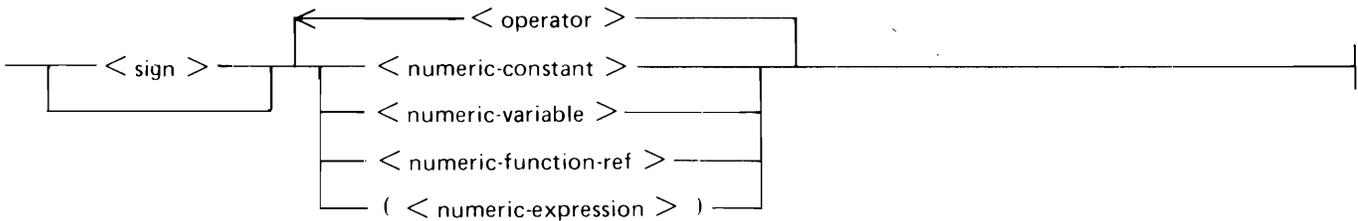
G18008

NUMERIC VARIABLE

<letter> <digit> (<subscript> , <subscript>)
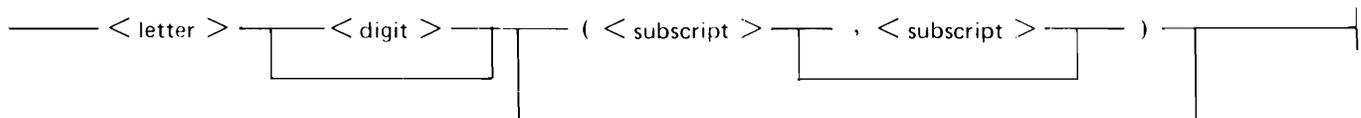
G18006

.OL COMMAND

G18146

ON GOSUB STATEMENT
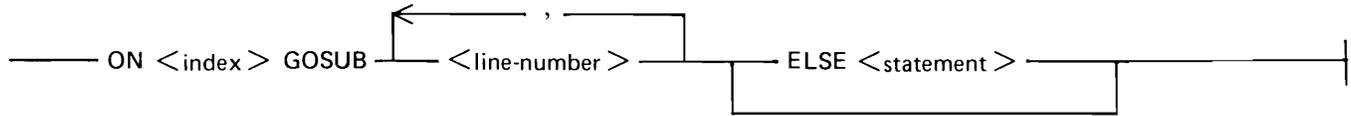
G18036

ON GOTO STATEMENT
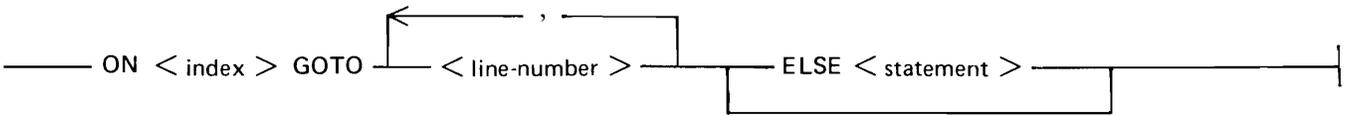
G18035

OPEN STATEMENT
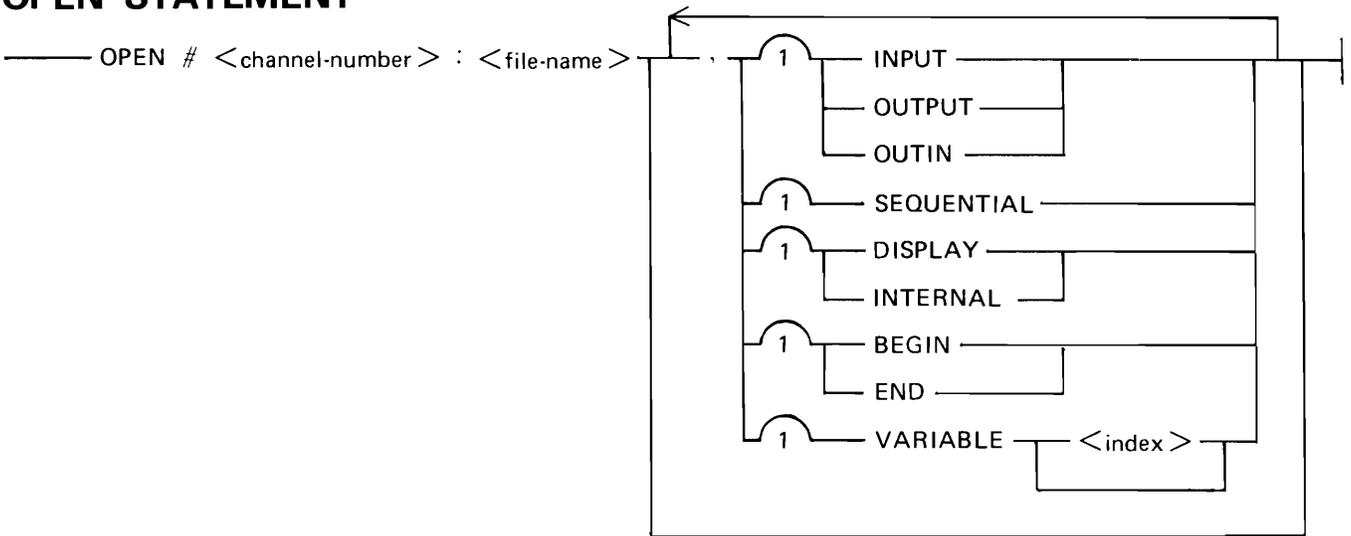
G18062

OPTION STATEMENT

G18147

ORD FUNCTION

```
————— ORD ( <string-expression> ) ————————————————————————————|
G18148
```

OUTPUT STATEMENT

```
————— OUTPUT # <channel-number> ————————————————————————————|

                                    ,
                                    ;
                          <expression>
                          TAB (<index>)                        ,
                                                                ;

G18066
```

.OVERLAY COMMAND

```
————— .OVERLAY ————————————————————————————————————————|
G18149
```

PACK NAME

```
————— <BASIC-file-name> ——————————————————————————————————|
     " <up to 10 blanks> "
G18075
```

PASSWORD COMMAND

```
————— PASSWORD <old-password> <new-password> <new-password> ————————|
G18087
```

PI FUNCTION

```
————— PI ———————————————————————————————————————————————|
G18150
```

POS FUNCTION

```
————— POS ( <string-exp> , <string-exp> —— <numeric-exp> —— ) ——————————|

G18151
```
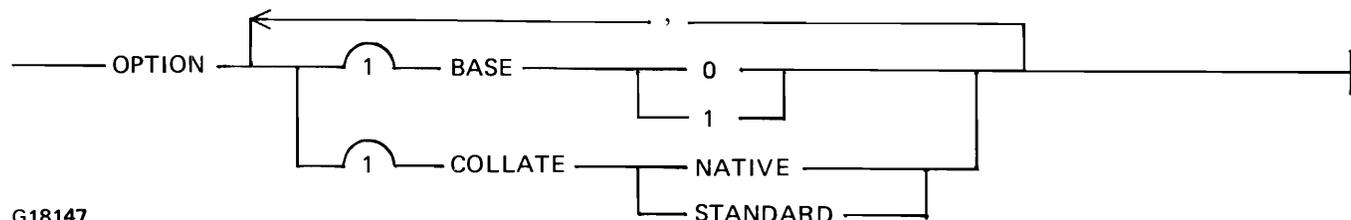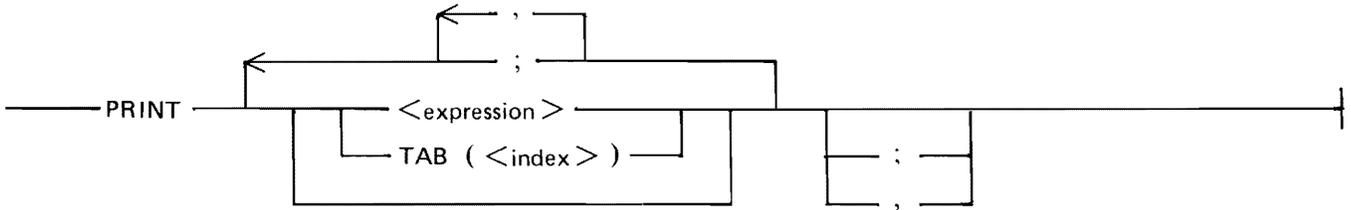
PRINT STATEMENT

G18053

PRINT USING STATEMENT
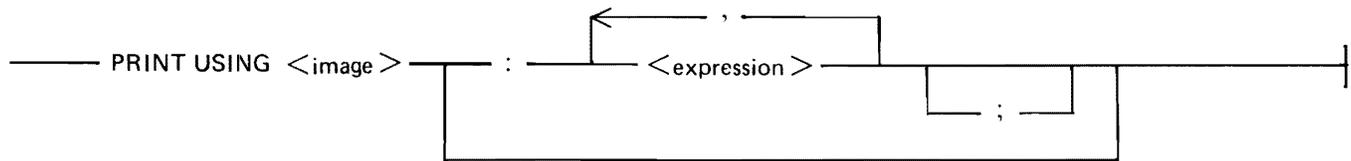
G18054

.PROMPT COMMAND

```
————— . PROMPT ————————————————————————————————————————————
```

G18152

RAD FUNCTION

```
————— RAD  (  <numeric-expression>  ) ——————————————————————
```

G18153

RANDOMIZE STATEMENT

```
————— RANDOMIZE ——————————————————————————————————————————
```

G18010

READ STATEMENT

G18047

RELATIONAL EXPRESSION

```
  ┌──────────────┐  ┌─ <exp><rel> <exp> ─────────────┐
──┤              ├──┤                                 ├──────────────────────────>
  └─ NOT ────────┘  └─ ( <relational-expression> ) ──┘

   ┌◄───────────────────────────────────────────────────────┐
>──┤                                                         ├──────────────────┤
   │  ┌─ AND ──┐  ┌─ NOT ──┐  ┌─ <exp><rel><exp> ──────────┐ │
   └──┤        ├──┤        ├──┤                             ├─┘
      └─ OR ───┘  └────────┘  └─ ( <relational-expression> ) ─┘
```
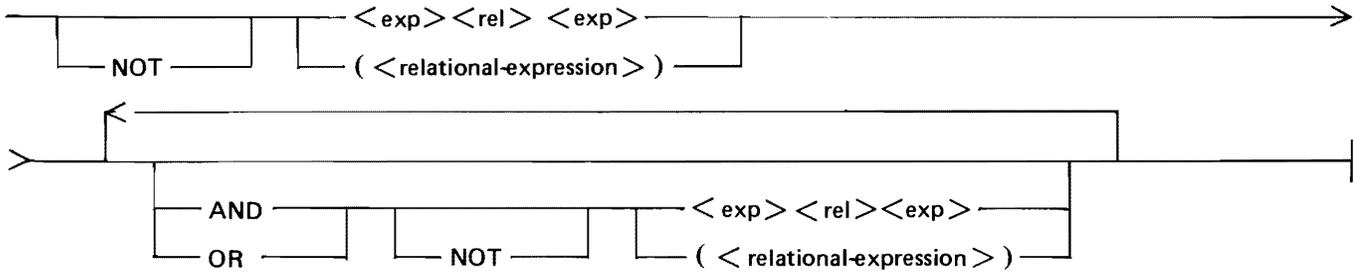
G18031

REM FUNCTION

```
────── REM  ( <numeric-expression> , <numeric-expression> ) ──────────────────┤
```
G18154

REM STATEMENT

```
────── REM  < remark-string > ────────────────────────────────────────────────┤
```
G18002

RENAME COMMAND
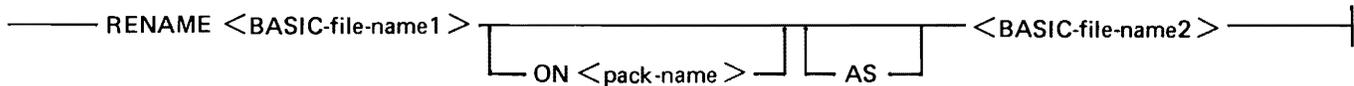
```
────── RENAME <BASIC-file-name1> ─┬──────────────────────┬─ <BASIC-file-name2> ──┤
                                  └─ ON <pack-name> ──┘ └─ AS ─┘
```
G18089

RENUMBER COMMAND
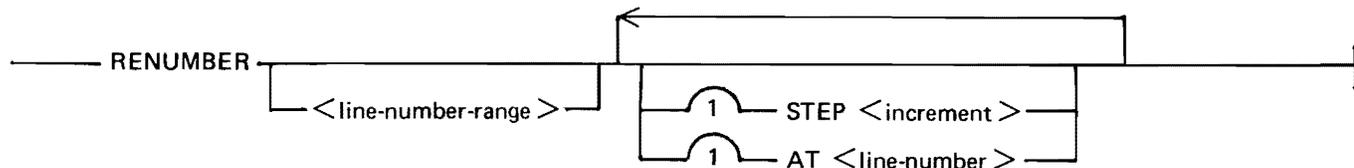
```
                                    ┌◄──────────────────────────┐
────── RENUMBER ─┬───────────────────┬──┤                        ├────────────────┤
                 └─ <line-number-range> ─┘  ┌─ 1 ─ STEP <increment> ─┐
                                            ├────────────────────────┤
                                            └─ 1 ─ AT <line-number> ──┘
```
G18090

RESTORE STATEMENT
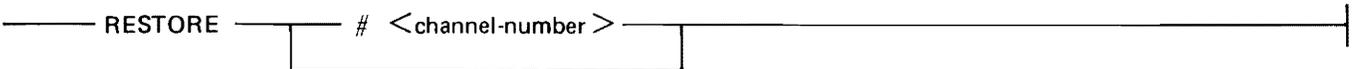
```
────── RESTORE ──┬── # <channel-number> ──┬──────────────────────────────────────┤
                 └─────────────────────────┘
```
G18155

RETURN STATEMENT

```
────── RETURN ─────────────────────────────────────────────────────────────────┤
```
G18034

D-18

RND FUNCTION

—— RND ————————————————————————————————|

G18156

RUN COMMAND

—— RUN ——┬————————————┬——┬—— 1 —— TRACE ——┬——————|
          └— <line-number> —┘ └—— 1 —— PRINT ——┘

G18091
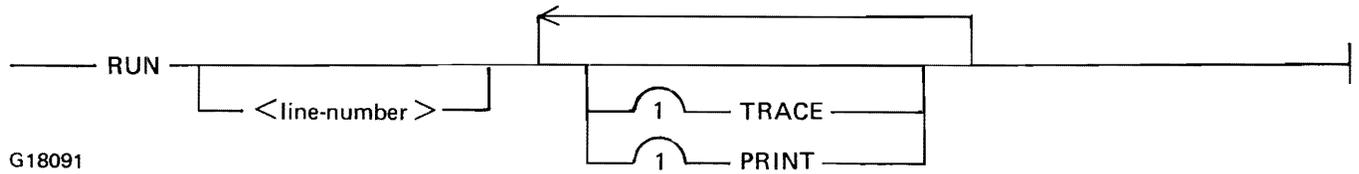
.RY COMMAND

—— .RY ————————————————————————————————|

G18157

SAVE COMMAND

—— SAVE ——┬————————————┬——┬————————————┬——┬——————————┬——|
          └— AS <BASIC-file-name> —┘ └— ON <pack-name> —┘ └— FOR CANDE —┘

G18092
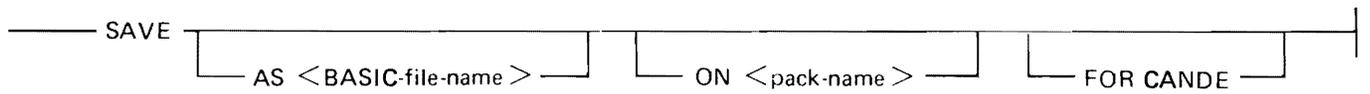
SCRATCH COMMAND

—— SCRATCH ——┬——————————————————————————————————┬——|
             └— <BASIC-file-name> ——┬————————————┬——┘
                                    └— ON <pack-name> —┘

G18093
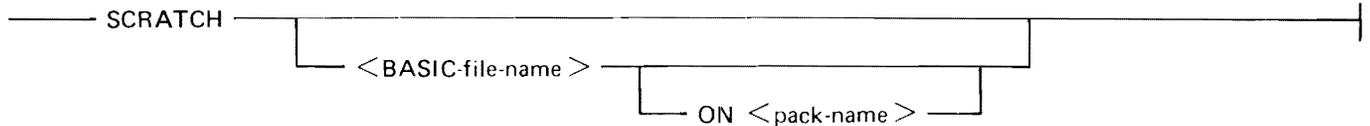
SCRATCH STATEMENT

—— SCRATCH # <channel-number> ————————————————————|

G18069

SEC FUNCTION

—— SEC (<numeric-expression>) ——————————————————|

G18158

SELECT CASE STRUCTURE

```
—— SELECT —— <expression> —— <eol> —→

> —— CASE ——┬── <constant> ──┬── <eol> ──┬── <statement> ──┬──→
             ├── <constant> — TO ──┤
             ├── <less-than-sign> ──┤
             └── <greater-than-sign> ──┘

> ——┬── CASE ELSE —— <eol> ──┬── <statement> ──┬── END SELECT ──┤
    └──────────────────────────────────────────┘
```

SEQUENCE COMMAND

```
—— SEQUENCE ──┬───────────────────────────────────┤
              ├──┬─ 1 ──┬── STEP —— <increment> ──┤
              │  └──────┘                          │
              └──┬─ 1 ──┬── AT ──┬── <line-number> ──┘
                 └──────┘
```
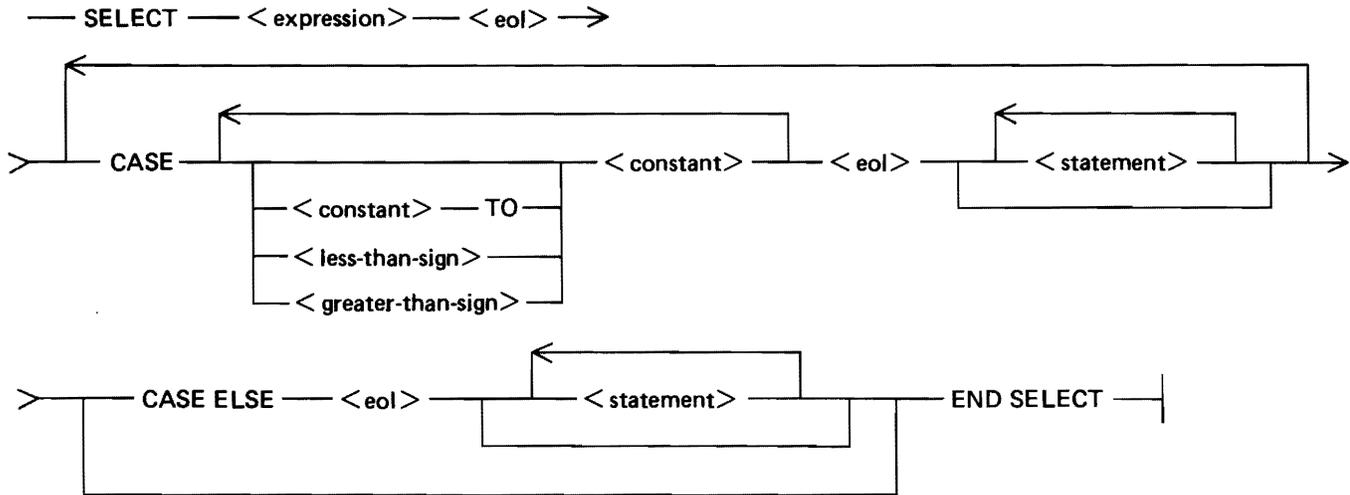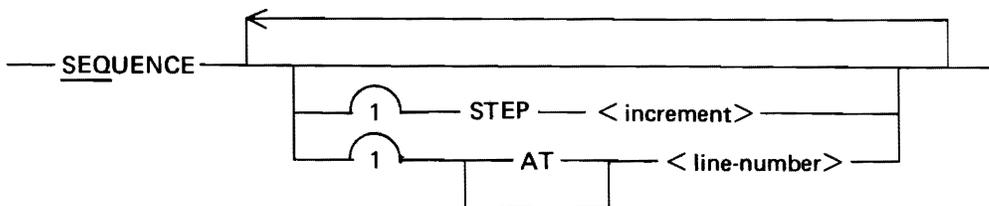
SGN FUNCTION

```
——— SGN ( <numeric-expression> ) ————————————————————┤
```
G18159

SIN FUNCTION

```
——— SIN ( <numeric-expression> ) ————————————————————┤
```
G18160

SINH FUNCTION

```
——— SINH ( <numeric-expression> ) ———————————————————┤
```
G18161

SQR FUNCTION

```
——— SQR ( <numeric-expression> ) ————————————————————┤
```
G18162

.SS COMMAND

——— .SS <string> —————————————————————————————————|

G18163

.ST COMMAND

——— .ST ———————————————————————————————————————|

G18164

STATEMENT LINE
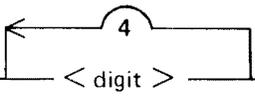
```
       ┌──← 4 ──┐
———┬── < digit > ──┬── < BASIC-language-statement > ——————————————|
```

G18000

.STATUSLINE COMMAND

——— .STATUSLINE ——————————————————————————————————|

G18165

STEP COMMAND

——— STEP ———————————————————————————————————————|

G18094

STOP STATEMENT

——— STOP ———————————————————————————————————————|

G18003

STR$ FUNCTION

——— STR$ (<numeric-expression>) ————————————————————————|

G18166

STRING ASSIGNMENT STATEMENT
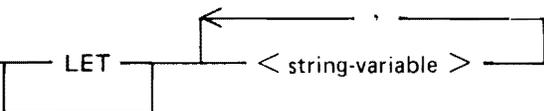
```
        ┌──────← , ──────┐
——┬── LET ──┬──┤── < string-variable > ──┤── = < string-expression > ————————|
  └─────────┘
```

G18013

STRING CONSTANT

```
            ┌──────← 254 ──────┐
———── " ──┬── < quoted-string-character > ──┬── " ——————————————————————|
          └─────────────────────────────────┘
```

G18011
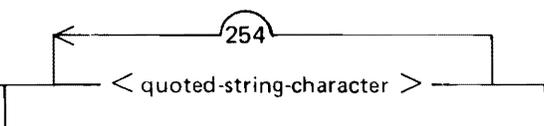
STRING EXPRESSION

G18014

STRING VARIABLE
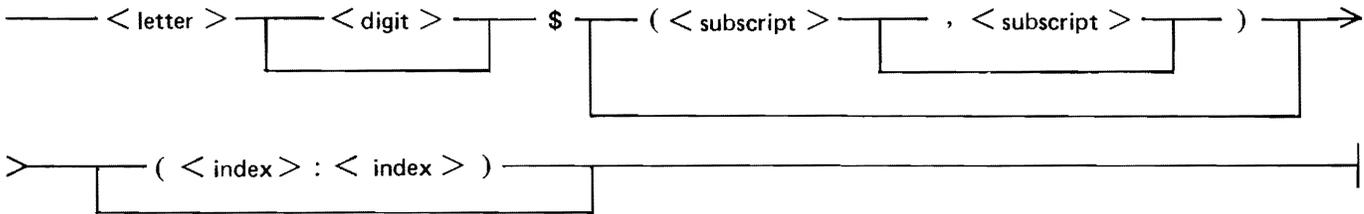
G18012

TAIL COMMENT
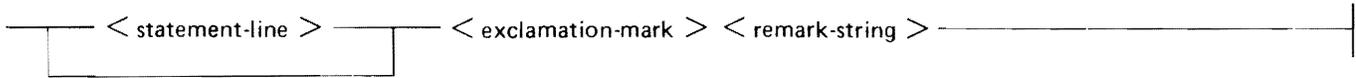
G18001

TAN FUNCTION

G18167

TANH FUNCTION

G18168

TEACH COMMAND
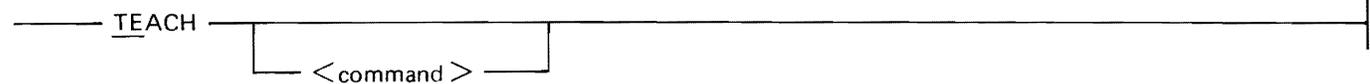
G18095

TIME FUNCTION

G18169

.TIME COMMAND

```
————— .TIME ——————————————————————————————————————————————————————|
G18170
```

TIME$ FUNCTION

```
————— TIME$ ——————————————————————————————————————————————————————|
G18171
```

TITLE COMMAND

```
————— TITLE  <BASIC-file-name> ———————————————————————————————————|
G18096
```

TRACE STATEMENT

```
————— TRACE ——┬—— ON ——┬————————————————————————————————————————|
              └—— OFF ——┘
G18072
```

UDIM

```
————— UDIM ( <array-name> , <numeric-expression> ) ——————————————|
G18172
```

USER COMMAND

```
————— USER <usercode> ——┬——————┬——┬——————————————┬———————————————|
                        └— / ——┘  └— <password> —┘
G18097
```

VAL FUNCTION

```
————— VAL ( <string-expression> ) ———————————————————————————————|
G18173
```

WALK COMMAND

```
————— WALK ——┬————————————————————┬———————————————————————————————|
             └— <line-number> ——┘
G18098
```

WHAT COMMAND

```
————— WHAT ———————————————————————————————————————————————————————|
G18099
```

WHERE COMMAND

```
———— WHERE ———————————————————————————————————————————————|
                 ├— FROM ——┤
                 └→ CALLED ——┘
```
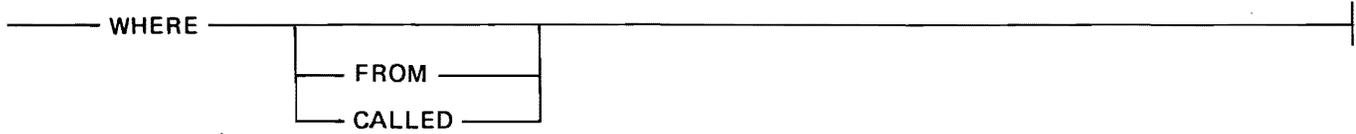
G18100

XREF COMMAND

```
—┬—XREF—┬—┬— <variable-name> ————————————————————————————————|
 └—FIND—┘ ├— <line-number> ————————————┤  ├— <line-number-range> —┘
          ├— <line-number-range> ——————┤
          ├— <user-defined-function> —┤
          └— <delim> <string> <delim> —┘
```

G18101

D-24
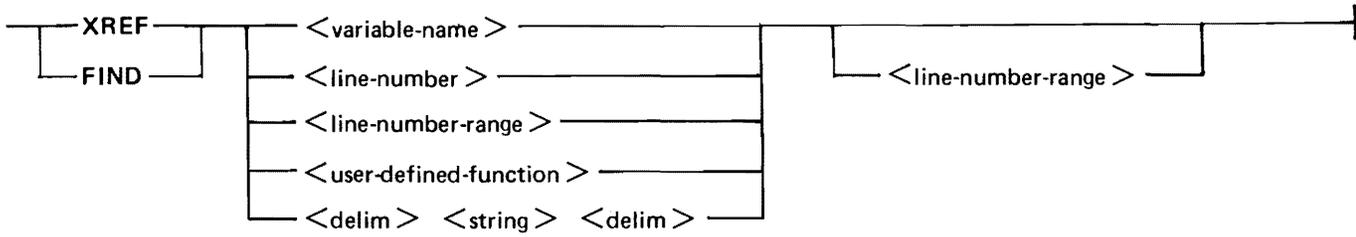
APPENDIX E
CHARACTER SETS

Tables E-1 and E-2 list the character sets available to IBASIC for comparing strings and for computing values with the CHR$ and ORD functions. Table E-1 contains the STANDARD character set (ASCII). Table E-2 contains the NATIVE character set (EBCDIC).

Hexadecimal representation is the standard convention for the 8-bit internal codes. Examples of the translation of these codes to the equivalent binary values follows.

Examples:

Hex Number Pair		8-Bit Internal Code	
		8 4 2 1	8 4 2 1
@39@	=	0 0 1 1	1 0 0 1
@BE@	=	1 0 1 1	1 1 1 0
@0F@	=	0 0 0 0	1 1 1 1

Table E-1. Standard BASIC Character Set (ASCII)

Ordinal Position	Hex Code	Graphic	ORD Mnemonic	Name
0	00		NUL	Null
1	01		SOH	Start of heading
2	02		STX	Start of text
3	03		ETX	End of text
4	04		EOT	End of transmission
5	05		ENQ	Enquiry
6	06		ACK	Acknowledge
7	07		BEL	Bell
8	08		BS	Backspace
9	09		HT	Horizontal tab
10	0A		LF	Line feed
11	0B		VT	Vertical tab
12	0C		FF	Form Feed
13	0D		CR	Carriage Return
14	0E		SO	Shift Out
15	0F		SI	Shift In
16	10		DLE	Data link escape
17	11		DC1	Device control 1
18	12		DC2	Device control 2
19	13		DC3	Device control 3
20	14		DC4	Device control 4
21	15		NAK	Negative Acknowledge
22	16		SYN	Synchronous idle
23	17		ETB	End of transmission block
24	18		CAN	Cancel
25	19		EM	End of medium

Table E-1. Standard BASIC Character Set (Cont) (ASCII)

Ordinal Position	Hex Code	Graphic	ORD Mnemonic	Name
26	1A		SUB	Substitute
27	1B		ESC	Escape
28	1C		FS	File separator
29	1D		GS	Group separator
30	1E		RS	Record separator
31	1F		US	Unit separator
32	20		SP	Space
33	21	!		Exclamation mark
34	22	"		Quotation mark
35	23	#		Number sign
36	24	$		Dollar sign
37	25	%		Percent sign
38	26	&		Ampersand
39	27	'		Apostrophe
40	28	(		Left parenthesis
41	29	)		Right parenthesis
42	2A	*		Asterisk
43	2B	+		Plus sign
44	2C	,		Comma
45	2D	–		Minus sign, hyphen
46	2E	.		Full stop, period, or decimal point
47	2F	/		Solidus
48	30	0		Zero
49	31	1		One
50	32	2		Two
51	33	3		Three
52	34	4		Four
53	35	5		Five
54	36	6		Six
55	37	7		Seven
56	38	8		Eight
57	39	9		Nine
58	3A	:		Colon
59	3B	;		Semicolon
60	3C	<		Less than sign
61	3D	=		Equals sign
62	3E	>		Greater than sign
63	3F	?		Question mark
64	40	@		At sign
65	41	A		Upper-case A
66	42	B		Upper-case B
67	43	C		Upper-case C
68	44	D		Upper-case D
69	45	E		Upper-case E
70	46	F		Upper-case F
71	47	G		Upper-case G
72	48	H		Upper-case H
73	49	I		Upper-case I
74	4A	J		Upper-case J

Table E-1. Standard BASIC Character Set (Cont) (ASCII)

Ordinal Position	Hex Code	Graphic	ORD Mnemonic	Name
75	4B	K		Upper-case K
76	4C	L		Upper-case L
77	4D	M		Upper-case M
78	4E	N		Upper-case N
79	4F	O		Upper-case O
80	50	P		Upper-case P
81	51	Q		Upper-case Q
82	52	R		Upper-case R
83	53	S		Upper-case S
84	54	T		Upper-case T
85	55	U		Upper-case U
86	56	V		Upper-case V
87	57	W		Upper-case W
88	58	X		Upper-case X
89	59	Y		Upper-case Y
90	5A	Z		Upper-case Z
91	5B	[		Left bracket
92	5C	\		Reverse solidus
93	5D	]		Right bracket
94	5E	^		Circumflex accent
95	5F	_	UND	Underline
96	60	`	GRA	Grave accent
97	61	a	LCA	Lower-case a
98	62	b	LCB	Lower-case b
99	63	c	LCC	Lower-case c
100	64	d	LCD	Lower-case d
101	65	e	LCE	Lower-case e
102	66	f	LCF	Lower-case f
103	67	g	LCG	Lower-case g
104	68	h	LCH	Lower-case h
105	69	i	LCI	Lower-case i
106	6A	j	LCJ	Lower-case j
107	6B	k	LCK	Lower-case k
108	6C	l	LCL	Lower-case l
109	6D	m	LCM	Lower-case m
110	6E	n	LCN	Lower-case n
111	6F	o	LCO	Lower-case o
112	70	p	LCP	Lower-case p
113	71	q	LCQ	Lower-case q
114	72	r	LCR	Lower-case r
115	73	s	LCS	Lower-case s
116	74	t	LCT	Lower-case t
117	75	u	LCU	Lower-case u
118	76	v	LCV	Lower-case v
119	77	w	LCW	Lower-case w
120	78	x	LCX	Lower-case x
121	79	y	LCY	Lower-case y
122	7A	z	LCZ	Lower-case z
123	7B	{	LBR	Left brace

Table E-1. Standard BASIC Character Set (ASCII) (Cont)

Ordinal Position	Hex Code	Graphic	ORD Mnemonic	Name
124	7C	|	VLN	Vertical line
125	7D	}	RBR	Right brace
126	7E	~	TIL	Tilde
127	7F		DEL	Delete

Table E-2. Native BASIC Character Set (EBCDIC)

Ordinal Position	Hex Code	Graphic	ORD Mnemonic	Name
0	00		NUL	Null
1	01		SOH	Start of heading
2	02		STX	Start of text
3	03		ETX	End of text
4	04			
5	05		HT	Horizontal tab
6	06			
7	07		DEL	Delete
8	08			
9	09			
10	0A			
11	0B		VT	Vertical tab
12	0C		FF	Form feed
13	0D		CR	Carriage return
14	0E		SO	Shift out
15	0F		SI	Shift in
16	10		DLE	Data link escape
17	11		DC1	Device control 1
18	12		DC2	Device control 2
19	13		DC3	Device control 3
20	14			
21	15		NL	New line
22	16		BS	Backspace
23	17			
24	18		CAN	Cancel
25	19		EM	End of medium
26	1A			
27	1B			
28	1C		FS	File separator
29	1D		GS	Group separator
30	1E		RS	Record separator
31	1F		US	Unit separator
32	20			
33	21			

Table E-2. Native BASIC Character Set (Cont) (EBCDIC)

Ordinal Position	Hex Code	Graphic	ORD Mnemonic	Name
34	22			
35	23			
36	24			
37	25		LF	Line feed
38	26		ETB	End of transmission block
39	27		ESC	Escape
40	28			
41	29			
42	2A			
43	2B			
44	2C			
45	2D		ENQ	Enquiry
46	2E		ACK	Acknowledge
47	2F		BEL	Bell
48	30			
49	31			
50	32		SYN	Synchronous idle
51	33			
52	34			
53	35			
54	36			
55	37		EOT	End of transmission
56	38			
57	39			
58	3A			
59	3B			
60	3C		DC4	Device control 4
61	3D		NAK	Negative acknowledge
62	3E			
63	3F			
64	40		SP	Space
65	41			
66	42			
67	43			
68	44			
69	45			
70	46			
71	47			
72	48			
73	49			
74	4A	[		Left bracket
75	4B	.		Full stop, period, decimal point
76	4C	<		Less than sign
77	4D	(		Left parenthesis
78	4E	+		Plus sign
79	4F	|	VLN	Vertical line
80	50	&		Ampersand
81	51			

Table E-2. Native BASIC Character Set (Cont) (EBCDIC)

Ordinal Position	Hex Code	Graphic	ORD Mnemonic	Name
82	52			
83	53			
84	54			
85	55			
86	56			
87	57			
88	58			
89	59			
90	5A	]		Right bracket
91	5B	$		Dollar sign
92	5C	*		Asterisk
93	5D	)		Right parenthesis
94	5E	;		Semicolon
95	5F	∧		Circumflex accent
96	60	−		Minus sign, hyphen
97	61	/		Solidus
98	62			
99	63			
100	64			
101	65			
102	66			
103	67			
104	68			
105	69			
106	6A			
107	6B	,		Comma
108	6C	%		Percent sign
109	6D	—	UND	Underline
110	6E	>		Greater than sign
111	6F	?		Question mark
112	70	!		Exclamation mark
113	71			
114	72			
115	73			
116	74			
117	75			
118	76			
119	77			
120	78			
121	79			
122	7A	:		Colon
123	7B	#		Number sign
124	7C	@		At sign
125	7D	'		Apostrophe
126	7E	=		Equals sign
127	7F	"		Quotation mark
128	80			
129	81	a	LCA	Lower-case a

Table E-2. Native BASIC Character Set (Cont) (EBCDIC)

Ordinal Position	Hex Code	Graphic	ORD Mnemonic	Name
130	82	b	LCB	Lower-case b
131	83	c	LCC	Lower-case c
132	84	d	LCD	Lower-case d
133	85	e	LCE	Lower-case e
134	86	f	LCF	Lower-case f
135	87	g	LCG	Lower-case g
136	88	h	LCH	Lower-case h
137	89	i	LCI	Lower-case i
138	8A			
139	8B			
140	8C			
141	8D			
142	8E			
143	8F			
144	90			
145	91	j	LCJ	Lower-case j
146	92	k	LCK	Lower-case k
147	93	l	LCL	Lower-case l
148	94	m	LCM	Lower-case m
149	95	n	LCN	Lower-case n
150	96	o	LCO	Lower-case o
151	97	p	LCP	Lower-case p
152	98	q	LCQ	Lower-case q
153	99	r	LCR	Lower-case r
154	9A			
155	9B			
156	9C			
157	9D			
158	9E			
159	9F			
160	A0			
161	A1	~	TIL	Tilde
162	A2	s	LCS	Lower-case s
163	A3	t	LCT	Lower-case t
164	A4	u	LCU	Lower-case u
165	A5	v	LCV	Lower-case v
166	A6	w	LCW	Lower-case w
167	A7	x	LCX	Lower-case x
168	A8	y	LCY	Lower-case y
169	A9	z	LCZ	Lower-case z
170	AA			
171	AB			
172	AC			
173	AD			
174	AE			
175	AF			
176	B0			
177	B1			

Table E-2. Native BASIC Character Set (Cont) (EBCDIC)

Ordinal Position	Hex Code	Graphic	ORD Mnemonic	Name
178	B2			
179	B3			
180	B4			
181	B5			
182	B6			
183	B7			
184	B8			
185	B9			
186	BA			
187	BB			
188	BC			
189	BD			
190	BE			
191	BF			
192	C0	{	LBR	Left brace
193	C1	A		Upper-case A
194	C2	B		Upper-case B
195	C3	C		Upper-case C
196	C4	D		Upper-case D
197	C5	E		Upper-case E
198	C6	F		Upper-case F
199	C7	G		Upper-case G
200	C8	H		Upper-case H
201	C9	I		Upper-case I
202	CA			
203	CB			
204	CC			
205	CD			
206	CE			
207	CF			
208	D0	}	RBR	Right brace
209	D1	J		Upper-case J
210	D2	K		Upper-case K
211	D3	L		Upper-case L
212	D4	M		Upper-case M
213	D5	N		Upper-case N
214	D6	O		Upper-case O
215	D7	P		Upper-case P
216	D8	Q		Upper-case Q
217	D9	R		Upper-case R
218	DA			
219	DB			
220	DC			
221	DD			
222	DE			
223	DF			
224	E0	\		Reverse solidus
225	E1			

Table E-2. Native BASIC Character Set (Cont) (EBCDIC)

Ordinal Position	Hex Code	Graphic	ORD Mnemonic	Name
226	E2	S		Upper-case S
227	E3	T		Upper-case T
228	E4	U		Upper-case U
229	E5	V		Upper-case V
230	E6	W		Upper-case W
231	E7	X		Upper-case X
232	E8	Y		Upper-case Y
233	E9	Z		Upper-case Z
234	EA			
235	EB			
236	EC			
237	ED			
238	EE			
239	EF			
240	F0	0		Zero
241	F1	1		One
242	F2	2		Two
243	F3	3		Three
244	F4	4		Four
245	F5	5		Five
246	F6	6		Six
247	F7	7		Seven
248	F8	8		Eight
249	F9	9		Nine
250	FA			
251	FB			
252	FC			
253	FD			
254	FE			
255	FF			

APPENDIX F
EXTENSIONS TO BASIC

The following extensions to the ANSI language are recognized only when the IBASIC program is executed with SW = W or with the initial parameter of DEBUG WW (refer to appendix C). The intent of these extensions is to enable the writing of external intrinsic functions and subprograms in BASIC. A side effect of this extension is that while the IBASIC program is being executed with SW = W or an initial parameter of DEBUG WW, the intrinsic file it is using is inaccessible to any other copy of the IBASIC program. Therefore, it is desirable to have a separate copy of the intrinsic file (not the system copy) when it is desired to run in this mode. By default, the IBASIC program uses the interpreter named *IBASIC/INTERP. If it is desired to use some other interpreter (for example, for experimental purposes), the IBASIC program should be initiated with SW = W or AX DEBUG WW and FILE INTRINSICS NAME <your own interpreter name>.

INTRINSIC STATEMENT

The INTRINSIC statement is used to identify an intrinsic.

Syntax:

G18174

Semantics:

The INTRINSIC statement must be the first statement of an intrinsic function. The <intrinsic-name> must match an entry in the intrinsic function table in the compiler and the <parameters> must match, in type and number, the parameters of this entry. The <local-variable-list> may be any valid BASIC simple variable (up to seven variables). The statements that follow may only reference those names that are either in the <parameters> or in the <local-variable-list> (enforced at COMPILE or RUN time). If the intrinsic being defined is run, it supercedes the intrinsic in the intrinsic file. The external intrinsic file is only changed after a COMPILE INTRINSICS command is executed. While the local intrinsic is being executed, the <parameters> and the <local-variable-list> are global. The compilation process and ensuing intrinsic call process "localize" the references.

The parameters may be arrays which are passed by reference. Array parameters are only allowed if the intrinsic requires them and are denoted by appending a left parenthesis "(" to the array name which is passed. Within the intrinsic, the array may be referenced with one or two subscripts (with obviously bad results if a 2-dimensional array is accessed with one subscript, and vice versa). The special function NDIM should be used to ensure correct usage. Refer to Special Functions in this section.

Within the intrinsic definition the following statements are invalid:

Array references (unless they are passed as parameters)
CHAIN
CLOSE
DATA
DEF (and any FN references)
DIM
END
FNEND
INPUT
LINPUT
MARGIN
MAT
OPTION
READ
RESTORE
SCRATCH
STOP
SUB
SUBEND
SUBEXIT

Example of an INTRINSIC statement:

INTRINSIC SIN (A) X,Y,Z

INTEND STATEMENT

The INTEND statement identifies the end of an intrinsic and returns the value of the intrinsic.

Syntax:

```
————— INTEND ————┬— <numeric-expression> —┬—————————————————————————|
                 └— <array-name> ( ————┘
```

G18175

Semantics:

The INTEND statement must be the last statement in the intrinsic. Its function is to return the value of the intrinsic or to return an array name to the calling code. There must be a corresponding INTRIN-SIC statement. Normal BASIC statements or another INTRINSIC statement may follow.

ERROR STATEMENT

The ERROR statement allows error messages to be displayed.

Syntax:

```
————ERROR ( ——┬—— FATAL ————————┬—— , <integer> ) ——————————————————|
               └—— NONFATAL ————┘
```

G18176

The ERROR statement causes the displaying of an error message which is the <integer>th one in the list of error messages in the compiler. If NONFATAL is specified, execution continues. If FATAL is specified, intrinsic, and thus program (or command) execution, is terminated immediately. The following error numbers are currently used with the intrinsics:

Error Number	Meaning
14	Numeric overflow
15	Numeric underflow
102	Zero argument
103	Negative argument
104	Negative number to nonintegral power
105	Argument too big, inaccurate result
106	Argument out of range
107	Both arguments zero
108	Zero to negative power
133	Square matrix required
134	Array dimension mismatch

SPECIAL FUNCTIONS

The following special functions are available with the extensions to BASIC.

NDIM(X)

Returns the number of dimensions of array X.

MSV(X)

Returns the number of elements that were originally defined for array X, whether the array was explicitly or implicitly dimensioned. Dynamic redimensioning does not change this value.

MXI

Returns the maximum integer that the current numeric representation may hold precisely.

RDUC(X)

Returns the value of X, such that $.5 <= X < 1$, as if X were divided or multiplied by 2 repetitively until X is in that range.

XPND(X,Y)

Returns the value of X as if multiplied by 2**Y.

XPON(X)

Returns the power to which the value 2 must be raised so that X/2**XPON(X) results in .5 < = X < 1.

XTIM

Returns the current cpu time usage of IBASIC in tenths of a second.

SPECIAL VARIABLE NAMES

The following special variables can be used when the extensions are enabled.

DET
   When used as a destination in an assignment statement, DET assigns a value to the function DET (no parameters).

RAN1, RAN2, and RAN3
   May be used as sources or destinations. These variables are initialized to particular values at the beginning of any particular RUN of a program and are intended for use by the RND intrinsic function.

COMPILE COMMAND

The COMPILE command compiles intrinsics.

Syntax:

———— COMPILE ———— INTRINSICS ——┤
G18178

Semantics:

The COMPILE command is used to convert the locally defined intrinsic function to a form which can be put into the external intrinsic file. This code is then written to the intrinsic file, overwriting whatever code may already have been compiled for the function or functions being defined locally. The local definition of the function is used until the local definition is deleted from the workfile. If a syntax error exists in the workfile, nothing is written to the external file, and Command mode invocations of the intrinsic function yield an "invalid op code" run-time error message.

EXTERNAL INTRINSICS

The following intrinsic functions are defined as external intrinsics and are maintained in a 'compiled' form in the intrinsics file:

ACOS
ASIN
ATN
ATN2
COS
COSH
COT
CSC
EXP
LOG
LOG10
LOG2
RND
SEC
SIN
SINH
SQR
TAN
TANH

INDEX

INDEX (CONT)

INDEX (CONT)

INDEX (CONT)

INDEX (CONT)

INDEX (CONT)

6

INDEX (CONT)

INDEX (CONT)

INDEX (CONT)

INDEX (CONT)

Documentation Evaluation Form

Title: __B 1000 Systems Interactive BASIC (IBASIC)__   Form No: __1152105__

Date: _____ *January 1984* _____

Burroughs Corporation is interested in receiving your comments and suggestions, regarding this manual. Comments will be utilized in ensuing revisions to improve this manual.

Please check type of Suggestion:

☐ Addition   ☐ Deletion   ☐ Revision   ☐ Error

Comments:

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

From:

Name _____

Title _____

Company _____

Address _____

_____

Phone Number _____   Date _____

Remove form and mail to:

Burroughs Corporation
Corporate Documentation — West
1300 John Reed Court
City of Industry, CA 91745
U.S.A.