DataGeneral

User's Manual

PROGRAM

EXTENDED BASIC

093-000065-05

ABSTRACT

Extended BASIC provides programmers with a BASIC system that operates under DGC'S Real Time Disk Operating System (RDOS) and Stand-alone Operating System (SOS). Extended BASIC may be configured with or without a disk, with or without the hardware Floating Point option, with or without swapping capabilities, with or without fixed point multiply/divide, with or without mapping facilities, and with single or double precision depending upon the operating system's configuration.

Extended BASIC offers a full implementation of the BASIC language as developed at Dartmouth College, plus additional language facilities such as string manipulation features, added user control of output formatting, assembly language callable subroutines, matrix operations, and completely automatic SYSGEN procedures.

User's Manual

PROGRAM

EXTENDED BASIC

093-000065-05

In addition, Extended BASIC has a large repertoire of commands that give the user immediate access to the central processor for desk calculator operation and dynamic testing and debugging.

User RDOS, users in both program mode and keyboard mode have access to the RDOS file capabilities including file I/O statements and commands, file directory maintenance, a program swapping facility, and a user file/directory accounting system.

Ordering No. 093-000065

Printed in the United States of America

Rev. 05, October 1974

NOTICE

Original Release - November, 1971
First Revision - May, 1972
Second Revision - September, 1972
Third Revision - March, 1973
Fourth Revision - September, 1973
Fifth Revision - October, 1974

This Revision of the Extended BASIC User's Manual, 093-000065-05, supersedes 093-000065-04, and constitutes a major revision. Specific changes include the incorporation of pages from Addendum No. 086-000010-00 and a completely revised Appendix D.

INTRODUCTION

Data General's Extended BASIC is available under either the Stand-alone Operating System (SOS) or under the Real Time Disk Operating System (RDOS). Extended BASIC systems may be either single-user systems or multi-user systems. Single-user systems may be configured with or without a disk.

Multi-user systems may be configured with or without a disk. When configured without a disk, the system is automatically designated as non-swapping, non-mapping. These multi-user non-swapping systems support a number of users simultaneously, allotting each user a fixed portion of memory. The system may be configured with multiplexor handler type 4060 or the 4100 multi-line asynchronous controller support, which support up to 32 users. It may also be configured with multiplexor handler type 4026, which supports up to 16 users.

Multi-user systems configured with a disk, i.e., operating under RDOS, may be configured as swapping or non-swapping Extended BASIC systems. Non-swapping multi-user disk systems operate the same as multi-user without disk configurations, described above. Multi-user Swapping Extended BASIC systems allow as many programs as user core will allow to run simultaneously. A program swap will occur when a program which is ready to execute is too large to fit into the unused portion of memory. One or more programs which are in core will be swapped out to disk to allow the 'new' program access to system resources.

Multi-user swapping systems may further be configured as mapping systems. Systems with the Memory Management and Protection Unit (MMPU) provide an absolute hardware protection to separate the foreground and background partitions. For example, while BASIC is running in the foreground, a FORTRAN IV program could be running in the background.

Extended BASIC systems under SOS treat devices as files and provide a set of file I/O statements and their comparable keyboard commands that allow the user to perform binary and ASCII I/O, chain from an executing program by bringing in another program from an input device, save a program, and enter a program to append it to the current program. Extended BASIC users under SOS may select from a wide variety of standard SOS I/O devices including one to eight magnetic tape units, one to eight cassette units, and a mark sense card reader. Under SOS up to eight device channels may be open at one time.

The Extended BASIC systems under RDOS support all the SOS devices as well as providing full disk file capabilities for either fixed or moveable head disk configurations. BASIC statements and commands allow users to search disk directories for files, load files located on disk into core, run the loaded files, chain from a running program by bringing in another program from disk or another input device,

save core images on disk files for later running, etc. Under RDOS up to 63 device/file channels may be open at any given time.

Extended BASIC systems offer all features of the BASIC language as originally developed at Dartmouth, as well as:

Keyboard mode of operation, which was provided in DGC's Stand-alone and Time-Sharing BASICs primarily for debugging and desk calculator uses, has been greatly expanded in Extended BASIC. The user now has the full range of BASIC statements and File I/O statements to use as keyboard commands, excluding only those statements that have no meaning except within program context (REM, FOR, etc.).

Error detection at program input time has been expanded so that all syntax errors are caught at this time.

String variables, string concatenation and string subsettings are implemented. String variables may appear in READ statements with corresponding literals in DATA statements.

Extended BASIC's format for storage of floating point numbers is compatible with other DGC software. Extended BASIC thus has FORTRAN/ALGOL compatibility through data files, and CALLed subroutines.

The user has been given far greater control over output formatting. Extended BASIC permits the user to use either the standard BASIC print formatting or to use statements and commands that include a picture specification of output similar to that available in COBOL.

Assignment statements that do not require the keyword LET are implemented.

A generalized IF statement, allowing a THEN clause that can be any statement, including another IF is implemented.

In addition, systems under RDOS have the following features:

BASIC statements and keyboard commands are available that allow the user to perform maintenance on his disk files: deleting files, changing file names, accessing file creation dates, etc.

A private subdirectory or subpartition can be allocated to each user for his files. By default, files in a subdirectory are private to a given user. However, these subdirectories can be shared among users if desired. This allows a flexibility whereby a user's file space can be limited to a specific amount of a disk or not, depending upon which choice is made.

Using the standard RDOS file structures, compatibility is provided so that these files could be processed by other DGC software.

The link capability of RDOS files may be utilized to provide sharing of user files for both reading and/or writing.

Multi-user systems provide capability for an accounting file. At SYSGEN time the user can request that such a file be provided. The file is used to keep a record by account ID of each user sign-on and the time.

The following Data General publications may be referred to for further discussion of an operating environment:

093-000075	Real Time Disk Operating System User's Manual
093-000083	Introduction to the Real Time Disk Operating System
093-000087	BATCH User's Manual
093-000062	Stand-Alone Operating System User's Manual

Below is a list of all BASIC keywords specifying whether they may be used as statement keywords or command keywords or both. (An X appearing in the column indicates yes.)

BASIC STATEMENTS AND COMMANDS

	BASIC KEY WORD	USED AS STATEMENT	COMMAND	PAGE REFERENCE
	BYE	X	X	3-3
	CALL	X		App. B
	CHAIN	X		5-16
	CLOSE	X	X	5-19
	CLOSE FILE	X	X	5-5
	CON		X	6-11
	DATA	X		3-40
	DEF	X		3-4
RDOS Only	DELETE	X	X	5-21
	DIM	X	X	3-5
	END	X		3-6
	ENTER	X	X	5-18
RDOS Only	FILES		X	6-4
	FOR	X		3-7
	GOSUB	X		3-10
	GOTO	X		3-12
	IF ... THEN	X	X	3-13
	IF ... GOTO	X	X	3-13
	IF ... GOSUB	X	X	3-13
	INPUT	X	X	3-15
	INPUT FILE	X	X	5-9
	LET	X	X	3-20
RDOS Only	LIBRARY		X	6-4
	LIST		X	6-6
	LOAD		X	6-5
	MAT			Chapter 4
	MAT INPUT	X	X	4-18
	MAT INPUT FILE	X	X	5-14

BASIC KEY WORD	USED AS STATEMENT	COMMAND	PAGE REFERENCE
MAT PRINT	X	X	4-19
MAT PRINT FILE	X	X	5-15
MAT READ	X	X	4-17
MAT READ FILE	X	X	5-12
MAT WRITE FILE	X	X	5-13
NEW	X	X	3-21
NEXT	X		3-7
ON ... GOTO	X		3-22
ON ... THEN	X		3-22
ON ... GOSUB	X		3-22
ON ESC THEN	X		3-22
ON ERR THEN	X		3-22
OPEN FILE	X	X	5-3
PAGE		X	6-15
PRINT	X	X	3-24
PRINT FILE	X	X	5-10
PRINT FILE USING	X	X	5-11
PRINT USING	X	X	3-30
PUNCH		X	6-8
RANDOMIZE	X	X	3-39
READ	X	X	3-40
READ FILE	X	X	5-6
REM	X		3-43
RENAME	X	X	5-22
RENUMBER		X	6-12
RESTORE	X	X	3-44
RETURN	X		3-10
RUN		X	6-9

BASIC KEY WORD	USED AS STATEMENT	COMMAND	PAGE REFERENCE
SAVE	X	X	5-17
SIZE		X	6-I3
STOP	X		3-45
TAB		X	6-15
WHATS		X	6-14
WRITE FILE	X	X	5-8

EXTENDED BASIC USER'S MANUAL

TABLE OF CONTENTS

TABLE OF CONTENTS

CHAPTER 3 - STATEMENTS (Continued)

CHAPTER 4 - MATRICES

CHAPTER 4 - MATRICES (Continued)

CHAPTER 5 - FILE I/O

CHAPTER 6 - KEYBOARD MODE OF OPERATION

CHAPTER 1

WRITING AND RUNNING A BASIC PROGRAM

PREPARING A BASIC PROGRAM

BASIC programs are made up of statements. Each statement is preceded by an integer that can be between 1 and 9999 inclusive. The number given a statement determines the order in which it is executed and listed. For example, two statements to be executed sequentially should be given sequential (but not necessarily consecutive) numbers.

Each statement is on a separate line. The programmer terminates each line at the teletype with a carriage return (RETURN).

Typing errors on the teletype can be corrected by using special control keys:

1. Pressing RUBOUT erases the last character typed. A back arrow (←) is printed, representing the erasure.

2. Pressing SHIFT and L at the same time deletes the entire line. A back slash (\ ) is printed, representing line deletion; BASIC gives a carriage return/line feed. The programmer may then type a new statement.

An example of a BASIC program is given below. The example will be described in detail later in this chapter.

```
100 READ A,B,D,E
110 LET G = A*E-B*D
120 IF G = 0 THEN 180
130 READ C,F
140 LET X = (C*E-B*F)/G
150 LET Y = (A*F-C*D)/G
160 PRINT X, Y
170 GOTO 130
180 PRINT "NO UNIQUE SOLUTION"
190 DATA 1,2,4
200 DATA 2,-7,5
210 DATA 1,3,4,-7
```

In the program, single letters represent program variables. A variable can be a single letter (e.g., Z) or a single letter followed by a single digit (e.g., Z4).

A BASIC program terminates when there are no more program statements, the program is out of data, or when an END or STOP statement is executed.

Most programs can be reduced to three steps:

1. Provide data.
2. Perform calculations.
3. Print answers.

Providing Data

One method to provide data is simply to write equations that contain the necessary valu The BASIC statement used for equations is the assignment (LET) statement; for example

```
20 LET X = 3.141  * 10.2          ← * means multiply
```

The statement will cause 3.141 and 10.2 to be multiplied and the resulting value will be stored in a variable named X.

However, writing values into equations is not very efficient. Programs are generally used for repetitive computations with a large number of different values. Instead of writing values into the equation, BASIC uses variables that can be assigned different values:

```
20 LET X = 3.141  *Y
```

To provide values, BASIC uses two statements, READ and DATA. The READ statement indicates the variables that are to have values and the DATA statement gives the value:

```
10 READ Y
20 LET X  = 3.141  *Y
30 DATA 10.2, 7.3, -56.11, -.003, 34
```

There are now five possible values that Y will assume, which are listed in the DATA statement. Upon execution, the order of the values in the DATA statement

Providing Data (Continued)

is the order in which values are assigned to a variable or to several variables given in READ statements in the program.

Repetitive Computations

In general, statements in BASIC programs execute in the sequential order indicated by their statement numbers. However, if a program is completely sequential, it is not possible to perform repetitive calculations on a number of input values. For example, in the program given under the "Providing Data" section:

```
10 READ Y
20 LET X = 3.141 *Y
30 DATA 10.2, 7.3, -56.11, -.003, 34
```

Five data values are given for Y, but only the first one, 10.2, will be used because the program is completely sequential. It is necessary to insert a statement that will allow the READ and LET statements to be executed more than once:

```
25 GO TO 10
```

The GO TO statement causes a transfer back to statement number 10. The program "reads in" the second value for Y, 7.3, and executes the LET statement again. The program will continue to loop in this way until it runs out of data values for Y.

Note that the GO TO statement was given statement number 25. The reason why most BASIC programs are not numbered consecutively is to allow the programmer to insert any statement he may need without rewriting the entire program.

The GO TO statement is a means of transferring control to a part of a program in a non-sequential manner. There are several ways to do this in BASIC. Another useful statement in transferring control is the IF statement.

```
10 READ Y
15 IF Y < = 0 THEN 10          ← 10 means statement number 10.
20 LET X = 3.141 *Y
25 GO TO 10
30 DATA 10.2, 7.3, -56.11, -.003, 34
```

Repetitive Computations (Continued)

Transfer in an IF statement depends upon whether the expression following the word IF is true or false. The expression is relational and uses the following symbols.

Relational Symbol	Meaning
=	Equal to
<=	Less than or equal to
<	Less than
>=	Greater than or equal to
>	Greater than
<>	Not equal to

The IF statement in the example would prevent the LET statement from being executed when the value of Y is zero or negative. The LET statement would only be executed for positive values of Y; otherwise, control would transfer back to the READ statement to read in another value.

Performing Calculations

The data provided as input must be computed into answers. A simple arithmetic statement of the calculations to be performed must be written in such a way that the BASIC system can recognize the operations required. The statement used is the assignment (LET) statement.

The LET statement is used to assign the result of a calculation to some variable. The calculation to be evaluated, called an expression, appears on the righthand side of the equals sign in the LET statement. The variable to which the expression is assigned appears on the lefthand side of the equals sign. In the previous example, the expression provides for multiplying 3.141 by some value assigned to the variable Y and then assigning the resultant value to the variable X.

The basic arithmetic symbols used in performing calculations are:

Symbol	Operator	Example	Meaning
+	Addition	A+B	Add B to A.
	Plus	+A	Positive A.
-	Subtraction	A-B	Subtract B from A.
	Minus	-A	Negative A.

Symbol	Operator	Example	Meaning
*	Multiplication	A*B	Multiply A by B.
/	Division	A/B	Divide A by B.
↑	Exponentiation	A↑B	Raise A to the power B (A^B).

An expression is made up of elements described in Chapter 2 -- simple variables, numbers, arrays, array elements, and functions, which are linked together by the arithmetic symbols.

Parentheses may be used in arithmetic expressions to enclose subexpressions that are to be treated as entities. A subexpression in parentheses is evaluated first. Within each subexpression, arithmetic operations are performed in the sequence -- exponentiation first, multiplication and division next, addition and subtraction last.

When two operations are of equal precedence, such as addition and subtraction, and there are no parentheses, evaluation proceeds from left to right in an expression.

In addition to arithmetic operations involving the arithmetic symbols, BASIC has a number of standard mathematical functions. These are described in Chapter 2; a few examples are:

SIN (X)	Sine of X, where X must be in radians.
EXP (X)	Natural exponential of X, e^X.
INT (X)	Integer part of X.

An example of an expression to be evaluated and assigned to a variable is:

```
100 LET S = S-(17+SIN(Z)) /3
```

In the example,

1. SIN (Z) is calculated. (Functions evaluated first.)

2. Result of step 1. is added to 17. (Parenthesized subexpression.)

Performing Calculations (Continued)

3. Result of step 2. is divided by 3. (Division has higher precedence than subtraction.)

4. Result of step 3. is subtracted from the value of variable S.

5. Result of step 4. is stored (=) into variable S as its new value.

Printing Output

The program is still not complete. It has data and performs calculations but the user has no way of knowing the results of those calculations. To complete the program, there must be a printout of results.

The PRINT statement is used to print out results of calculations. For example, if the program were written:

```
10 READ Y
20 LET X = 3.141  *Y
22 PRINT X
25 GO TO 10
30 DATA 10.2, 7.3, -56.11, -.003, 34
```

The PRINT statement is made part of the loop, so that a value for X is printed out each time the LET statement is executed. Each value of X will be printed out on a new line. The fact that the item X in the PRINT statement is terminated by a carriage return means 'print next value on a new line'. The output would look as follows:

```
32.0382
22.9293
   .
   .
   .
106.794
```

It is also possible to print out verbatim text using the PRINT statement. The user might want an explanation of each value printed. For example, the program could be written:

Printing Output (Continued)

```
10 READ Y
20 LET X = 3.141 *Y
22 PRINT "FOR Y= ";Y;"  X=  ";X
25 GO TO 10
30 DATA 10.2, 7.3, -56.11, -.003, 34
```

The verbatim text is enclosed in quotation marks. It will be printed out exactly as shown with the same number of blank spaces. The semicolon between the items in the PRINT list means 'print on the same line without spacing'. The output would now be printed as:

```
FOR Y = 10.2   X = 32.0382
FOR Y = 7.3   X = 22.9293
   .
   .
   .
FOR Y = 34     X = 106.794
```

EXAMPLE OF A BASIC PROGRAM

A BASIC program for solving simultaneous linear equations would be:

```
100 READ A,B,D,E                    ← obtain values for constants
110 LET G = A * E - B * D           ← evaluate denominator
120 IF G = 0 THEN 180               ← if G is 0, there is no unique solution
130 READ C,F                        ← obtain remaining constant values
140 LET X = (C*E - B*F)/G           ← solve for X
150 LET Y = (A*F - C*D)/G           ← solve for Y
160 PRINT X,Y                       ← print solutions for X and Y
170 GO TO 130                       ← loop to new values for C and F
180 PRINT "NO UNIQUE SOLUTION"      ← message printed if G = 0
190 DATA 1, 2, 4                    ← data for A, B and D
200 DATA 2, -7, 5                   ← data for E and first values for C and F
210 DATA 1, 3, 4, -7                ← other values for C and F
```

The program solves the following equations:

x+2y=-7	x+2y=1	x+2y=4
4x+2y=5	4x+2y=3	4x+2y=-7

The program prints the paired X-Y values for each set of equations and issues an error message after printing the third pair.

Note that READ and DATA must both be included to provide input data for the BASIC program. The division of values among the DATA statements is arbitrary as long as the values are in correct order. The programmer could have written the DATA statements as:

```
190 DATA 1,2,4,2
200 DATA -7,5
210 DATA 1,3,4,-7
```

or as:

```
190 DATA 1,2,4,2,-7,5,1,3,4,-7
```

The blank spaces used in the BASIC program are only for readability. They could have been omitted. For example, the following statements are equivalent:

```
120 LET G = A * E - B * D

120LETG=A*E-B*D
```

Within quotation marks, however, blanks in text are significant. If the programmer had written statement 180 as:

```
180 PRINT "NOUNIQUESOLUTION"
```

the resultant message (if G had been 0) would have been

```
NOUNIQUESOLUTION
```

WRITING, EDITING, and RUNNING A PROGRAM

Writing and Editing a Program

The user controls the contents of his current program by statement number. In effect, every statement the user types at the terminal must have an initial statement number. This number is matched by BASIC against statement numbers existing in the current program. By this means, the user can delete, insert, or change any given statement as shown below, where n represents a statement number, statement represents a BASIC statement, and the carriage return which the user must press to terminate the statement is represented by the symbol (↵).

User Types	BASIC Response	Example
n ↵	BASIC searches the current program for the statement numbered n. If found, the statement is deleted. If not found, no action is taken.	0300 ↵ Deletes the statement numbered 300.
n_1,n_2 ↵	BASIC searches the current program until statement numbered n_1 is found. BASIC will delete statements n_1 through n_2.	100,500 ↵ BASIC will delete statements numbered 100 to 500 inclusive.
n_1, ↵	Delete statements from the current program starting with statement numbered n_1 and ending with the last (highest numbered) statement within the current program.	55, ↵ Delete statements starting with number 55 until the end of the program.
,n_2 ↵	Delete statement from the current program from the beginning (lowest numbered statement) through n_2 .	,789 ↵ Delete the statements in the program from the beginning of the program through 789.
n statement ↵	BASIC searches the current program for the statement numbered n. If n is not found, statement is inserted in the current program. If n is found, the statement in the current program is replaced by statement.	1200 GO TO 50 ↵ Either inserts statement 1200 or replaces the current statement 1200 with GO TO 50.

Running a Program

When the programmer has written and edited his program, he can cause execution by giving the command:

```
RUN )
```

The program will be run from the lowest numbered statement. If no fatal program errors occur (Appendix A), the BASIC system will print out any output from the program and give the prompting message:

* (space)

when execution is complete.

When a RUN command is given without a statement number argument, the user is effectively running a program for the first time. Arrays must be dimensioned, strings must be given lengths, and variables as yet have no associated values.

The programmer has the option to interrupt his program's execution either by pressing the ESC key at the keyboard or by a programmed STOP statement. When a running program is interrupted in this manner, all current string lengths, array dimensions, and variable values are maintained until the programmer issues another RUN command.

However, the programmer has the option to retain all information. To do so, he resumes execution by giving the command:

RUN n) - where n is the number of some statement in the program at which execution is to resume.

The programmer can resume running at the statement at which running was interrupted or at any other statement within the program. For example, he can resume running with all current values intact at the lowest numbered statement in the current program if he wishes.

CHAPTER 2

ARITHMETIC AND STRING OPERATIONS

ARITHMETIC OPERATIONS

Numbers

BASIC systems can be generated which provide either all single precision or all double precision calculations. In both cases, all forms of unformatted PRINT output provides single precision type significance, while formatted PRINT USING output allows the user to control the number of significant digits output.

Single Precision Calculations

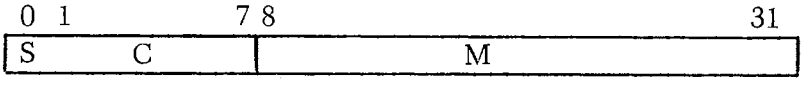
On "PRINT" output, any real or integer number that consists of 6 or less digits is printed out without using exponential form. A real or integer number that requires more than six digits will be printed in 6-digit format, followed by the letter E, followed by an exponent.

Double Precision Calculations

On "PRINT" output, any real or integer number that consists of 8 or less digits is printed out without using exponential form. A real or integer number that requires more than six digits will be printed in 8-digit format, followed by the letter E, followed by an exponent.

Number Represented	Output Format S. P.	Output Format D. P.
2,000,000	2E+6	2000000
20,000,000,000	2E+10	2E+10
108.999	108.999	108.999
.0000256789	2.56789E-5	2.56789E-5
25	25	25
.16	.16	.16
1/16	.0625	.0625

Internally, Extended BASIC stores numbers in a format compatible with other DGC software such as FORTRAN IV and the relocatable assemblers. Single precision floating point numbers are stored in two consecutive 16-bit words of the form:

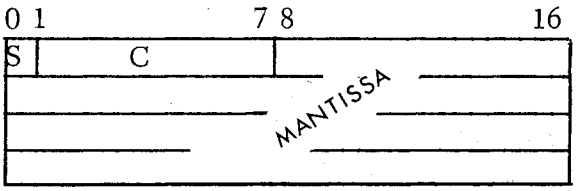


where: S is the sign of the mantissa M. 0 = positive, 1 = negative.

M is the mantissa, considered to be a normalized six digit hexidecimal fraction.

C is the characteristic and is an integer exponent of 16 in excess 64_{10} code.

Double precision floating point numbers add a word of precision to the mantissa, which can be represented as:



The range of floating point numbers is approximately

$5.4 * 10^{-79}$ through $7.2 * 10^{75}$

For additional information on floating point storage, of double precision numbers, see Appendix C of "How to Use the Nova Computers."

Arithmetic Variables

The names of arithmetic variables are either a single letter or a single letter followed by a single digit:

A
A3
Z
Z6

Arithmetic Expressions

Arithmetic expressions can be composed of simple variables, arrays, array elements, and functions, linked together by parentheses and by the arithmetic operators. The arithmetic operators are:

SYMBOL	MEANING	SYMBOL	MEANING
+	Addition	*	Multiplication
+	Plus (positive)	/	Division
-	Subtraction	↑	Raise to the power
-	Minus (negative)		

The order in which operations are evaluated affects the result. In BASIC, unary minus or plus is evaluated first, then exponentation, then multiplication and division, and last addition and subtraction. When two operators are of equal precdence (*and/), evaluation proceeds from left to right.

Arithmetic Expressions (continued)

For example:

Z-A+B*C↑D

1. C↑D is evaluated.
2. B is multiplied by the value from 1.
3. A is subtracted from Z.
4. The value from 2. is added to the value from 3.

The programmer can change the order of evaluation by enclosing subexpressions in parentheses. A parenthesized subexpression is evaluated first. Parentheses can be nested, and the innermost parenthesized operation is always evaluated first.
For example:

Z-((A+B) *C) ↑D

1. A+B is evaluated.
2. The value from 1. is multiplied by C.
3. The value from 2. is raised to the power D.
4. The value from 3. is subtracted from Z.

Some examples of expressions are:

```
L1+1
INT(C/D)/10
(Z(J,J)*Z(I,J))/A*(ABS(I))
J-5
SQR (ABS(X)
```

Arrays

An array represents an ordered set of values. Each member of the set is an array element. Names of arrays are written as a single letter (A-Z). The letter must be unique; it cannot be used as the name of a variable in the program or an error message will result. An attempt to dimension a variable name, such as Z3, will cause an error.

Declaring an Array

Most arrays are declared in a DIM statement, which gives the name of the array and its dimensions. An array can have either one or two dimensions. The lower bound of a dimension is always Ø; the upper bound is given in the DIM statement. There is no limitation on the number of elements in a given array dimension other than restrictions due to available core.

Dimensioning information is enclosed in either parentheses or square brackets immediately following the name of the array in the DIM statement.

```
5 DIM A(15),B[2,3]   ← A is a one-dimensional array of 16 elements (0-15).
                       B is a two-dimensional array of 12 elements.
```

Declaring an Array (Continued)

If the programmer uses an array but does not declare it in a DIM statement, BASIC sets aside 11 elements (0-10) for each dimension. An undeclared one-dimensional array cannot have more than 11 elements. If the programmer does not need 11 or 121 elements for a given array and wishes to conserve space, he should declare the array with the required number of elements. There are no restrictions on the number of elements an array may contain, other than restrictions due to available core.

Array Elements

Each of the elements of an array is identified by the name of the array followed by a parenthesized subscript. (The subscript could, alternatively, be enclosed in square brackets.) The elements of array B[9] would be:

```
B[0], B[1], B[2], ..., B[8], B[9]
```

For a two-dimensional array, the first number gives the number of the row and the second gives the number of the column for each element. The elements of array C[2,3] would be:

```
C(0,0)   C(0,1)   C(0,2)   C(0,3)
C(1,0)   C(1,1)   C(1,2)   C(1,3)
C(2,0)   C(2,1)   C(2,2)   C(2,3)
```

An array element can be referenced with integer or expression subscripts. Any variable or expression that is used for a subscript must evaluate to a datum in the range:

$$\emptyset \leqslant \text{value} \leqslant \text{upper bound declared in DIM}$$

If the variable or expression does not evaluate to an integer, the BASIC system will convert it to fixed form using the INT function, described in the section on functions. For example, some elements of array E(24,5) might be:

Array Elements (Continued)

```
E (I-3, J*K)

E(0, 5)

E (ABS(R), 5)          ← ABS is a function described on page 2-6.
```

If a subscript evaluates to an integer larger than the limit of the dimension for the array, an error message will be printed.

Redimensioning Arrays

It is possible to redimension a previously defined array during execution of a program. Redimensioning does not affect the amount of storage previously defined for the array. It is however, useful for run-time formatting of arrays.

Redimensioning is used primarily to change the subscripting of two-dimensional arrays. Suppose the user originally defines a 3x4 array A.

```
100 DIM A[2, 3]
```

Statement defining A.

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

Row/column assignment of values to elements of array A. A[0, 0] contains 1, A[0, 1] contains 2, ..., A[2, 2] contains 11, and A[2, 3] contains 12.

Later the user might redimension A using the keyboard command DIM. (See Chapter 5.)

```
DIM A[3, 2] ↵
```

Command transposing the dimensions of A.

	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9
3	10	11	12

Row/column assignment of values to elements of A. The values remain the same but the subscripts required to retrieve those values have changed:

Value	New Subscript	Old Subscript
4	A[1, 0]	A[0, 3]
6	A[1, 2]	A[1, 1]
8	A[2, 1]	A[1, 3]
11	A[3, 1]	A[2, 2]

Redimensioning Arrays (Continued)

An array can only be redimensioned so that it has the same or fewer elements. For example, redimensioning a 3x5 array as a 4x4 array will cause an error.

Subscript references outside the defined range of subscripts will cause errors. For example, once array A above is redefined as A(3,2), use of 3 as a column subscript , e.g., A[2,3] will cause an error.

Redimensioning an array to have fewer elements (e.g., redimensioning B[3,5] as B[4,3] or redimensioning C[20] as C[15] merely makes referencing the unused locations impossible. It does not free the locations for other storage.

FUNCTIONS

Some of the examples shown before contained functions. Certain standard mathematical functions are supplied as part of the BASIC system. They are:

SIN(X)	Sine of X where X is in radians.
COS(X)	Cosine of X where X is in radians.
TAN(X)	Tangent of X where X is in radians.
ATN(X)	Arctangent of X where X is in radians. $X = TAN(ATN(x))$ ($-\pi/2 \leq ATN(X) \leq \pi/2$)
LOG(X)	Natural logarithm of X. ($X>0$)
EXP(X)	e^X ($-178 \leq X \leq 175$)
SQR(X)	Square root of X. ($X \geq 0$)
ABS(X)	Absolute value of X.

The arguments of SIN, COS, TAN, ATN, and ABS are confined to the range of acceptable real numbers.

The LOG and SQR functions require positive arguments. A negative or zero argument in the LOG function or a negative argument in the SQR functions will cause the system to respond with an error message.

The argument of the EXP function is confined within the range of values that will generate the largest and smallest acceptable real numbers, i.e., for e^X the range is: $-178 \leq x \leq 175$.

FUNCTIONS

In addition to the standard mathematical functions, the following functions are supplied as part of the BASIC system.

INT(X) The greatest integer not larger than X.

RND(X) A random number between 0 and 1. (There must be a predefined variable or a constant as an argument, though the argument value has no significance.)

SGN(X) The algebraic sign of X. (1 if positive, 0 if 0, and -1 if negative.)

LEN(S) The current length of string variable S.

DET(X) The determinant of the last matrix inverted. (There must be a predefined variable or a constant as an argument, though the argument value has no significance.)

SYS(X) Where X is a digit, 0-10, returning the system information:

0 - the time of day (seconds past 00:00)
1 - the day of the month (1-31)
2 - the month of the year (1-12)
3 - the year in four digits (e.g., 1974)
4 - the terminal line number (-1 if operator's console)
5 - CPU time used in tenths of seconds
6 - I/O usage (number of system calls made)
7 - the error code of the last error
8 - the file number of the file most recently referenced in a file I/O statement
9 - page size
10 - tab size

Note: the values of SYS(7) and SYS(8) are invalid if no run-time error occurred, or if a file has never been referenced in a file I/O statement.

TAB(X) Tabulate to character position X. The TAB function is described fully on page 3-27.

EOF(X) End-of-file function. Returns 1 if file X is at end-of-file, if not it returns a zero. The EOF function is described fully on page 5-6.

FUNCTIONS (Continued)

The INT function yields the largest integer less than or equal to its argument.

```
INT(7.25)      =   7
INT(-7.25)     =  -8
INT(12)        =  12
INT(-.1)       =  -1
INT(X+2)       =  16      If X evaluates to 14.9
INT(1.5)       =   1
```

INT may be used to round a number to the nearest integer. To round the value, add 0.5 to the arguments:

```
INT(X+0.5)
```

The RND function yields a random number having a value in the range: $0 \leq value < 1$. The function requires an argument, although the argument does not affect the resulting random number. The argument can be any constant or previously defined variable.

```
RND(1)      might produce      .654318
RND(0)      might produce      .005461
```

The SGN function generates its result as +1 if the argument is positive, 0 if the argument is 0, and -1 if the argument is negative.

```
SGN(.452)      =   1
SGN(0.00)      =   0
SGN(-24.9)     =  -1
```

The LEN function produces an integer representing the current length of the string variable argument. (Strings are described in the next section of this chapter.) If string variable A$ contains the string TOTAL SALES then:

FUNCTIONS (Continued)

```
LEN(A$)      =   11        ← the space between TOTAL and SALES
                              counts as a character.
LEN(B$)      =   0         ← B$ is unassigned or equals the null string.
```

The DET function yields the determinant of the last matrix inverted. The function requires an argument, although the argument does not affect the resultant determinant. The argument can be any constant or previously defined variable.

If B is a matrix as follows:

```
3    1
7    3
```

(Determinant B = 2)

then:

```
25 MAT C = INV(B)        ← inverse of matrix B becomes matrix C
30 LET X = DET(1)        ← assign determinant of last inverted matrix
35 PRINT X                 to X and print X.
```

The output would be: 2

STRINGS

String Literals

A string literal is written enclosed in quotation marks.

```
"DATA GENERAL CORPORATION"
```

All blank spaces within the quotation marks are significant. The delimiting quotation marks are not printed if the string literal is output.

If the user wishes to insert an ASCII character or a control code into a literal string, he encloses the decimal equivalent of the ASCII code in angle brackets in the form:

```
< n >
```

where: the range of n is $0 \leq n \leq 255_{10}$.

String Literals (Continued)

If the string containing the ASCII character is output, the left and right angle brackets will not be printed.

```
" AN ASCII CONTROL CHARACTER SOUNDS THE TTY BELL, I.E., < 7 > "
```

If the format is not as specified, i.e., if n is outside the range or if n is not enclosed in both left and right brackets, no such error message occurs. The angle bracket and the number will be treated as any other string literal characters.

```
"TEN  < 25  "     ← causing output of TEN < 25
```

String Variables and Expressions

Extended BASIC permits use of string variables as well as literals. String variables are indicated by a dollar sign ($) appended to a letter or letter-digit.

```
R$  or  R2$
```

String variables must be declared in DIM statements. The 'dimension' gives the maximum number of characters the string can contain. There is no restriction on the maximum length of a string, other than restrictions due to available core.

```
DIM A$(25), B3$(215)        ← A$ can contain up to 25 characters.
                              B3$ can contain up to 215 characters.
```

A string variable cannot be assigned more characters than the maximum given in the DIM statement, and if a string variable is 'redimensioned', the maximum number of characters must be less than that given in the original dimensioning statement.

A string expression is a string literal or a string variable. A variable reference to a string may be subscripted or unsubscripted as shown following:

```
A$          ← References the entire string.
A$(2)       ← References the second character through the last character in
              the string inclusive.
A$(I)       ← References position I through the last character in the string
              inclusive.
A$(3,7)     ← References characters occupying positions 3 through 7 inclusive.
A$(I,J)     ← References characters occupying positions I through J inclusive,
              where I and J are evaluated to character positions in the string
              and I ≤ J.
A$(1,1)     ← References only the first character in the string.
```

Thus a subscripted variable lets the programmer reference a subset of one or more characters within a string. String expressions can be used in assignment (LET) statements, PRINT statements, INPUT statements, READ statements, and in relational expressions of IF statements.

```
20 PRINT A$(1,4)                    ← Print first 4 characters of A$.
30 LET B$ = "RESULTS ARE:"          ← Assign string literal to B$.
40 IF A$(I,I) = B$(J,J) GO TO 100   ← If the Ith character of A$ is equal to the
                                      Jth character of B$, transfer to statement
                                      100.
50 INPUT C$, D$(2,2)                ← At the terminal a datum of one or more
                                      characters can be input for C$ and a single
                                      character for D$(2,2).
```

On the righthand side of an assignment statement, string expressions may be concatenated, where the concatenation operator is a comma (,).

```
100 DIM A$(50), B$(50)
110 LET A$="@$2.50 EACH, THE PROFIT MARGIN IS 15.8%."
120 LET B$=A$(1,4), "25", A$(7,35), "1.2%."
```

B$ would contain the following after statement 120 was executed:

```
@$2.25 EACH, THE PROFIT MARGIN IS 11.2%.
```

String Variables and Expressions (Continued)

Following are some string assignment considerations:

```
20 LET A$ = B$          ← contents of A$ are replaced by the contents of B$.
25 LET A$ = " "         ← contents of A$ are replaced by a null string.
30 LET A$ = A$, B$      ← contents of B$ are appended to the current contents of
                          A$.
35 LET A$ = B$, A$      ← produces garbage, since A$ no longer exists at the
                          point at which it is to be appended.
```

When characters are assigned to a string or part of a string, the number of characters to be assigned determines what will be stored. For example:

```
100 LET A$ = "ABCDEF"
110 LET B$ = "1"
120 LET A$(3,3) = B$              ← produces AB1DEF
130 LET A$(3,6) = B$              ← produces AB1
140 LET A$(3) = B$                ← produces AB1
150 LET A$(3) = B$, B$, B$        ← produces AB111
```

When strings are used in the relational expression of an IF statement, the strings are compared character by character on the basis of the ASCII collating sequence until a difference is found. If a character in a given position in one string has a higher ASCII code than the character in that position in the other string, the first string is greater. If the characters in the same positions are identical but one string has more characters than the other, the longer string is the greater of the two. Use of strings in relational expressions is described again in Chapter 3, the IF statement.

```
200 LET A$ = "ABCDEF"
300 LET B$ = "25 ABCDEFG"
             .
             .
             .
310 IF A$ >B$ GOTO 500            ← True. Transfer occurs.
320 IF A$> B$(4) GOTO 500         ← False. No transfer.
330 IF A$(1,4) = B$(4,7) GOTO 500 ← True. Transfer occurs.
```

Some further examples of string manipulations are:

```
100 DIM A$(20), B$(20), D$(20), C$(50)
110 LET A$ = "RESULT IS 25.2%"
120 LET B$ = "$155.24 PER ITEM"
130 LET C$ = A$(1,10), B$(1)
140 IF A$(1,4) = C$(1,4) GO TO 400
150 LET D$(1,8) = B$(13,16), " NO. "
```

When statement 130 is executed, C$ contains: "RESULT IS $155.24 PER ITEM". When statement 150 is executed, D$ contains: "ITEM NO. ", and the relational expression is true if statement number 130 is executed in the sequence shown.

CHAPTER 3

STATEMENTS

As shown in Chapter 1, only a few BASIC statements are needed to write a simple BASIC program. However, the statements available in Extended BASIC allow the user to write programs using more advanced programming techniques as his familiarity with BASIC statements increases. The statements listed below are described in detail on pages following in this chapter. They constitute the statements of Extended BASIC with the following exceptions.

Matrix manipulation statement MAT is described in Chapter 4.

The CALL statement that invokes an external program is described in Appendix B.

Statements that constitute file I/O are described in Chapter 5.

The statements described in this chapter are:

Statement	Usage
BYE	Terminate user/system interaction.
DEF	Define a user function.
DIM	Dimension arrays and string variables.
END	Optional terminator of program.
FOR and NEXT	Set up programming loop.
GOSUB and RETURN	Transfer to and from an internal subprogram.
GO TO	Transfer control to a nonsequential statement.
IF	Conditional transfer to another part of the program.
INPUT	Request data from the teletype.
LET	Assign values to variables.
NEW	Clear current program, close all open channels.

Statement	Usage
ON	Provide a series of possible transfer points.
PRINT	Output data.
PRINT USING	Output data in accordance with "picture" format.
RANDOMIZE	"Reseed" random number generator.
READ and DATA	Input data.
REM	Comment.
RESTORE	Reinitialize the pointer to the start of the data block.
STOP	Halt program execution and switch to keyboard mode.

BYE

Format: BYE

Purpose: The BYE statement terminates interaction between the BASIC system and the user and places the user's terminal into idle mode. The BYE statement does not terminate the system, but idles the user's terminal. The result of the execution of this statement is different depending on which terminal issued the statement, either a user terminal, or the master console.

When a BYE statement (or command) is issued from a user terminal under multi-user RDOS, the system will print certain sign-off information after which only the user terminal which issued the statement is idled. This sign-off information will appear as:

07/09/73	14:23	SIGN-OFF,	00	(terminal number)
07/09/73	14:23	CPU-USED,	1	(time used in tenths of seconds)
07/09/73	14:24	I/O-USED,	13	(number of system calls made)

When a BYE statement (or command) is issued from the master console using any RDOS system, the sign-off information is printed on the terminal as shown above. Then the words:

DIRECTORY SPECIFIER:

are printed on the teletypewriter requesting the master console user to type in a directory name. After the user does so, BASIC is active with a different user directory being used. There is no way to idle a Master Console. If at any time the user wishes to deactivate BASIC, the operator system command #KILL should be issued from the master console (see Appendix C).

In SOS environment, when a BYE statement or command is issued, the terminal it was issued from is idled. No sign-off information is printed.

Example: 100 BYE)

DEF

Format:

DEF FN*a* (*d*) = *expression*

where: *a* is a single letter, A-Z.

d is a dummy arithmetic variable that may appear in expressior

Purpose: To permit a user to define a function that can be referenced several times during a program. The function returns a value to the point of reference.

When a function is referenced, the constant, variable, or arithmetic expression appearing in the reference argument dummy argument *d* in the expression.

In the function definition, *expression* can be any legal arithmetic expression including one containing other user-defined functions. Functions may be nested to a depth of four.

Function definition is limited to those formulas that can be expressed on a single line of text. For longer formulas, sub-routines should be used.

Examples:

```
100 DEF  FNE (X) = EXP (X ↑ 2)        ← definition of the function

200 LET Y = Y*FNE (.1)                ← function reference; argument = .1

300 IF FNE (A+3) > Y THEN 150         ← function reference; argument = A+3
```

```
30 LET P = 3.14159
40 DEF FNR (X) = X*P/180
50 DEF FNS (X) = SIN (FNR(X))         ← Function FNR is nested within
60 DEF FNC (X) = COS(FNR(X))            FNS and FNC.
70 FOR X = 0 TO 45 STEP 5
80 PRINT X, FNS(X), FNC(X)            ← FNS and FNC are referenced with
90 NEXT X                               X having values 0, 5, 10, ..., 45
```

DIM

Format:

```
DIM array1 (dims), string1 (chars), ...
```

Purpose: To give the dimensions of one and two dimensional arrays and to give the maximum number of characters in string variables. The information in this non-executable statement is used to allocate storage.

Arrays are dimensioned as follows:

1. The lower bound is always 0 and does not appear in the DIM statement.

2. The upper bound is given in parentheses or square brackets following the array name.

3. If there are two upper bounds, the bounds are separated by a comma.

String variable names are followed by a single "dimension" in parentheses or square brackets; this gives the upper limit of the number of characters that the string may have.

Arrays and strings may appear in any order in a DIM statement.

Example:

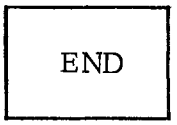
```
2 DIM A(5,6), C(20), X(17), B$(25), C$(30), Y(14,10)

        A is a 6x7-element two dimensional array.
        C is a 21-element one dimensional array.
        X is a 18-element one dimensional array.
        B$ is a string with a maximum of 25 characters. *
        C$ is a string with a maximum of 30 characters. *
        Y is a 15x11-element two dimensional array.
```

*Note: the 0th element of a string is not used and is not included in its length.

END

Format:



Purpose: Many BASIC systems require an END statement as the last program statement or as the terminating statement of a main program that calls one or more subroutines. In Data General's BASIC, all programs terminate at the last logically executed statement in the program (if an END statement or STOP statement is not encountered). However, the implementation allows END statements for compatibility with BASIC programs written for other systems. Multiple END statements may appear in the same program.

FOR AND NEXT

FOR

Format:

FOR control variable = $expression_1$ TO $expression_2$

FOR control variable = $expression_1$ TO $expression_2$ STEP $expression_3$

Purpose: To establish beginning, terminating, and incremental values for control variable, a variable that determines the number of times statements contained in a loop are executed.

The loop consists of statements following the FOR statement up to a NEXT statement that contains the name of control variable. The variable in a FOR statement cannot be subscripted.

$expression_1$ is the first value of the variable.
$expression_2$ is the terminating value of the variable.
$expression_3$ is the increment added to the variable each time the loop is executed. If not given, the increment is +1.

When the NEXT statement containing the variable name is encountered, the loop is executed again. The looping ends, and the statement after NEXT is executed when control variable exceeds the terminating value, $expression_2$.

FOR loops may be nested to a depth of seven. The FOR statement and its terminating NEXT statement must be completely nested. For example:

Legal Nesting

```
FOR X = ...
  FOR Y = ...
    FOR Z = ...
    NEXT Z
  NEXT Y
NEXT X
```

Illegal Nesting

```
FOR X = ...
  FOR Y = ...
NEXT X
NEXT Y
```

NEXT

Format:

```
NEXT control variable
```

Purpose: To terminate the loop beginning with a FOR statement. The control variable contained in the NEXT statement must precisely match the control variable contained in the last uncompleted FOR statement preceding NEXT.

When the FOR statement conditions have been fulfilled, execution continues at the statement following the NEXT statement.

FOR and NEXT Examples

```
 5 FOR X = .1 TO .005 STEP -0.01
10 LET X = X*LOG(X)
20 NEXT X
```

```
10 FOR I = 1 TO 45
20 PRINT 2 ↑ I
30 NEXT I
```

```
10 DIM A(25)
20 READ N
30 FOR I = 1 TO N
40 READ A(I)
50 NEXT I
```

```
100 FOR I = 1 TO 3
120 FOR J = 1 TO 20 STEP I
130 READ B(I, J)
140 NEXT J
150 NEXT I
```

J loop

I loop

FOR and NEXT Examples (Continued)

```
 90 FOR I = 1 TO 9
100 NEXT I
110 PRINT I
RUN ↵
9                                  ← final value of I loop is the terminating value, 9.
120 FOR J = 1 TO 9 STEP 3
130 NEXT J
140 PRINT J
RUN ↵
7                                  ← final value of the J loop is the last value before
                                     the terminating value is exceeded.
```

GOSUB and RETURN

GOSUB

Format:

```
GOSUB statement number
```

Purpose: To transfer control to statement number, the first statement in a subroutine. Control will turn to the next sequential statement after the GOSUB statement when a subroutine RETURN statement is executed. (see below).

A portion of a program is written as a subroutine when it is executed at several different places in the program. A subroutine is an arbitrary set of BASIC statements which contains at least one RETURN statement.

RETURN

Format:

```
RETURN
```

Purpose: To exit a subroutine, returning to the first statement after the GOSUB statement that caused the subroutine to be entered.

A given subroutine may contain a number of RETURN statements when logic might cause the subroutine to terminate at a number of different statements.

Examples of GOSUB and RETURN

In the example following, RETURN causes return to statement number 120 when the subroutine is entered from statement 110; return is to statement 140 when the subroutine is entered from statement 130, etc. Note that there are two RETURNs in the subroutine. Values for X and Y will be printed and return will be made from statement numbered 560 as long as Y is less than 100. Otherwise, statement 540 in the subroutine is executed, and a return is made to the calling program without printing values for X and Y.

```
100 LET X = 5
110 GOSUB 500
120 LET X = 7
130 GOSUB 500
140 LET X = 11
150 GOSUB 500
160 STOP
    :
500 LET Y = 3*X
510 LET Z = 1.2 * EXP(Y)
520 LET Y = SQR (Z+2)
530 IF Y <100 THEN 550
540 RETURN
550 PRINT X, Y
560 RETURN
```

In the example following, the subroutine calls itself.

```
100 LET X = 3.5
200 GOSUB 500          ← call to subroutine from another part of the
    .                    program.
    .
    .
500 LET Y = X ↑ 2
520 PRINT X, Y
530 LET X = X + 2.5
540 IF X > 10.0 GOSUB 500   ← subroutine calls itself.
550 RETURN
    .
    .
    .
```

GOTO

Format:

GO TO statement number

Purpose: To transfer control to a statement that is not in normal sequential order. If control is transferred to an executable statement, that statement and those following will be executed. If control is transferred to a non-executable statement (e.g., DATA), the first executable statement following the one to which transfer was made will be executed.

Examples:

```
190 DATA 19, -5, -2, 5, -6, 10, 10, 60, 20, 5, 50, 10
200 READ X, Y, Z
220 LET A = SQR(X ↑ 2 + Y ↑ 2 - 2*X*Y*FNC(Z))
230 PRINT X, Y, Z, A
240 GO TO 200                ← control will continue to transfer back to statement
                               200 until all values for X, Y, and Z have been read.
```

```
190 DATA 19, -5, -2, 5, -6, 10, 10, 60, 20, 5, 50, 10
200 READ X, Y, Z
220 LET A = SQR (X ↑ 2 + Y ↑ 2 - 2*X*Y*FNC(Z))
230 PRINT X, Y, Z, A
240 GO TO 190                ← same as previous example.
```

IF

Format:

IF relational-expression { GOTO / THEN } statement-number

IF relational-expression GOSUB statement-number

IF relational-expression THEN statement-number

Purpose: To transfer control on the basis of whether relational-expression is true or ~~flase~~. false

The IF-GOTO (THEN) statement format causes control to be passed to the statement whose number appears following GOTO if relational-expression is tyre. If relational-expression is not true, control is passed sequentially to the next statement following the IF statement.

The IF-GOSUB statement format causes control to be passed to the beginning of a subroutine whose statement number appears following GOSUB if relational-expression is true. If relational-expression is not true, control is passed sequentially to the next statement following the IF statement.

The IF-THEN statement format is a generalized form of the IF statement. Any statement, including an IF statement, may follow the THEN.

Relational Expression: A relational expression consists either of two arithmetic expressions and a relational operator or of two string expressions and a relational operator and has the form:

expression1 relational operator expression2

The relational operators are:

Symbol	Meaning	Example
=	Equal	A = B
<	Less than	A < B
<=	Less than or equal	A <= B
>	Greater than	A > B
>=	Greater than or equal	A >= B
<>	Not equal	A <> B

When relational operators are used to compare string expressions, strings are compared character-by-character until a difference is found.

```
IF  "ABCDEF" = "ABCDEFG"  THEN ...

IF "AB" = "AB" THEN ...

IF  "ABC"    > "AB"

IF "BAC "  >   "A"

IF "D"  >  "AAAA"
```

The branch then depends upon the values of the ASCII codes of the pair of characters which first differ. The higher ASCII code value indicates the greater string value. If one string has more characters than the other, (but they have a common prefix), the longer string is the greater of the two. To be equal both strings must have the same characters, in the same order, and be of the same length.

"ABCDEF" = "ABCDEFG" is not true, the second string is the greater of the two, as it contains more characters.

"AB" = "AB" is true since both strings contain the same characters, in the same order, and are of the same length.

"ABC" > "AB" is true since the first string is identical to the second string except for an added character.

"BAC" > "A" is true since B has a higher ASCII code than does A, the first pair of characters to be examined.

"D" > "AAAA" is true since D has a higher ASCII code value than A where the first difference occurred. **

A numeric expression may be used in place of a relational expression following IF. The numeric expression is considered false if it has a value of 0 and is considered true in all other cases.

** Note: The ASCII code value of D is 104; A's ASCII code value is 101.

Examples:

```
100 IF X + Y = 0 THEN 1000
150 IF .01 > = SQR(X) GO TO 410
200 IF A$ < > "YES" GOSUB 650
```

Relational expressions, where values are compared to determine the truth value.

```
101 IF X+Y THEN 1000
151 IF ABS (X) GO TO 410
```

Numeric expressions that evaluate to zero or non-zero. All non-zero values are true. Note that statement 101 is the reverse of statement 100.

```
102 IF X+Y = 0 THEN LET I = 0
```

If X+Y=0 is true, the LET statement is executed and control passes to the next statement in the program; if X+Y=0 is false, the LET statement is not executed and control passes immediately to the next statement in the program.

```
152 IF X THEN IF .01 > = SQR(X) GO TO 410
```

The first IF checks the value of X. If it is zero, control passes to the next statement in the program. If it is not zero, the IF statement following THEN is executed and control passes to the next statement in the program or to the statement 410, depending upon the value of the square root of X.

INPUT

Format:

```
INPUT variable-list
```

where: variable-list can contain arithmetic variables, array elements, string variables, and string literals.

Purpose: To input values for variables and string variables at run time from the user's terminal. The usage of the INPUT statement (without containing a string literal) contains a list of variable names and/or string variable names separated from each other by commas:

```
55 INPUT A,B,C )
60 INPUT P$; )
70 INPUT D, S$, A$(1,4), J )
```

The INPUT statements are all terminated with a carriage return and may be written with a semi-colon preceding the carriage return (i.e., statement number 60). Arithmetic and string variables may be interspersed within the variable list of the INPUT statement (i.e., statement number 70).

When an INPUT statement containing no string literal is executed, the BASIC system types ? at the terminal, requesting data for the variables. (When string literals occur, they replace the question mark prompt. These are discussed later within this section.) An example of passing data to BASIC in response to question mark prompts is:

```
55 INPUT A, B, C )
60 INPUT P$; )
70 INPUT D, S$, A$(1,4), J )
:
RUN )
? 10 ) ? 25 ) ? 33 )
? ABCDEFG ) ? 123 ) ? AMOUNT ) ? ITEM ) ? 456 )
```

The first three items input by the user were separated by carriage returns (10, 25, and 33); and since statement number 55 ended with a carriage return, a carriage return/line feed occurred before executing statement number 60. Input for statement number 60 (INPUT P$) is a string which was also terminated by a carriage return, but, because statement number 60 ended with a semicolon, statement number 70's request for input was continued on the same line as the input received in response to statement number 60.

Purpose: The programmer types the list of data values fpr input immediately following the ?. Each datum is delimited from the next by either a carriage return (as in our example) or by a comma. If a semicolon appears at the end of an INPUT statement a carriage return/line feed will not occur after the last inputted item in response to that statement. But, if there is no semicolon at the end of the INPUT statement, a carriage return/line feed will occur before the next statement is executed. The data list typed in response to the INPUT statement must match the variable list in both type of datum and number of data items. The last data value input by the user in response to a INPUT statement must be terminated by a carriage return.

Character strings in the data list may optionally be enclosed within quotation marks. Character strings may include any characters including digits and angle bracket delimiters enclosing the decimal equivalent of an ASCII character. Since a datum is delimited by commas and carriage returns, a comma or carriage return cannot be a part of the character string unless the character string is not enclosed within quotation marks, leading blanks will be ignored on input:

25.34, THE RESULT IS: , 0	←	The second item in the data list is a string of 14 characters including enclosed blanks.
25.34, " THE RESULT IS:", 0	←	The second item in the data list is a string of 15 characters including enclosed blanks and one leading blank.

Pressing carriage return during typing of the data list does not cause an actual carriage return; it merely signals BASIC that a value has been terminated. If the return is pressed and BASIC does not have enough values to fill the input list, the BASIC system types: ? and sounds the teletype bell. The programmer can then add the needed values to the list.

If the data list contains an error detected by the system, for example, a string value for a numeric value, the BASIC system

INPUT (Continued)

Purpose:

types: \ ? after which the programmer can type the correct value. During the typing of the data list, the programmer may use the line erase (SHIFT L) or character erase (RUBOUT) to correct errors within his list.

It is useful to precede the INPUT statement in the program with a PRINT statement containing a string that will clarify for the user at the teletype which variables the values are requested for.

```
40 PRINT "VALUES FOR A, B, C"     ← string within quotation
                                    marks will be printed on
                                    line before BASIC

50 INPUT A, B, C
```

String literals may be utilized in the INPUT statement to print these prompts; thus the above example could have been written as:

```
50 INPUT "VALUES FOR A, B, C ",A, B, C
```

If an INPUT statement is incorporated into a continuous loop, the programmer can terminate program execution but pressing the ESC key.

INPUT Examples:

```
10 DIM A$(10), B$(10)
20 INPUT Z, A$, X, B$
30 PRINT A$, B$
40 PRINT Z, X
RUN )
? 2.55 ) ? "TEST" ) ? 34 ) ? "ITESTOK" )
TEST            ITESTOK        }     BASIC Printout; data list
2.55            34             }     separated by carriage returns.
```

INPUT (Continued)

```
10 DIM A$(10), B$(10)
20 INPUT Z, A$, X, B$
30 PRINT A$, B$
40 PRINT Z, X
RUN )
? 2.55 , TEST , 34 , "ITESTOK" )
TEST          ITESTOK
2.55          34
```

BASIC Printout; data list separated by commas.

```
:
RUN )
? 2.55 ? "TEST" ? "STRING" \ ? 34 , "ITESTOK " )
TEST          ITESTOK
2.55          34
```

BASIC Printout

(Note: the third item in the data list , the user tried to input a string (STRING) in the place of a required number. Basic types \ ? . The user merely retyped the correct numeric value and the final string value.)

```
10 DIM A$(10), B$(10)
20 INPUT Z, X, A$, B$ ;
30 PRINT A$, B$
40 PRINT Z, X
RUN )
? 2.55 , STRING \ ? 34 , "STRING", TEST STRING                TEST
2.55          34
```

BASIC Printout

BASIC Printout

(Note: the second item in the data list is a string in place of the required number. BASIC types \ ? after which the programmer typed the correct numeric value.)

```
20 INPUT "VALUES OF X, Y, Z", X, Y, Z
       :
RUN )
VALUES OF X, Y, Z 2.5 , -44.1 , .5
```

← Informs user as to requested values.

← user supplies 3 values.

```
10 PRINT "VALUES OF X, Y, Z"
20 INPUT X
30 INPUT Y
40 INPUT Z
       :
RUN )
VALUES OF X, Y, Z
? 2.5 , -44.1 , .5
```

← Each INPUT statement contains one variable in list.

← Error message will result since only one value (X) is expected.

```
10 PRINT "VALUES OF X, Y, Z";
20 INPUT X
30 INPUT Y
40 INPUT Z
       :
RUN )
VALUES OF X, Y, Z ? 2.5 )
? -44.5 )
? .5 )
```

← Data list separated by carriage return.

```
0005 DIM G$(8)
0010 PRINT "FAHRENHEIT";
0020 INPUT F, G$;
0030 LET C = (F-32)*5/9
0040 PRINT "CENTIGRADE = " C, G$
0050 PRINT
0060 GOTO 0010
RUN )
FAHRENHEIT ? 32 , " FOR F = 32"      CENTIGRADE = 0 FOR F = 32
FAHRENHEIT ? 50 , " FOR F = 50"      CENTIGRADE = 10 FOR F = 50
FAHRENHEIT ? (user presses ESC)
STOP AT 0020
```

ASSIGNMENT STATEMENT (LET)

Format:

```
variable = expression

LET variable = expression
```

Purpose: To evaluate expression and assign the value to variable.

String expressions may be assigned to string variables, and arithmetic expressions may be assigned to arithmetic variables.

The variable may be subscripted.

Use of the mnemonic LET is optional.

Examples:

```
10 LET A = 4.17+G

40 X = X+Y↑3.5

80 LET W7 = ((W-X)↑4.3)*SQR(Z-A)/B

90 J(I, INT(K/10)) = COS(FNA(K+I))
```

```
100 DIM A$(10), B$(10), C$(20), D$(10)
140 LET A$ = "NOW"
150 B$ = "TIME"
160 C$ = A$," IS THE ",B$                    ← string concatenation
170 LET D$ = A$(1,2), B$(1,1), "␣", B$(3,4)      ← string concatenation

            A$ contains NOW
            B$ contains TIME
            C$ contains NOW IS THE TIME
            D$ contains NOT ME
```

NEW

Format:

```
NEW
```

Purpose: The NEW statement clears all currently loaded statements and variables, and closes any open channels. It is usual to give this statement before beginning input processing of a new current program. This statement can be the last executable statement within the current program; thereby, after executing the program, and printing out all results, the program will be immediately cleared from memory. This statement, in conjunction with ON ERR or ON ESC statements, can be used to prevent unauthorized reading of a program. (More elaborate techniques can be used to check a user password upon detection of an ESC or ERR to decide if read access should be permitted.)

Examples:

```
100 READ A
110 LET C = A * 23
120 PRINT C;
125 GOTO 100
130 NEW
135 DATA 1,2,3,4,5,6
RUN )
23  46  69  92  115  138
*
LIST )                          ;MEMORY HAS BEEN CLEARED.
```

```
5  ON ESC THEN NEW
6  ON ERR THEN NEW
10 - - -
. - - -
. - - -
. - - -
```

ON

Format:

```
ON expression GO TO statement number list

ON expression THEN statement number list

ON expression GOSUB statement number list

ON ERR THEN statement

ON ESC THEN statement
```

Purpose: The ON statement as specified in the first three statement formats, is written for the purpose of providing several possible transfer points. The statement to which transfer will be made depends upon the evaluation of expression. The value of expression must correspond to the sequence number within the list of one of the statement numbers.

If expression does not evaluate to an integer, it is truncated to an integer by the INT function.

If expression evaluates to an integer that is greater than the sequence number of the last statement number in the list or that is less than or equal to zero, the ON statement is ignored and control passes to the next statement.

ON-GOTO and ON-THEN are equivalent formats. ON-GOSUB must contain a list of statement numbers, each of which represents the start of a subroutine within the program.

ON ERR THEN will, when an error results from the execution of the program, execute statement. statement may be any legal Extended BASIC statement. ON ESC THEN will, when the user hits the ESC key, execute statement; statement may be any legal statement. If the ON ESC statement does not execute a GOTO, then control will eventually return to the user's program at the last line executed before the escape was typed.

Examples:

```
ON M-5 GOTO 500, 75, 1000
```

If M-5 does not evaluate to 1, 2, or 3, the statement is ignored. If M-5 evaluates to 1, transfer is made to statement 500; if the expression evaluates to 2, transfer

is made to statement 75, and if the value is 3, transfer is made to statement 1000.

```
21 ON X GOSUB 1000,2000,3000,4000   ;X must be 1,2,3, or 4 to effect transfer.
31 ON Y*I THEN 100,50,90,550,80,75  ;Y*I must have a value of 1 through 6.
55 ON ERR THEN GOSUB 160            ;Transfer to an error subroutine.
78 ON ESC THEN GOTO 100             ;Transfer to statement 100 when ESC occurs.
```

Note: Caution must be exercised with use of the ON ESC THEN statement. Usually, if the user presses ESC, it indicates to the system that whatever is occurring must stop. If a program is executing, execution will cease; if data is being output, output will cease. But, with the use of the ON ESC THEN statement, instead of halting, the system transfers control to the statement appearing within the ON ESC THEN statement.

As an example, assume that the user includes the statement 100 ON ESC THEN PRINT X, Y, Z in his program. When the user presses ESC during program execution, program execution does not cease; instead, control passes to statement 100 and the values of X, Y, and Z will be printed. Control continues in the program as if line 100 were never executed, that is, in the example below control continues at line 141 if line 140 were the last to complete before the escape is processed.

```
100 ON ESC THEN PRINT X, Y, Z
    .
    .
140 PRINT X
141 Y = Z
    .
    .
```

In order to stop the execution of a program when using the ON ESC THEN statement, a statement must appear in the program after the ON ESC THEN statement which will instruct the system to stop. (For example, a STOP statement or an ON ESC THEN STOP statement). The latter will also restore normal use of the ESC key.

In the example following, execution of one of the RETURN statements will return control not to statement 30 but to the statement following the last to complete execution after the escape key was struck.

```
10 ON ESC THEN GOSUB 500
20 DIM X(1000)
30 FOR I = 1 TO 1000
40 X(I) = A*I 2+B*I+C
50 NEXT I
60 STOP
    .
    .
500 PRINT I,X(I)
510 INPUT "CONTINUE (0) OR NEW INPUTS (1)", D
520 IF D = 0 THEN RETURN
530 INPUT "NEW VALUES FOR A,B,C", A,B,C
540 RETURN
```

PRINT or ;

Format:

```
PRINT expression list
    ; expression list
```

where: expression list is a list of numeric variables, subscripted variables, arithmetic expressions, string literals, and string variables.

PRINT and; are equivalent statement forms.

Purpose: To output current values for any expressions and variables appearing in the expression list of the PRINT statement, and to output verbatim text for any string literals in the expression list.

Output Formatting: The PRINT statement allows the user to either control output formatting or to accept default formatting.

Number Representation

Any real or integer number which can be represented as six digits ** and a decimal point is printed out without using exponential form. A minus sign is printed if the number is negative; a space is left before a positive number. All other numbers are printed in the format:

```
[ -] n[.] nnnnnE+ e[e]
```

where: n is a digit

E indicates exponentiation.

e is a digit of the exponent.

[ ] square brackets indicate optional parts of the number.

If the number is positive, the sign position is left blank.

Number	Printed Output	Printed Output D. P.
.00000002	2E-8	.00000002
-.0002	-.0002	-.0002
200	200	200
-200.002	-200.002	-200.002
2,000,000	2E+6	2000000
-20,000,000,000	-2E+10	-2E+10

** Eight digits for double precision, see Appendix F, and pages 2-1, 2-2.

PRINT or ; (Continued)

Zone Spacing of Output (,)

The terminal line is divided into zones. By default, each zone is set to 14 spaces, which is a typical spacing for a 72-character teletypewriter:

```
0   14   28   42   56 (14-space zone, allowing 5 columns of data to
                       be output on a 72-character teletypewriter.)
```

TAB = ...

The user can set the number of spaces per zone at the beginning of a console session, or at any time during the console session by means of the keyboard command, TAB, which is described on page 6-16. The zone may be set in the range from one character up to the limit of the page width of the terminal device.

A comma between items in the expression list of the PRINT statement indicates "space to the next zone." If the last print zone has been filled, the next value is printed in the first print zone of the next line.

```
10 LET X = 5
50 PRINT X, (X*2) ↑ 6, X*2,
60 PRINT X ↑ 4, X-25, (X*2) ↑ 8, X-100
```

(Assume a 14-space zone. Note terminating comma on first PRINT statement controls the output of the first value of the next PRINT statement.)

```
0             14            28            42            56
↓             ↓             ↓             ↓             ↓
 5             1E+6          10            625          -20
 1E+8         -95
```

When an output value is longer than a single zone, for example, a long character string, the teletype is spaced to the next free zone to print the next value.

```
10 LET X = 25
20 PRINT "THE SQUARE ROOT OF X IS:", SQR(X)

0                14            28      ← position
↓                ↓             ↓
THE SQUARE ROOT OF X IS         5      ← value
```

Compact Spacing of Output

The user can obtain more compact output by use of the semicolon between list items. It inhibits spacing to a print zone, leaving only a single space between values output for list items. Note that like the comma, a semicolon at the end of a PRINT statemen will determine the position of the first value of the next PRINT statement.

```
10 LET X = 5
20 PRINT X; (X*2) ↑ 6; X*2; (X*2) ↑ 4;
30 PRINT X-25; (X*2) ↑ 8; X-100

0 1 2 3 4       10   14      20    25     30          ← position
↓ | | ↓       ↓    ↓       ↓     ↓      ↓
 5  | 1E+6     10   10000   -20   1E+8   -95         ← value
```

Spacing to the Next Line

If there is no comma or semicolon terminating the last item of the list of a PRINT statement, the edited output will be followed by a line feed/carriage return so that the next PRINT statement will begin printing on the next line.

```
10 LET X = 5
20 PRINT X, (X*2) ↑ 6
30 PRINT X*2
40 PRINT X-25; (X*2) ↑ 8
50 PRINT X-100

0      5          15              ← position
↓      ↓          ↓
 5                 1E+6            ← value
 10
-20    1E+8
-95
```

Output Formatting: (Continued)

Tabulation

It is possible to tabulate to a particular character position to output a value using the TAB function:

```
TAB (expression)
```

where: expression evaluates to an integer representing the character position to begin output of the next list item following the TAB function. The TAB function only tabulates to the given position if the carriage is not set beyond the desired character position. If the expression in the TAB function evaluates to a number greater than the carriage length, the expression is reduced modulo carriage length. If expression evaluates to zero, TAB(0) causes a carraige return.

```
10 DATA 5, -7, 9, -11
20 READ A, B, C, D
30 PRINT TAB(5); A; TAB(10); B; TAB(15); C; TAB(20); D


0     5     10    15    20          ← position
↓     ↓     ↓     ↓     ↓
      5     -7    9     -11         ← value
```

```
10 DIM B[4]
20 DATA 5, -7, 9, -11
30 FOR I = 0 TO 3
40 READ B[I]
50 ; TAB(5); B[I];
70 NEXT I

0     5  7   10  13
      ↓  ↓   ↓   ↓
      5  -7   9  -11          ← Note that the TAB function only affects the first array value, since the TAB function is only effective at its first encounter.
```

PRINT or ; (Continued)

Output Formatting : (Continued)

Following are a few additional examples of output printing:

```
10 FOR I = 1 TO 10
20 ;I,                                   ← If zone = 15 spaces.
30 NEXT I

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

```
10 FOR I = 1 TO 10
20 PRINT I                               ← Carriage return delimiter.
30 NEXT I

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

```
10 FOR I = 1 TO 10
20 PRINT I,                              ← If zone = 14 spaces.
30 NEXT I

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

PRINT OR ; (Continued)

String Variables

String expressions may be printed by the use of the PRINT statement.

```
10 DIM A$(25), B$(25)
20 LET A$ = "IF X IS - "
30 LET B$ = " THEN X SQUARED IS - "
40 READ X
50 LET C = X ↑ 2
60 PRINT A$;X;B$;C
70 GOTO 40
80 DATA 5, 10, 15, 20, 25
RUN )
IF X IS -  5 THEN X SQUARED IS -  25
IF X IS -  10 THEN X SQUARED IS -  100
IF X IS -  15 THEN X SQUARED IS -  225
IF X IS -  20 THEN X SQUARED IS -  400
IF X IS -  25 THEN X SQUARED IS -  625
```

```
10 DIM A$(25),B$(25)
20 LET A$ = "ABCDEF"
30 LET B$ = "GHIJKLM"
40 PRINT A$,B$
RUN )
ABCDEFGHIJKLM
```

PRINT USING

Format:

```
PRINT USING format-string, expression-list
```

where: expression-list is a list of numeric variables, subscripted variables, arithmetic expressions, string literals, and string variables.

format-string may be a string literal or a previously defined string variable, specifying formats of the fields in which the value of each of the expressions in the list is to be output.

Purpose: To output current values for any expressions and variables appearing in the expression list of the PRINT USING statement in accordance with the field formats specified by format-string.

Formatting Rules and Examples:

1. Since the output field formats are specified by format-string, all formatting conventions used in the PRINT statement (TAB function, comma, and semicolon) are ignored within expression-list. However, any comma or semicolon terminating the expression-list will follow PRINT statement conventions.

2. Within string-expression, a number of format fields and string literals for output may appear. One or more format fields may be given in format-string; a format field is made up of combinations of the following characters:

   + - # , . $

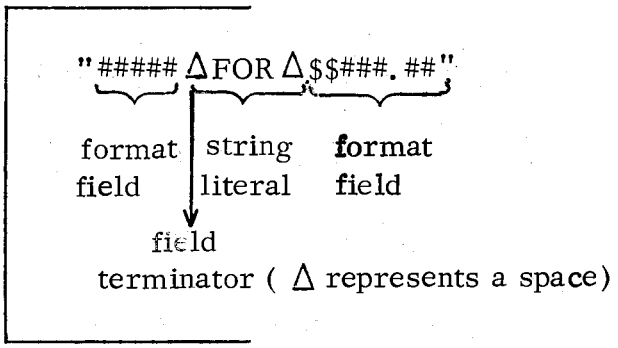
   The special format field characters may appear as part of string literals within format - string as well as in format fields. BASIC differentiates format fields from string literals by the characters that appear in format fields. For example:

```
"TWO FOR $1.25"      ← $1.25 are characters of a string literal.
"TWO FOR $$$.##"     ← $$$.## is a field format (a $ followed by
                        an appropriate field format character --
                        another $ in this case.)
"ANSWER IS -85"      ← -85 are characters of a string literal.
"ANSWER IS -###"     ← -### is a field format (a - followed by an
                        appropriate field format character --
                        a # in this case.)
```

3. The format fields may be specified in format-string by referencing a previously defined string variable; for example:

```
 5 DIM S$(10)
10 LET S$ = "##.## "
20 PRINT USING S$, 1.5, 2
```

4. A format field is terminated by the appearance of the first non-format field character.



5. String literals may appear in the expression list of the PRINT USING statement and will be superimposed on a field format in the following manner:

   a. Each character of the string replaces a single format field character, which may be any one of the format field characters.

   b. Strings are left justified in the format field, with a fill of spaces if necessary.

   c. If the number of characters in the string is greater than the number of characters in the field format, the string will be truncated.

```
PRINT USING "###,###.##","TEST","CHARACTER","SEVENTY-FIVE"

TESTΔΔΔΔΔΔCHARACTERΔSEVENTY-FI

(where: each Δ represents a space)
```

PRINT USING (Continued)

5. When there are more expressions in the expression list than field formats in string-expression, the existing formats will be used repetitively.

```
"#### @$###.## PER ###"

The first, fourth, seventh, etc., expressions in the list will be formatted
using the field format ####.

The second, fifth, eighth, etc., expressions in the list will be formatted
using the field format $###.##.

The third, sixth, ninth, etc., expressions in the list will be formatted
using the field format ###.
The embedded blanks, @ sign, and PER are string literals.
```

```
10 PRINT USING "A[#]△=△##.#", I, A[I]

A[1] = 17.9                    ← possible output; number of expressions in the list
                                 equals the number of field formats.

5 PRINT USING "###.##△", I, A, B

△△1.00△△17.90△△25.77△          ← possible output; number of expressions
                                 in the list exceeds the number
                                 of field formats.
```

PRINT USING (Continued)

6. The special characters:

+ - # , . $

are used in formatting numeric output as follows:

a. Digit Representation (#)

For each # in the field format, a digit (0-9) is substituted.

Field Format	Datum	Representation	Remarks
#####	25	△△△25	Right justify digits in field with leading blanks.
#####	-30	△△△30	Signs and other non-digits are ignored.
#####	1.95	△△△△2	Only integers are represented; the number is rounded to an integer.
#####	598745	*****	If the datum is too large for the field, all asterisks will be output.

b. Decimal Point (.)

The decimal position indicator (.) places a decimal point within the string of digits in the fixed character position in which it appears. Digit positions (#) following the decimal point will be filled; no blank spaces are left in these digit positions. Where the datum contains more fractional digits than the field format decimal indicator allows, the fraction will be rounded to the limits of the field. When the datum contains less digits to the right of the decimal than ther are # positions in the format string, zeroes are output to fill out the format field.

Field Format	Datum	Representation	Remarks
#####.##	20	△△△20.00	Fractional positions are filled with zeroes.
#####.##	29.347	△△△29.35	Rounding occurs on fractions.
	0.079	△△△△0.08	
#####.##	789012.34	********	When the datum has too many significant digits to the left of a decimal, a field of all asterisks, including the decimal position, is output.

c. Fixed Sign (+ or -)

A fixed sign character appears as a single plus (+) or minus (-) sign in either the first character position in the format field or the last character position in the format field. The signs have the following effect:

+ prints a + in the given field position if the datum is positive and prints a - in the given field position if the datum is negative.

- prints a - in the given field position if the datum is negative and leaves a blank space in that field position if the datum is positive.

When a fixed sign is used, any leading zeroes appearing in the datum will be replaced by blanks, except for a single leading zero immediately preceding a decimal point.

Field Format	Datum	Representation	Remarks
+##.##	20.5	+20.50	
+##.##	1.01	+△1.01	Blanks precede the number.
+##.##	-1.236	-△1.24	
+##.##	-234.0	******	
###.##-	20.5	△20.50△	
###.##-	000.01	△△0.01△	The last leading zero before the decimal point is not suppressed.
###.##-	-1.236	△△1.24-	
###.##-	-234.0	234.00-	

d. Floating Sign (++ ... or -- ...)

A floating sign appears as the first two (or more) signs in the field format. Floating positive (++) outputs either a plus or minus sign immediately preceding the datum; floating negative (--) outputs either a blank space or minus sign immediately preceding the datum.

Positions occupied in the field format by the second sign and any additional signs can be used for numeric positions in the datum without field overflow occurring as shown in some of the examples.

Field Format	Datum	Representation	Remarks
---.##	-20	-20.00	Second and third signs are treated as digit positions (#) on output.
---.##	-200	******	
---.##	2	△△2.00	

Either a floating sign or a floating $ sign (see section f.) can be used but not both.

e. Fixed $ Sign ($)

A fixed $ sign appears as either the first character or second character in the string, causing a $ to be placed in that character position. The $ may appear as the second character if it is preceded by a fixed sign. A fixed $ causes leading zeroes in the datum to be replaced by blanks.

Field Format	Datum	Representation
-$###.##	30.512	△$△30.51
$###.##+	-30.512	$△30.51-

f. Floating $ Sign ($$ ...)

A floating $ consists of at least two $ characters beginning at either the first or second character position in the string, and causing a $ sign to be placed in the character position immediately preceding the first digit.

If the floating $ sign begins in the second character position of the string, it is preceded by a fixed sign (+ or -).

Only one floating character (sign or $) is permitted in a given field.

Field Format	Datum	Representation	Remarks
+$$$$#.##	13.20	+△△$13.20	Extra $ signs may be replaced by digits as with floating + and - signs.
$$##.##-	-1.0	△$01.00-	Leading zeroes are not suppressed in the # part of the field.

g. Separator (,)

The separator (,) places a comma within a string of digits in the fixed character position in which it appears in the field format. However, if the comma would be output in a field of suppressed leading zeroes (blanks), a blank space will be output in the comma's position.

Field Format	Datum	Representation	Remarks
+$#,###.##	30.6	+$△△△30.60	Space printed for comma.
+$#,###.##	2000	+$2,000.00	
++##,###	00033	△+00,033	Comma is printed when leading zeroes are not suppressed.

h. Exponent Indicator (↑)

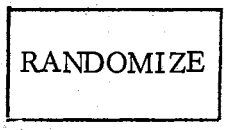
Four consecutive arrows (↑↑↑↑) are required to indicate an exponent field and will be filled by E±nn, where each n is a digit.

If the exponent field does not contain four up arrows, a run-time error will result.

Field Format	Datum	Representation
+##.##↑↑↑↑	170.35	+17.03E+01
+##.##↑↑↑↑	-.2	-20.00E-02
++##.##↑↑↑↑	6002.35	+600.24E+01

RANDOMIZE

Format:



Purpose: The RANDOMIZE statement is used when the programmer wishes the random number generator to calculate different random numbers each time it is used. When the RANDOMIZE statement is executed, it will cause the RND function to choose a random starting value, so that a program which is run twice will give different results.

Example:

```
    .
    .
    .
 80 FOR L = 1 TO 20
 90 LET X(L) = L*RND(0)
100 NEXT L
    .
    .
    .
199 NEW
200 RANDOMIZE
210 FOR I = 1 TO 20
220 PRINT RND(0)
230 NEXT I
    .
    .
    .
```

Within the program, the random number generator is used in calculations.

When statements 210 through 230 are executed the random number generator will generate different number than those used in the calculations performed by statement numbers 80 through 100.

Note: If the user should wish to use the same random numbers within his program or within two different programs, the user should not use the RANDOMIZE statement. By merely issuing the RUN command, the BASIC system will reinitialize the random number generator to a fixed start point.

READ

Format:

```
READ variable list
```

where: variable list can contain arithmetic variables, array elements and string variables.

Purpose: To read values from the data block into variables listed in the READ statement.

The order in which variables appear in the READ statement is the order in which values for the variables are retrieved from the data block.

Values appearing in all DATA statements in a program are stored, before a program is executed, into a single data block for use as values of variables in the READ statement.

Normally, READ statements are placed in the program at those points at which data is to be manipulated, while DATA statements may be placed anywhere.

A pointer is moved to each consecutive value in the data block as values are retrieved for variables in READ statements. If the number of variables in the READ statement exceeds the number of values in the data block, an "out of data" error message is printed. The RESTORE statement can be used to reset the pointer to the beginning of the data block.

The type of variable in the list of the READ statement must match the corresponding value in the list of the DATA statement. An attempt to read an arithmetic value with a string variable will result in an error message.

DATA

Format:

```
DATA constant list
```

Purpose: To provide values to be read into variables appearing in READ statements.

Numbers and string literals may appear in DATA statements.

Each number or string literal is separated from the next datum by a comma.

DATA is a non-executable statement. The values appearing in the DATA statement or statements are read into a single data block before the program is run. The values in the data block are ordered from the data statements by line number, and within a data statement from left to right.

Examples of READ and DATA

```
150 READ X, Y, Z
     .
     .
     .
200 READ A
     .
     .
     .
250 FOR I = 0 TO 10
255 READ B(I)
260 NEXT I
     .
     .
     .
400 DATA 4.2, 7.5, 25.1, -1, .1, .01, .001, .0001
450 DATA .2, .02, .002, .0002, .015, .025, .3, .03, .003
```

The first three data values are read for X, Y, and Z respectively. The value -1 is read into A. The next eleven values, .1 through .3, are read into the eleven elements of array B. This ordering holds true even if statement 450 is entered before statement 400.

Examples of READ and DATA (Continued)

```
    .
    .
100 READ A, B, C
    .
    .
300 GO TO 100
    .
    .
500 DATA 1, 10, .333
510 DATA -1, 1, .555
520 DATA 0, -1, .1
```

Each series of data values, contained in the three DATA statements will, in turn, be read into variables A, B, C.

```
    .
    .
    .
 50 READ A, A$, B, B$, C
    .
    .
    .
550 DATA 1, "TIME:", 10.5, "TEMPERATURE:", 43
```

Numeric values in the DATA statement correspond to numeric variables in the READ statement; string constants correspond to string variables.

REM

Format:

```
REM text comment
```

Purpose: To insert explanatory comments within a program. The text following REM is stored before the program is run and is reproduced exactly as it appears in the statement when a listing of the program is printed. Although the REM statement is non-executable, note that storage space is required for the text.

Example:

```
100 REM PROGRAM TO FIND COMPOUND INTEREST
    .
    .
    .
```

RESTORE

Format:

Purpose: To permit reuse of the data block. RESTORE sets the data block pointer to the first value in the data block. The next READ statement following execution of a RESTORE statement will begin reading values from the start of the data block.

Example:

```
 20 FOR K = 0 TO 10
 30 READ B[K]          ← Data values 1, 2, ..., 11 read into elements of
 40 NEXT K                array B.
 50 RESTORE
 60 READ X, Y, Z       ←  Data values 1, 2, 3 read into X, Y, Z respectively.
 70 RESTORE
     .
     .
     .
200 READ ---           ← Next READ; values start at 1 again.
500 DATA 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13
```

STOP

Format:

Purpose: To halt the execution of a program at some point returning the user to keyboard mode. When STOP is encountered, the BASIC system will cease execution and type the message:

STOP AT xxxx

where: xxxx is the line number of the STOP statement.

The system will wait for a keyboard command.

Example:

```
 80 FOR K = 0 TO M-1
 90 LET X = B + K*P
100 IF X-M*INT(X/M) = A THEN 130
105 NEXT K
110 PRINT "ERROR"
120 STOP              ← Stop program if error occurs.
130 LET P = P*M
      .
      .
      .
```

CHAPTER 4

MATRICES

MATRIX STATEMENTS

A special set of statements allows users to manipulate two-dimensional arrays as matrices. Matrix statements begin with the word MAT. Following is a list of the matrix statements available in BASIC.

Statement	Meaning
MAT READ A, B, ... MAT READ A (3,4), B(5,5), ...	Read DATA values for previously dimensioned arrays or for arrays having the dimensions given in the statement.
MAT INPUT A, B, ... MAT INPUT A (2,4), B(3,3), ...	Input values from keyboard for previously dimensioned arrays or for arrays having the dimensions given in the statement.
MAT PRINT A, B, ...	Print current values of previously dimensioned arrays. PRINT delimiters comma (,) and semicolon (;) may be used in MAT PRINT statements.
MAT A = B	Matrix A is dimensioned to the dimensions of matrix B and the values of B are stored into A.
MAT A = B + C MAT A = B - C	Matrix add or subtract B and C. Dimension A to the dimensions of the resulting matrix expression and store the values into A. The dimensions of B and C must be identical.
MAT A = B * C	Matrix multiply B and C. Dimension A to the dimensions of the resulting expression and store the values into A. The dimensions of A, B, and C must be compatible as defined later in the description of matrix multiplication.
Illegal Statements: MAT A = A*B MAT A = B*A	
MAT A = (expression) * B	Scalar multiply matrix B by the parenthesized expression. Dimension A to the dimensions of the resulting expression and store the values into A.

MATRIX STATEMENTS (Continued)

Statements	Meaning
MAT A = INV(B)	Invert matrix B. Dimension A to the dimensions of the resulting expression and store the value of the inverse matrix into A. B must be a square matrix.
MAT A = TRN (B) Illegal Statement: MAT A = TRN (A)	Transpose matrix B. Dimension A to the dimensions of the resulting expression and store the values into A. A and B must be two distinct arrays.
MAT A = ZER MAT A = ZER (3,4) MAT A = ZER (10)	Store zero matrix in A. A can be dimensioned in the statement. A single dimension produces a one-column matrix.
MAT A = CON MAT A = CON (5,6) MAT A = CON (8)	Store matrix of all ones in A. A may be dimensioned in the statement. A single dimension produces a one-column matrix.
MAT A = IDN MAT A = IDN (2,5) MAT A = IDN (5)	Store the identity matrix in A. A can be dimensioned in the statement. A single dimension produces a one-column matrix.

The matrix file I/O statements are described in Chapter 5. They are:

MAT READ FILE [n], A, B, ... MAT READ FILE [n], A(3,4),B(5,5)...	Read binary values from file n for previously dimensioned arrays or for arrays having the dimensions given in the statement.
MAT INPUT FILE [n], A, B, ... MAT INPUT FILE [n], A(3,4),B(5,5)...	Read ASCII values from file n for previously dimensioned arrays or for arrays having the dimensions given in the statement.
MAT PRINT FILE [n], A, B, ...	Output to file n in ASCII format current values of previously dimensioned arrays.
MAT WRITE FILE [n], A, B, ...	Output to file n in binary format current values of previously dimensioned arrays.

MATRIX SUBSCRIPTS

Caution should be observed when manipulating arrays as matrices. Matrices do not have zero subscripts. That portion of a previously declared array that has zero subscripts will be ignored. For example the following coding samples will produce identical printouts:

```
0010 DIM A[4,4]
0020 FOR I = 0 TO 4          ← values stored in zero-subscript elements
0030 FOR J = 0 TO 4
0040 READ A[I,J]
0050 NEXT J
0060 NEXT I
0070 MAT PRINT A
0080 DATA 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5
```

25 values for array A

```
0010 DIM A[4,4]
0020 FOR I = 1 TO 4          ← no values stored in zero-subscript elements
0030 FOR J = 1 TO 4
0040 READ A[I,J]
0050 NEXT J
0060 NEXT I
0070 MAT PRINT A
0080 DATA 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, ← 16 values for matrix A
```

In the first case data is stored into all locations of array A; in the second example data is stored only into those locations with non-zero subscripts. When the MAT PRINT statement is executed the following will be the result in both cases:

2	2	2	2	MATRIX A
3	3	3	3	
4	4	4	4	
5	5	5	5	

MATRIX SUBSCRIPTS (Continued)

Like all BASIC arrays, matrix elements are stored by row in ascending locations in memory. A matrix dimensioned as

```
10 MAT READ (3, 3)
```

← first dimension represents rows and second dimension represents columns.

will be stored as:

rows ↓	0	1	2	3	← columns
0	A[0, 0]	A[0, 1]	A[0, 2]	A[0, 3]	
1	A[1, 0]	A[1, 1]	A[1, 2]	A[1, 3]	
2	A[2, 0]	A[2, 1]	A[2, 2]	A[2, 3]	
3	A[3, 0]	A[3, 1]	A[3, 2]	A[3, 3]	

The elements would be stored in the following order:

Element Position	Element
1	A(1, 1)
2	A(1, 2)
3	A(1, 3)
4	A(2, 1)
5	A(2, 2)
6	A(2, 3)
7	A(3, 1)
8	A(3, 2)
9	A(3, 3)

CHANGING MATRIX DIMENSIONS

A number of matrix statements allow dimensioning or redimensioning of a matrix. A previously dimensioned matrix can be redimensioned as long as the new dimensions do not exceed the size of the matrix given in the DIM statement.

```
20 DIM  A  (15,14)          ←210 elements in matrix (240 in array A)
40 MAT A = CON (20,7)       ←140 elements
60 MAT A = ZER (10,10)      ←100 elements
80 MAT A = IDN (20,8)       ←160 elements
```

The statements are described in more detail later in the chapter.

MATRIX MANIPULATION STATEMENTS

Descriptions of matrix manipulation statements following are primarily intended for users who are unfamiliar with matrix arithmetic.

Store Copy of Matrix

```
MAT  A = B
```

Elements of matrix B are stored in matrix A. Given the statement:

```
20 MAT  A=B
```

where B is the matrix:

```
2  4  6  8

1  3  5  7
```

Matrix A will assume the identical dimensions and values:

```
2  4  6  8

1  3  5  7
```

MATRIX MANIPULATION STATEMENTS (Continued)

Addition and Subtraction

```
MAT A = B+C   or   MAT A = B-C
```

Matrices B and C must have the same dimensions. Only a single arithmetic operation is permitted in one statement. One of the two operands of the matrix expression may be the name of the matrix appearing on the lefthand side of the = sign.

```
A = B+C-D      ← illegal

A = B+C
A = A-D        ← legal
```

Matrix addition and subtraction is scalar arithmetic performed element by element. Given the statement:

```
20 MAT A = B+C
```

If B and C are matrices having the values:

```
-2    -5        6    4
 3     4       -2   15
.5     .1      1.5   4
 B                C
```

Addition and Subtraction (Continued)

Then the resultant value for A will be:

```
4       -1
1       19
2       4.1
```

Scalar Multiplication

```
MAT A = (expression) *B
```

where: expression may be any numeric expression and must be enclosed in parentheses.

Scalar multiplication is performed element by element. The matrix in the expression may be the same as the matrix variable on the lefthand side of the = sign. Given the statement:

```
30 MAT A = (COS (X)) *B
```

COS (X) is evaluated. If COS (X) evaluates to .254 and B is the matrix:

```
-.5      .8
1.5      -1
```

Then A will be the matrix:

```
-.127      .2032
.381       -.254
```

Zero Matrix

```
MAT A = ZER

MAT A = ZER (d1)

MAT A = ZER (d1, d2)
```

where: d_1 is the number of rows of the matrix.
d_2 is the number of columns of the matrix

A matrix, except for values in row zero or column zero positions, is set to all zeroes by this statement. If the matrix exists and was previously dimensioned, the format

```
MAT A = ZER
```

is used. If the matrix was not previously dimensioned or is to be redimensioned, one of the other formats is used. For example:

```
100 MAT A = ZER (3, 3)

0     0     0

0     0     0          a 3x3 matrix

0     0     0
```

```
110 MAT B = ZER (5)

0
0
0                      a 5x1 matrix
0
0
```

MATRIX MANIPULATION STATEMENTS (Continued)

Zero Matrix (Continued)

```
120 MAT INPUT C (2,4)
      :
      :
220 MAT C = ZER

0    0    0    0
0    0    0    0
```

a 2x4 matrix

Unit Matrix

```
MAT A = CON
MAT A = CON (d1)
MAT A = CON (d1 , d2)
```

where: d_1 is the number of rows of the matrix.

d_2 is the number of columns of the matrix.

A matrix, except for values in row zero or columm zero positions, is set to all ones by this statement. If the matrix exists and was previously dimensioned, the format

```
MAT A = CON
```

is used. If the matrix was not previously dimensioned or is to be redimensioned, one of the other formats is used. For example:

```
100 MAT A = CON (3, 2)
1    1
1    1
1    1
```

a 3x2 matrix

Unit Matrix (Continued)

```
110 MAT B = CON (6)
1
1
1
1
1
1
```

a 6x1 matrix

```
150 MAT C = ZER (2, 3)
    :
    :
300 MAT C = CON
1     1     1
1     1     1
```

a 2x3 matrix

Identity Matrix

```
MAT A = IDN
MAT A = IDN (d1)
MAT A = IDN (d1, d2)
```

where: d_1 is the number of rows of the matrix.

d_2 is the number of columns of the matrix.

The major diagonal of the matrix is set equal to ones and the remaining element of the matrix are zeroed by the statement.

The major diagonal is the diagonal that starts at the final element of the array and runs diagonally upward from the last element until the first row is encountered.

Identity Matrix (Continued)

If the matrix has been previously dimensioned, the format

```
MAT A = IDN
```

can be used. If the matrix was not previously dimensioned or is to be redimensioned one of the other formats is used. Some examples of the identity matrix are:

```
100 MAT A = IDN (4,4)
1       0       0       0
0       1       0       0
0       0       1       0
0       0       0       1
```

In a square matrix, the major diagonal terminates at the first element of the matrix.

```
130 MAT B = IDN (4)
0
0
0
1
```

If a matrix contains only one column, only the last element of the matrix is considered to belong to the major diagonal.

```
140 MAT C = CON (2, 3)
      :
      :
170 MAT C = IDN
0       1       0
0       0       1
```

If a matrix is two-dimensional but not square, the major diagonal terminates at row 1 but not at column 1.

Matrix Transposition

```
MAT A = TRN (B)
```

A matrix is transposed by reversing its rows and columns. A matrix cannot be transposed into itself.

```
200 MAT A = TRN (B)
```

where: B = 4 5 7 9
0 0 0 0
1 3 5 7

When the statement is executed

```
      4   0   1
      5   0   3
A  =  7   0   5
      9   0   7
```

Matrix Multiplication

```
MAT A = B*C
```

Within the matrix expression, the number of columns of the first matrix (B) must match the number of rows of the second matrix (C). The resultant matrix will be dimensioned to have the same number of rows as B and the same number of columns as C. For example:

```
100 DIM B(3,5), C(5,4),A(6,6)
     :
     :
500 MAT A=B*C
```

A will be a 3x4 matrix: A(3,4)

The matrix appearing on the lefthand side of the equals sign cannot appear as a matrix within the expression. Since the columns of B must match the rows of C, an expression of the form:

```
600 MAT A = B*B
```

means B must be a square matrix

To obtain the matrix product of B*C, each row of B is multiplied by each column of C. Each row/column set is added together to find the resultant matrix element. For example, given the following two matrices, B(3,2) and C(2,2):

```
2   3        -1   -2
1   5         7    8
                C
0   4
  B
```

Matrix Multiplication (Continued)

then:

```
30 MAT A = B*C

    [ B(1,1) *C(1,1) +B(1,2) *C(2,1)]      [ B(1,1) *C(1,2) +B(1,2) *C(2,2)]

A = [ B(2,1) *C(1,1) +B(2,2) *C(2,1)]      [ B(2,1) *C(1,2) +B(2,2) *C(2,2)]

    [ B(3,1) *C(1,1) +B(3,2) *C(2,1)]      [ B(3,1) *C(1,2) +B(3,2) *C(2,2)]

    [ 2*(-1) + 3*7]      [ 2*(-2) + 3*8]        19      20

A = [ 1*(-1) + 5*7]      [ 1*(-2) + 5*8]   =    34      38

    [ 0*(-1) + 4*7]      [ 0*(-2) + 4*8]        28      32
```

Matrix multiplication is non-associative. For example, an attempt to execute the statement, MAT A=C*B, using the matrices B(3, 2) and C(2, 2) defined above, will result in an error message since the number of columns of C do not match the number of rows of B. As another example, given the following two square matrices:

```
2    3     0    -1

1    5     4     6
  B           C
```

then:

```
30 MAT A = B*C  →    (2*0 +3*4)    (2*(-1) +3*6)    =    12    16
                     (1*0 +5*4)    (1*(-1) + 5*6)         20    29
                                                          A
```

If the expression is reversed:

```
40 MAT A = C*B  →    (0*2 + (-1)*1)    (0*3 + (-1)*5)    =    -1    -5
                     (4*2 + 6*1)       (4*3 + 6*5)            14    42
                                                              A
```

Inverse Matrix

```
MAT A = INV(B)
```

The matrix appearing in the expression must be a square matrix (at least 2x2). The matrix appearing on the lefthand side of the statement may appear on the righthand side, i.e., matrices may be inverted into themselves.

The arithmetic of matrix inversion requires a knowledge of matrix determinants and of cofactors of matrix elements. Determinants and cofactors for 2x2 matrices will be described here. For larger matrices, consult a mathematics text.

The determinant of a 2x2 matrix is obtained by multiplying along the diagonals and subtracting the second diagonal from the major diagonal:

$$\begin{vmatrix} 1 & 2 \\ 3 & 4 \end{vmatrix} = (1*4) - (2*3) = -2$$

$$\begin{vmatrix} 1 & 5 \\ 3 & 20 \end{vmatrix} = (1*20) - (5*3) = 5$$

An inverse matrix is defined such that the product of the determinants of the matrix and its inverse is always one. The two matrices would have inverse matrices whose determinants were -.5 and .2 respectively.

Cofactors of matrix elements of a 2x2 matrix are obtained by:

1. Reversing the elements along the major diagonal.
2. Changing the signs of the elements along the other diagonal.

To obtain the inverse matrix, scalar multiply the cofactors by the determinant of the inverse matrix:

If:

$$\text{MAT A} = \begin{matrix} 1 & 2 \\ 3 & 4 \end{matrix}$$

then:

$$\text{INV(A)} = (-.5)\begin{pmatrix} 4 & -2 \\ -3 & 1 \end{pmatrix} = \begin{matrix} -2 & 1 \\ 1.5 & -.5 \end{matrix}$$

If:

$$\text{MAT B} = \begin{matrix} 1 & 5 \\ 3 & 20 \end{matrix}$$

then:

$$\text{INV(B)} = (.2)\begin{pmatrix} 20 & -5 \\ -3 & 1 \end{pmatrix} = \begin{matrix} 4 & -1 \\ -.6 & .2 \end{matrix}$$

By obtaining the determinants of the inverse matrices, we can show that they are in fact the reciprocals of the determinants of the original matrices.

$$\begin{vmatrix} -2 & 1 \\ 1.5 & -.5 \end{vmatrix} = (1) - (1.5) = -.5$$

$$\begin{vmatrix} 4 & -1 \\ -.6 & .2 \end{vmatrix} = (.8) - (.6) = .2$$

Extended BASIC will invert any square matrix except one that has one or more zero elements along the major diagonal.

INPUT AND OUTPUT OF MATRICES

MAT READ Statement

```
MAT READ list of matrices
```

The MAT READ statement is used to read values from the data block into the elements of a matrix or a list of matrices. The matrix may have been previously dimensioned or may be dimensioned in the MAT READ statement.

```
20 MAT READ (M(5,6))
      .
      .
      .
50 DATA 0,2,4,6,8,10,-9,-8,-7,-6,-5,-4,-3,-2,-1,0,1,3,5,7,9,11
60 DATA .1,.0,.5,7,-8,15,-15,35,41,13,18
```

Values from the data block will be read into the 30-element matrix dimensioned as 5x6 in the MAT READ statement. For example:

```
M(1,1) will contain 0
M(1,6) will contain 10
M(3,1) will contain -3 and
M(5,6) will contain 35
```

MAT INPUT Statement

```
MAT INPUT list of matrices
```

The MAT INPUT statement is used to read values from the keyboard into the elements of a matrix or a list of matrices at run time. A matrix appearing in the MAT INPUT statement may have been previously dimensioned or may be dimensioned in the statement.

```
5 MAT INPUT X[2, 3]
```

At run time, the BASIC system will issue the request for input data:

```
?
```

The programmer writes the data values for the first row of the matrix, delimited by commas. The value for the last column of the first row is terminated by a carriage return. The system will then query the programmer for data for the next row.

```
? 2,4,6          ← data values for first row of matrix A[2, 3].
? 7,7,-9         ← data values for second row of matrix A[2, 3].
                 ← matrix is complete and program execution resumes.
```

The programmer must supply the exact number of values to fill a row before giving a carriage return. A line of data containing too many or too few values will be ignored by BASIC, which will continue to query the programmer until each row of the matrix has been filled by a matching data line.

MAT PRINT Statement

MAT PRINT list of matrices

The MAT PRINT statement is used to output values of elements of a matrix or a list of matrices. A matrix appearing in the MAT PRINT statement must have been previously dimensioned.

```
0010 DIM A[10,10],B[10,10]
0020 READ N
0030 MAT A=CON[N,N]
0040 MAT B=CON[N,N]
0050 FOR I=1 TO N
0060   FOR J=1 TO N
0070     LET A[I,J]=1/(I+J-1)
0080   NEXT J
0090 NEXT I
0100 MAT B=INV(A)
0120 PRINT "MAT B = INV(A) = "
0130 MAT PRINT B;
0190 DATA 5

RUN
MAT B = INV(A) =

 1   .5   .333333   .25   .2
 .5   3   .25   .2   .166667
 .333333   .25   5   .166667   .142857
 .25   .2   .166667   7   .125
 .2   .166667   .142857   .125   9
```

CHAPTER 5

FILE I/O

Extended BASIC may be used with Data General's Real Time Disk Operating System (RDOS described in document 093-000075) or with Data General's Stand-alone Operating System (SOS described in document 093-000062). To use Extended BASIC with one of the operating systems, the user should be familiar with file concepts applicable to the particular operating system.

Briefly, a file is a collection of information that is known by and accessible by a file name. The Stand-alone Operating System uses only I/O devices such as the card reader, paper tape punch, cassette and magnetic tape units for File I/O, and all file names are reserved device names. Reserved device names are four characters in length and are either of the form $xxx ($PTR, $CDR, etc.) or of the form xxx: (MT0:, CT1:, etc.). Appendix D contains a complete list of reserved device names. Under RDOS, devices may also be used to contain files, but in addition, files are stored on disk. Files stored on disk are accessible by file names that are listed in special disk files, called directories. Names of files that are available to all system users will be listed in the library disk directory. Names of files available only to a given user are listed in that user's directory.

FILE NAMES

File names are written as string literals or string variables in BASIC. A file name appearing in a BASIC console command must be a string literal. However, a file name appearing in one of the BASIC file statements may be either a literal or a variable.

Some examples of file names might be:

"AØ"	string literals
"AØ.1"	
"AØ.SR"	
F$	string variables
F$(2)	
F$(I+1,J)	

The file names must conform to RDOS requirements for extended file names.

FILE NAMES (Continued)

consisting of alphanumerics and the character $ with an optional extension separated from the file name by a decimal point (.). In addition, BASIC file names must be less than or equal to ten characters (not including the extension).

Only device names may begin with the character $, and these names are re-
** served. Any unreserved file name references a disk file.

Each file on disk or each device is opened for reading or writing by associating the file name with a user file number. Each user may open up to eight files for reading or writing corresponding to file numbers 0 through 7.

** RDOS Extended BASIC only.

OPEN FILE STATEMENT

Purpose: The OPEN FILE statement links a user file name or a system device name with a user file number for further I/O referencing. The statement also determines the mode for using the file (reading, writing, random access, or appending).

Format:

```
OPEN FILE [num-exp1, num-exp2], file-name
```

where: file-name is a literal file name or a string variable evaluating to a file name.

num-exp1 is a numeric expression giving the user file number, which must evaluate to a number in the range 0 through 7 (since 8 is the limit of simultaneously open user channels). The file number is associated with the file name and is used for further references to the file (for reading, writing, closing, etc.).

num-exp2 is a numeric expression giving the mode in which the file is to be opened and must evaluate to a number in the range 0 - 3. Each mode is defined as follows:

Mode 0 - Random Access (Input/Output) **

Only disk files may be opened in random mode. When opened, a random access file can either be read or written. If no file having the name given in the OPEN FILE statement is found in the user directory, an entry for the new file name will be made in the directory.

Mode 1 - Output (write a new file)

Either a disk file or appropriate output device can be opened in this mode. Only writes are permitted to the file. If a file of this name already exists in the user directory, the previous copy is first deleted from the disk. In either case, a new file is created (initialized with 0 length) in the user's directory.

** RDOS Extended BASIC only.

** Mode 2 - Output (append to an already written file)

Any appropriate output file may be opened in append mode. When opened, the file is positioned to the end of the current file so that subsequent data written to the file will extend it. If the file does not exist in the user directory, an entry for the file name will be made in the user's directory.

Mode 3 - Input

Either a disk file or appropriate input device can be opened in this mode. If a disk file is opened in this mode, the file must already exist. Only reads are permitted for a file open in Mode 3. If the file is not found in the user's directory, a search for the file is made in the public directory.

Examples:

```
100 OPEN FILE [0,1], "TEST.1"

110 OPEN FILE [1,3], "$PTR"

120 OPEN FILE [I,M], S$
```

** RDOS Extended BASIC only.

CLOSE FILE STATEMENT

Purpose: The CLOSE FILE statement disassociates a file name and a user file number so that the file can no longer be referenced.** Files are closed when file I/O is complete. Also, it may be necessary to change the mode of an open file. To do so, the file must first be closed and then re-opened using the new mode argument.

Format:

```
CLOSE FILE [num-exp1]
```

where: num-exp$_1$ is the user file number previously associated with a file name in an OPEN FILE statement.

Examples:

```
200 CLOSE FILE [2]

300 CLOSE FILE [I-1]
```

** See page 5-19 for a description of the CLOSE statement, which closes all open channels.

READ FILE STATEMENT

Purpose: The READ FILE statement causes data in binary format to be read from a file for those variables listed in the statement.

Format: The first format is used for reading sequential files; the second format is used for reading a record from a random file.

```
READ FILE [num-exp1], variable-list

READ FILE [num-exp1 , num-exp2], variable-list
```

where: variable-list is a list of numeric and string variables for which values are to be read from the file.

num-exp$_1$ is a numeric expression evaluating to the user file number of a file that has been opened in Mode 3 for sequential access or in Mode Ø for random access.

num-exp$_2$ is a numeric expression evaluating to the number of the record to be read from a randomly accessed file. **

Notes: Each variable in variable-list of the READ FILE statement must correspond in data type to each value being read from the file or the record within the file. If the file contains both numeric and string values, then variables of the appropriate type must be given in the correct order in the READ FILE statement.

In reading a random access file, a read of a record that was never written will input a record of all zeroes.

The EOF function can be used to detect the end of data when transferring data from a file to core. The EOF function is used in conjunction with a transfer statement to provide a statement to which to transfer in case of detection of end-of-file. The format of the EOF function is:

```
EOF (file-number)
```

** RDOS Extended BASIC only.

where: file-number is the number of a file opened for reading.

The EOF function evaluates to an integer indicating whether or not the last read of the file, given by file-number, detected an end-of-file. If an end-of-file was detected, the function returns a 1; otherwise, the function returns a 0. Conditional transfer can be effected if the EOF function is used as a numeric expression in an IF statement.

Examples:

```
100 OPEN FILE [1,3], "$PTR"

120 READ FILE [1], A, B, C, D, E, F, G

130 IF EOF (1) THEN 800

200 OPEN FILE [2,0], "BB"

220 READ FILE [2,50], X, Z$, Y, Z
```

WRITE FILE STATEMENT

Purpose: The WRITE FILE statement causes output of data in binary format to a sequentially accessed file or a record of a randomly accessed file.

Format: The first format writes data to a sequentially accessed file; the second writes data to a record of a randomly accessed file.

```
WRITE FILE [num-exp1], expression-list

WRITE FILE [num-exp1,num-exp2], expression-list
```

where: expression-list is a list of numeric expressions or string variables or literals evaluating to numeric or string values for output.

num-exp$_1$ is a numeric expression evaluating to the user file number of a file previously opened in Mode 1 or 2 for sequential access or in Mode 0 for random access.

** num-exp$_2$ is a numeric expression evaluating to the number of the random record to be written.

Notes: String variables or literals for WRITE FILE in sequential mode must be 132 bytes or less in length.

When writing a random record, the record must have 128 or fewer bytes. (A numeric expression requires 4 bytes; * a string of n bytes requires n+1 bytes). The length of the record is the total of all the bytes in the expression-list.

Examples:

```
0060 OPEN FILE [0,1], "XX.2"
0090 FOR I = 1 TO 50
0100 WRITE FILE [0], A[I], A[I]/I, S$, T$
   :
0700 OPEN FILE [1,0], "DATA5"
0800 WRITE FILE [1,37], I, J, B[I]/A[J]
```

** RDOS Extended BASIC only.

* 8 bytes if floating point hardware option is being used, see Appendix F.

BINARY	ASCII
READ FILE	INPUT FILE
WRITE FILE	PRINT FILE

INPUT FILE STATEMENT

Purpose: The INPUT FILE statement causes data in ASCII format to be read from a file, where the data in the file is formatted as it would be in a teletyped response to an INPUT statement.

Format:

```
INPUT FILE [num-exp1], variable-list
```

where: variable-list is a list of numeric or string variables, such that each variable corresponds in data type to the associated datum in the input file.

num-exp$_1$ is a numeric expression evaluating to the user file number of a file previously opened in mode 3.

Example:

```
0040 OPEN FILE [1,3], "$PTR"

0070 INPUT FILE [1], Z, Y, X, A$, B$
```

The first three data that are read from the paper tape reader must be numerics and the last two must be strings.

The paper tape data file must be formatted for INPUT with either commas or carriage returns between data items.

The EOF function, described on pages 5-6 and 5-7 for the READ FILE statement, can be used to provide a statement to which to transfer in case of detection of end-of-file on the file from which data is being input. The argument to EOF is the number of the file opened for input.

Example:

```
0050 OPEN FILE [1,3], "DATA"
0100 INPUT FILE [1], A,B,C,D,E,F,F1,F1$,F$,G[100]
0110 IF EOF (1) GO TO 1000
      :
1000 PRINT "OUT OF DATA"
```

PRINT FILE STATEMENT

Purpose: The PRINT FILE statement causes output of data in ASCII form. The output file produced is formatted as would terminal copy produced by a PRINT statement. The file may be output directly to an ASCII device such as the line printer or to a disk file for later off-line printing.

Format:

```
PRINT FILE [num-exp1], expression-list
```

where: expression-list is a list of numeric expressions, string variables, and string literals, separated by formatting delimiters (, or ; or TAB function).

num-exp$_1$ is a numeric expression evaluating to the user file number of a file previously opened in Mode 1 or Mode 2.

Examples:

```
0100 PRINT FILE [1], "OUT 6"

0550 PRINT FILE [0], "X="; XSQR="; X ↑ 2; "XCUBE="; X ↑ 3
```

PRINT FILE USING STATEMENT

Purpose: The PRINT FILE USING statement causes values for the expressions given in the statement to be output to a previously opened file in the format specified by a string field given in the statement.

Format:

```
PRINT FILE [num-exp1], USING string-expression ; expression-list
```

where: expression-list is a list of numeric expressions, string variables, and string literals whose values are to be output.

string-expression specifies the format of the field in which the value of each expression is to be output, and is identical to the specification of string-expression given for the PRINT USING statement in Chapter 3.

$num\text{-}exp_1$ is a numeric expression evaluating to the user file number of a file previously opened in either Mode 1 or Mode 2.

Examples:

```
0050 OPEN FILE [0,2], "T5"

0150 PRINT FILE [0], USING "++####.###,", A, B, C, D, E, F
```

The output file could conceivably be used later as the input file for an INPUT FILE statement, because each value is formatted to terminate with a comma.

MAT READ FILE STATEMENT

Purpose: The MAT READ FILE statement causes data in binary format to be read from a file for the arrays listed in the statement. The arrays may have been previously dimensioned or may be dimensioned in the MAT READ FILE statement.

Format: The first format is used for reading sequential files. The second format is used for reading a single record from a random file.

```
MAT READ FILE [num-exp1], array-list

MAT READ FILE [num-exp1, num-exp2], array-list
```

where: array-list is a list of arrays for which values are to be read from this file.

num-exp$_1$ is a numeric expression evaluating to the user file number of a file that has been opened in Mode 3 for sequential access or in Mode 0 for random access.

** num-exp$_2$ is a numeric expression evaluating to the number of the record to be read from a randomly accessed file.

Notes: Previously dimensioned arrays may be listed by array name only. Arrays that are not already dimensioned must be dimensioned in the MAT READ FILE statement.

In reading a random access file, a read of a record that was never written will input a record of all zeroes.

The EOF function, described on pages 5-6 and 5-7 for the READ FILE statement can be used to provide a statement to which to transfer in case of detection of end-of-file on the file from which data is being input. The argument to EOF is the number of the file opened for reading.

Example:

```
0040 OPEN FILE [1,3], "VALUES"
0060 MAT READ FILE [1], A,B,C(3,4),D(5)
0080 IF EOF(1) GO TO 500
0500 PRINT "OUT OF VALUES"
```

** RDOS Extended BASIC only.

MAT WRITE FILE STATEMENT

Purpose: The MAT WRITE FILE statement causes output of data in accordance with previously dimensioned arrays. Output is in binary format to a sequentially accessed file or record of a randomly accessed file.

Format: The first format is used in writing to a sequentially accessed file; the second format is used in writing a single record to a random file.

```
MAT WRITE FILE [num-exp1], array-list

MAT WRITE FILE [num-exp1, num-exp2], array-list
```

where: array-list is a list of previously dimensioned arrays for which values are to be written to the file.

num-exp$_1$ is a numeric expression evaluating to the user file number of a file that has been opened in Mode 1 or 2 for sequential access or in Mode 0 for random access.

num-exp$_2$ is a numeric expression evaluating to the number of the random record to be written. **

Example:

```
0050 OPEN FILE [0,1], "AAA"

0080 MAT WRITE FILE [0], B, C, X
```

** RDOS Extended BASIC only.

MAT INPUT FILE STATEMENT

Purpose: The MAT INPUT FILE statement causes ASCII formatted data to be read from a file for the arrays listed in the statement. The arrays may have been previously dimensioned or may be dimensioned in the MAT INPUT FILE statement.

Format:

```
MAT INPUT FILE [num-exp1], array-list
```

where: array-list is a list of arrays for which values are to be read from the file.

num-exp1 is a numeric expression evaluating to the user file number of a file previously opened in Mode 3.

Notes: Previously dimensioned arrays may be listed by array name only. Arrays that are not already dimensioned must be dimensioned in the MAT INPUT FILE statement.

The EOF function, described on pages 5-6 and 5-7 for the READ FILE statement can be used to provide a statement to which to transfer in case of detection of an end-of-file on the file from which data is being input. The argument to EOF is the number of the file opened for reading.

Example:

```
0010 OPEN FILE [2,3], "XX.AA"

0050 MAT INPUT FILE [2], X(5,5), Y, Z
```

MAT PRINT FILE STATEMENT

Purpose: The MAT PRINT FILE statement causes output of data in accordance with previously dimensioned arrays. Output is in ASCII format to a sequentially accessed file or a record of a randomly accessed file in the case of disk files, or may be to an ASCII device such as the line printer.

Format:

```
MAT PRINT FILE [num-exp1], array-list
```

where: array-list is a list of previously dimensioned arrays for which values are to be written to the file.

num-exp$_1$ is a numeric expression evaluating to the user file number of a file that has been opened in Mode 1 or 2 for sequential access or in Mode 0 for random access.

Example:

```
OPEN FILE [0,0], "Z.22"

MAT PRINT FILE [0], B
```

CHAIN STATEMENT

Purpose: The CHAIN statement provides a means of invoking a BASIC program on disk or on an input device from the currently running program.

The CHAIN statement has the following effect:

If the program is on disk, the system searches the user's directory for filename; if not found, the system will search the library disk directory.

The user's currently running program is cleared from core if the program is found and the new program is loaded into core. If filename is not found, the current program remains in core.

The newly loaded program is run, by default, from the lowest numbered statement in the new program. Optionally, the user may specify where control is to be transferred within the new program, using the CHAIN filename THEN GOTO form of the statement. Thus, the user can specify where, other than the lowest numbered statement, execution is to begin.

(The CHAIN statement is the equivalent of a RUN filename command, see Chapter 6.)

Format:

```
CHAIN filename [ THEN GOTO statement-no. ]
```

where: filename is a literal file name or a string variable evaluating to a file name.

statement-no. is any statement number existing in the program with the specified name, filename.

Examples:

```
100 CHAIN "SUB1"
350 CHAIN "Z$"
200 CHAIN "SQRT" THEN GOTO 356
```

Note: The program chained to must be in SAVE file format.

SAVE STATEMENT

Purpose: The SAVE statement causes the current program (source statements and data) to be written in binary format to a binary output device such as the binary paper tape punch or to a disk file. If written to a disk, the file name is entered into the user's directory, replacing any file of the same name.

A SAVEd program can be reloaded using the LOAD command/statement, via a CHAIN statement, or via a RUN filename statement. Saving a program in binary format (rather than in ASCII format using the LIST command) is recommended as a means of saving a program in a compact format, thereby reducing system overhead. Additionally, a program which may have been partially executed may be saved in this format so that when later LOADed execution can be resumed. That is, it is as if the program were never removed from the system (even though the user may have signed off and back on). Also, a SAVEd program that is reloaded can be edited and listed in ASCII at a later time.

Format:

```
SAVE filename
```

where: filename is the name of a device to which the current program is to be written or the name to be stored in the user directory if the current program is to be written to a disk file.

Examples:

```
100 SAVE "FA.BC"

253 SAVE "$PTP"

555 SAVE "CT0:2"

 725 SAVE S$(1,7)
```

ENTER STATEMENT

Purpose: The statement causes the BASIC statements contained in the ASCII file given by filename to be entered into the current program.

When statements in the file have the same statement number as a line in the current program, the ENTERed line will replace the current line. Where statements in the file have statement numbers different from those of the current program, the statements will be inserted in their proper sequence in the current program. The user can write or edit lines into the current program using the ASCII file as input in much the same way as he would input new program lines at the teletypewriter.

The file to be ENTERed may have been created by a LIST file command (see LIST command write-up) or created as the output of another BASIC program which used PRINT FILE statements, or could have been created outside of the Extended BASIC system. Any ASCII format input device can be used for ENTERing a program. If filename is a disk file, the BASIC system will first search the user directory for the file name. If not found, the library disk directory will be searched. Error messages are returned if either the file does not exist, or it is not in source format (ASCII).

The data portion of an executing program is undisturbed by the ENTER command. That is, variable assignments and all other program statuses remain fixed. Thus, ENTER provides a facility for running subprograms as overlays with all program variables as "common".

Format:

```
ENTER  filename
```

where: filename is the name of a disk file or I/O device containing BASIC statements in ASCII format.

Examples:

```
55 ENTER "$CDR"

10 ENTER "LINES.BC"
```

CLOSE STATEMENT

Purpose: The CLOSE statement will close all open channels. (See page 5-5 for a description of the CLOSE FILE statement which will close a user specified channel.) It is necessary to note that if the user should issue a CLOSE statement when all channels have already been closed, no error message will occur.

Format:

```
CLOSE
```

Examples:

```
50 CLOSE
```

DIRECTORY MAINTENANCE STATEMENTS

When using Extended BASIC with a disk, a number of directories are maintained that contain the file names of files on the disk together with the size in words of each file. Each user has his own directory containing the names of his files. There is also a library directory containing the names of files that are available to all users.

The statements that pertain to directories are those that delete a file name from a directory, and rename a file in a directory.

These statements are:

DELETE
RENAME

Each is described on the following pages.

DELETE **

Format:

```
DELETE filename
```

where: filename is the name of a file in the user's directory.

Purpose: To delete the disk file named filename from the user's directory effectively deleting the file from disk.

Examples:

```
100 DELETE "A0.1"

555 DELETE "TEST"
```

** RDOS Extended BASIC only.

RENAME STATEMENT

Format:

```
RENAME  filename1,  filename2
```

where: filename1 is the current file name.

filename2 is the new name that replaces filename1 in the user directory.

Purpose: The command replaces a file name in the user's directory with a new name.

Examples:

```
123 RENAME "TEST", "SQRT2"

566 RENAME S$, "A"
```

FILE I/O COMMANDS

As described in Chapter 6, the user may issue commands directly from the terminal. Included in these commands are commands having the same format and meaning as the File I/O statements described in this chapter. In addition, there are a number of commands described in Chapter 6 that provide for maintenance of disk file directories.

CHAPTER 6

KEYBOARD MODE OF OPERATION

In keyboard mode of operation, the user can:

Specify file I/O and perform file directory maintenance
Execute programs
Request information about the contents of his program
Edit programs
Perform dynamic debugging
Perform simple desk calculator operations
Vary output page format

These functions are carried out by using certain control keys and issuing keyboard commands. Keyboard commands start with a command word, which may be followed by arguments, and terminate with a carriage return. Some of the commands are keyboard versions of certain BASIC statements; BASIC can recognize such a command since it is not preceded by a statement number.

CONTROL KEYS

ESC

Pressing the ESC key essentially means "interrupt the current operation". The effect depends upon the current state of the system:

1. If a program is being executed and an ON ESC THEN... statement has not been encountered, execution ceases, and the message:

   STOP AT xxxx

   is printed, where xxxx is the statement number before which execution ceased. The system reverts to keyboard mode.

2. If a program is being executed, and an ON ESC THEN statement has been encountered within the program, then control will transfer to statement. It is not possible then to interrupt an executing program unless a statement executed after statement instructs the system to stop. This can be accomplished via ON ESC THEN STOP.

3. If a keyboard command is being executed, it is terminated and the system awaits keyboard input.

CONTROL KEYS (Continued)

ESC (Continued)

4. If the system is in idle mode, ESC activates the terminal for operation. The system is in idle mode immediately after the system has been loaded or after the user issues a BYE command. (See page 6-18.) Activating an idle system must always be done by pressing ESC. The user can then start the sign-on sequence.

5. If the user wishes to issue a keyboard command, and the system is operating in one of the modes indicated above, pressing the ESC key will change the mode to allow the system to accept a keyboard command.

SHIFT L

When the user is writing and editing BASIC programs at the keyboard and when he is responding to an INPUT request, pressing both the SHIFT and L keys simultaneously results in deletion of the line he is currently typing. He may then retype the line.

The symbol \ is printed at the teletype to indicate SHIFT L. BASIC then gives a carriage return/line feed and the user may replace the deleted line, as shown in the example following:

```
90 PROMPT  "OMTEREST \
90 PRINT"INTEREST @ 5% IS:" ; I
```

RUBOUT

When the user is writing and editing BASIC programs at the keyboard and when he is responding to an INPUT request, pressing RUBOUT results in the deletion of the last character in the current line. He may then retype the character.

The symbol ← is printed at the teletype to indicate RUBOUT. The following example shows character deletion and replacement:

```
90  PRO←INT "OM ←←INTEREST @ 6% ←←5% IS: "; I
```

The statement within the program will appear as:

```
90  PRINT "INTEREST @ 5% IS:"; I
```

KEYBOARD COMMANDS

Keyboard commands begin with a key word recognized as a command by BASIC. Some commands include one or more arguments following the key word. A keyboard command is terminated by pressing carriage return ( ) and is immediately executed by BASIC.

Most of the key words described as statements in Chapters 3, 4 and 5 can be used as commands. A complete list of all BASIC key words and the way(s) in which they are used is found on page iv.

There are a number of key words which are recognized solely as keyboard command indicators. They are listed below and are described on pages following.

CON	Continue execution after last statement.	
FILES	List file names in the user's directory.	**
LIBRARY	List file names in the library directory.	**
LIST	List statement(s) from the current program to the terminal or to a file.	
LOAD	Clear any current program (implicit NEW), load a SAVEd file (binary file).	
PAGE	Specify width of output page.	
PUNCH	List statements from the current program to the teletypewriter punch with leader and trailer of nulls.	
RENUMBER	Renumber statements in the current program.	
RUN	Execute the current program or other named program.	
SIZE	Print size in words of current program and user memory space still available.	
TAB	Specify the setting of tabulation zones for output.	
WHATS	Prints directory information about specific file.	**

** RDOS Extended BASIC only.

DIRECTORY MAINTENANCE COMMANDS

When using Extended BASIC with a disk, a number of directories are maintained that contain the file names of files on the disk together with the size in words of each file. Each user has his own directory containing the names of his files, logically referred to as a user directory. There is also a library directory containing the names of those files that are available to all users.

The keyboard commands that pertain to directories are those that list the file names contained in either the user directory or the library directory, FILES and LIBRARY respectively; each is described following.

FILES Command **

Format: FILES

Purpose: The command causes a list of all file names in the user directory to be printed to the terminal. Files names, when printed, will be seperated by a tab.

Example: FILES)

LIBRARY Command **

Format: LIBRARY

Purpose: The command causes a list of file names contained in the library disk directory to be printed to the terminal. File names, when printed, will not be seperated by a tab.

Example: LIBRARY)

** RDOS Extended BASIC only.

COMMANDS THAT LOAD, MODIFY, AND EXECUTE PROGRAMS

The user can, using specific BASIC commands, load a program into core from a saved disk file. Once this program is read into core it is called the current program. A current program may be listed, punched, modified if necessary, and executed. The commands which implement these functions are described on this and pages following.

LOAD Command

Format:

```
LOAD filename
```

where: filename is the name of a binary file created by a previous SAVE command.

Purpose: The command executes an implicit NEW, clearing the current program if any. The file specified is then read into core, becoming the current program. The file named may be on disk or may be on a binary input device such as the paper tape reader. In all cases, however, only a file that was previously SAVEd (see page 5-17) can be LOADed.

If a disk file is specified, BASIC first searches the user's directory. If the file name is not in the user's directory, BASIC searches the library directory for the file name.

When a file is LOADed, it can be listed, modified or executed as desired.

Examples:

```
LOAD "$PTR" ↵

LOAD "MATH3" ↵

LOAD "MT0:1" ↵
```

LIST Command

Format:

LIST

LIST statement-no$_1$

LIST TO statement-no$_2$

LIST statement-no$_1$ { TO , } statement-no$_2$

} [filename]

where: statement-no$_1$ is the first statement to be listed.

statement-no$_2$ is the last statement to be listed.

filename is the name of a device or of a disk file.

Purpose: The LIST command causes all or part of the current program to be listed in ASCII either to the file given by filename or by default to the terminal if no filename is given. (Output of a listing of the current program to the teletype punch is described on the next page, PUNCH command). The range of statements to be listed is determined as follows:

LIST ↵	-	List the entire program starting at the lowest numbered statement.
LIST n_1 ↵	-	List only the single statement numbered n_1.
LIST TO n_2 ↵	-	List from the lowest numbered statement through statement n_2.
LIST n_1 { TO , } n_2 ↵	-	List from the statement numbered n_1 through statement n_2.

When the filename argument is given, the command causes the specified lines to be written to a disk file, called filename, or to the device given by filename. The file created by the LIST command can be read back into core using the ENTER command (see ENTER command writeup). If statements are LISTed to a disk file, filename is entered in the user's directory, replacing any previous file of the same name.

LIST Command (Continued)

Examples:

```
LIST 700 TO 9999 ↵
LIST 200 ↵
LIST "$LPT" ↵
LIST 600 TO 900 "F2.2" ↵
```

List statements 600 to statement 900 to file F2.2

List the entire current program to the line printer.

List statement number 200, by default, to the terminal.

List statements 700 to statement 9999, by default, to the terminal.

PUNCH Command

Format:

```
PUNCH

PUNCH statement-no1

PUNCH TO statement-no2

PUNCH statement-no1 {TO / ,} statement-no2
```

where: statement-no$_1$ is the first statement to be punched.

statement-no$_2$ is the last statement to be punched.

The PUNCH formats shown above are identical in meaning to those given for the LIST command (previous page) except that output is to the terminal punch rather than to the terminal printer or to a file.

Purpose: The PUNCH command is the equivalent of a LIST command when output is to the terminal punch. A leader of null characters precedes the punched listing and a trailer of null characters follows the listing.

The number of null characters punched as both leader and trailer is equivalent to the number of characters given as the limit of page size width (see PAGE command, page 6-15). This represents 7-8 inches of leader or trailer for a 132-character line.

Note that input of the command PUNCH to BASIC does not turn on the punch. The following procedure should be followed:

1. Type the desired PUNCH command followed by a carriage return and immediately press the ON button on the terminal punch.
2. BASIC will punch a null leader, followed by the desired listing, followed by null trailer.
3. When punching is completed, press the OFF button on on the punch.

If the user turned on the punch before typing all or part of his punch command, he should tear off that part of the tape, in front of the leader.

Example:

```
PUNCH 20, 1000 )
```

COMMANDS THAT LOAD, MODIFY, AND EXECUTE PROGRAMS

The user can, using specific BASIC commands, load a program into core from a saved disk file. Once this program is read into core it is called the current program. A current program may be listed, punched, modified if necessary, and executed. The commands which implement these functions are described on this and pages following.

LOAD Command

Format:

```
LOAD filename
```

where: filename is the name of a binary file created by a previous SAVE command.

Purpose: The command executes an implicit NEW, clearing the current program if any. The file specified is then read into core, becoming the current program. The file named may be on disk or may be on a binary input device such as the paper tape reader. In all cases, however, only a file that was previously SAVEd (see page 5-17) can be LOADed.

If a disk file is specified, BASIC first searches the user's directory. If the file name is not in the user's directory, BASIC searches the library directory for the file name.

When a file is LOADed, it can be listed, modified or executed as desired.

Examples:

```
LOAD "$PTR" ↵

LOAD "MATH3" ↵

LOAD "MT0:1" ↵
```

LIST Command

Format:

LIST

LIST statement-no$_1$

LIST TO statement-no$_2$

LIST statement-no$_1$ { TO / , } statement-no$_2$

} [filename]

where: statement-no$_1$ is the first statement to be listed.

statement-no$_2$ is the last statement to be listed.

filename is the name of a device or of a disk file.

Purpose: The LIST command causes all or part of the current program to be listed in ASCII either to the file given by filename or by default to the terminal if no filename is given. (Output of a listing of the current program to the teletype punch is described on the next page, PUNCH command). The range of statements to be listed is determined as follows:

LIST ↵	-	List the entire program starting at the lowest numbered statement.
LIST n_1 ↵	-	List only the single statement numbered n_1.
LIST TO n_2 ↵	-	List from the lowest numbered statement through statement n_2.
LIST n_1 { TO / , } n_2 ↵	-	List from the statement numbered n_1 through statement n_2.

When the filename argument is given, the command causes the specified lines to be written to a disk file, called filename, or to the device given by filename. The file created by the LIST command can be read back into core using the ENTER command (see ENTER command writeup). If statements are LISTed to a disk file, filename is entered in the user's directory, replacing any previous file of the same name.

LIST Command (Continued)

Examples:

```
LIST 700 TO 9999 ↵
LIST 200 ↵
LIST "$LPT" ↵
LIST 600 TO 900 "F2.2" ↵
```

List statements 600 to statement 900 to file F2.2

List the entire current program to the line printer.

List statement number 200, by default, to the terminal.

List statements 700 to statement 9999, by default, to the terminal.

PUNCH Command

Format:

```
PUNCH
PUNCH statement-no1
PUNCH TO statement-no2
PUNCH statement-no1 {TO / ,} statement-no2
```

where: statement-no$_1$ is the first statement to be punched.

statement-no$_2$ is the last statement to be punched.

The PUNCH formats shown above are identical in meaning to those given for the LIST command (previous page) except that output is to the terminal punch rather than to the terminal printer or to a file.

Purpose: The PUNCH command is the equivalent of a LIST command when output is to the terminal punch. A leader of null characters precedes the punched listing and a trailer of null characters follows the listing.

The number of null characters punched as both leader and trailer is equivalent to the number of characters given as the limit of page size width (see PAGE command, page 6-15). This represents 7-8 inches of leader or trailer for a 132-character line.

Note that input of the command PUNCH to BASIC does not turn on the punch. The following procedure should be followed:

1. Type the desired PUNCH command followed by a carriage return and immediately press the ON button on the terminal punch.

2. BASIC will punch a null leader, followed by the desired listing, followed by null trailer.

3. When punching is completed, press the OFF button on on the punch.

If the user turned on the punch before typing all or part of his punch command, he should tear off that part of the tape, in front of the leader.

Example:

```
PUNCH 20, 1000 )
```

SYSTEM INFORMATION REQUESTS

The following commands may be issued to obtain information on the size of the current program and remaining space available, on the attributes associated with a particular file, a file's byte length, the date on which the file was created and the date on which the file was last used.

SIZE Command

Format:

Purpose: The command causes a printout at the terminal of the number of bytes used by the program and the total number of bytes that are still available. This printout is printed with decimal numbers.

Example:

```
SIZE ↵
USED:  6700 BYTES
LEFT:  8077 BYTES
```

Format:

```
WHATS filename
```

**

where: filename is the name of a file currently on disk.

Purpose: This command will print out on the terminal information pertaining to the file with the specified name filename. The type of information printed, and the format in which it will be printed is as follows:

filename attributes byte length date created (date last used)

Example:

```
WHATS "ABC" )
ABC   D  2039  06/14/73 (07/21/73)
```

** RDOS Extended BASIC only.

SPECIFYING THE OUTPUT PAGE FORMAT

PAGE Command

Format:

PAGE = n

where: n is an integer in the range:

$$1 \leq n \leq 132$$

Purpose: The command sets the limit of page width where n is the maximum number of characters that may be output on a line of a given terminal device.

If the command is not given, the default maximum of 72 characters will result.

Example: PAGE = 132 ↵

TAB Command

Format:

TAB = n

where: n is an integer in the range:

$$1 \leq n \leq \text{size of page}$$

Purpose: The command sets the zone spacing desired between output data. If the command is not given, the default zone spacing is 14 characters, which allows five columns of output data to the 72-character teletype-writer line.

Since the maximum range of zone spacing depends upon the page width, it is good practice to set the page width first and then the zone spacing.

Example:
PAGE = 132 ↵
TAB = 12 ↵

COMMANDS DERIVED FROM BASIC STATEMENTS

Any BASIC statement that can meaningfully be written as a keyboard command can be used in that mode. Certain statements having meaning only within the context of a program cannot be used as keyboard commands. These are CALL, CHAIN, DATA, DEF, END, FOR, GOSUB, GOTO, NEXT, ON, REM, RETURN, and STOP. All other statements are implemented as keyboard commands; some use of these statements are:

Perform File I/O

The opening and closing of files and input/output of programs and data from files and devices can be handled by keyboard commands derived from the file I/O statements described in Chapter 5.

```
OPEN FILE [1,3] , "$PTR" ↵

READ FILE [1], A, B, C, D, E, F, G[5] ↵
```

Desk Calculator

The PRINT command can be used as a desk calculator. The command PRINT (;) is followed by any expression. Upon the user's striking RETURN, the system immediately computes the value of the expression and prints it on the same line. The examples show expressions consisting of literal operands.

```
;EXP ( SIN (3.4/8)) 1.51032

;USING "+####.## ↑↑↑↑", EXP(SIN(3.4/8))+1510.32E-03
```

BASIC responds with value on same line.

Desk Calculator - Using Program Values

Besides literal operands, the user can include values assigned to program variables. The user can interrupt a running program and use assigned program values in obtaining values for calculations.

```
0010 DIM A$[10], B$[10]
0020 LET A$ = "IOU $10.50"
0030 B$ = "XRAY"
RUN ↵
(ESC)
;B$(4);A$(2,3) YOU
```

Desk Calculator - Using Program Values (Continued)

```
0010 DIM A[3,3]
      :                                    User writes and runs a BASIC program.
      :
RUN ↵
      (ESC)
STOP AT 0500
;USING "+#.#####E↑↑↑↑", A(1,2),A(1,2)*9  +5.12100E+02  +4.60890E+03
```

Dynamic Program Debugging

A running program can be interrupted (using ESC or by programmed STOP statements) at a number of different program points. The current values of the variables can then be checked at those points and corrections made in the program, either to statements or variables, as necessary. The programmer can then use the RUN statement-no command to restart the interrupted program without losing either the values of the variables at the point of interruption or the newly inserted values and statements.

```
      :
      :
(ESC)
STOP AT 1100
IF A < > B THEN PRINT B, A ↵     ←User command conditionally provides for
.025              .5               examination of A and B.
```

```
      :
      :
2.33333                          ←results of a series of program calculations
5.41234                            being printed.
8.99999
(ESC)
STOP AT 0570
READ X1, X2, X3 ↵                ←user spaces over the next 3 values in the data
RUN 570 ↵                          block and resumes program execution at the
                                   statement at which it was interrupted.
```

Dynamic Program Debugging (Continued)

```
(ESC)
STOP AT 1100
;A 0                      ←User checks value of variable A.
A = -1 ↵                 ←User changes string variable C$ and the value of arith-
C$ = "% OF LOSS" ↵         metic variable A and resumes running at statement 505.
RUN 505 ↵
```

```
      .
      .
      .
20 DIM A[4,4]
      .
      .
      .
(ESC)
STOP AT 500

DIM A[3,5] ↵             ←User redimensions array A.
```

APPENDIX A

ERROR MESSAGES

Error messages are printed as two digit codes, followed by a brief explanatory message when operating under RDOS. There are three types of error messages:

1. Errors that are recognized during program input by BASIC. If the user input the statement in error from the teletype, the incorrect statement will appear on the teletypewriter printer just above the error message. If the statement in error was input from a file or other input device, BASIC will first print the statement in error before printing the error message. The form of the error message is:

   ERROR xx text

   where: xx is a two-digit decimal error code.
   text is a brief description of the error which is printed when operating under RDOS.

   All syntax errors are recognized during program input.

2. Errors, other than file I/O, that are recognized at run time. BASIC system run-time errors cause printout of an error message of the form:

   ERROR xx AT yyyy text

   where: xx is a two-digit decimal error code.
   yyyy is the line number at which the error was detected.
   text is a brief description of the error which is printed when operating under RDOS.

3. I/O error messages. File I/O errors are printed in the format:

   I/O ERROR xx (AT yyyy) text

   where: xx is a two-digit decimal error code.
   yyyy is the line number which is printed if the I/O error is detected at run-time.
   text is a brief description of the error which is printed when operating under RDOS.

The meanings of the decimal error codes for errors other than file I/O are given in the list beginning on the following page. Following each message is a brief description of the message and an example showing its occurrence.

Following the BASIC system errors is a list of the I/O errors and their meanings. Under the heading TYPE are one letter characters, either E or A. E indicates that the error occurs directly after the user has entered the command line on the teletypewriter, after pressing the carriage return key. The letter A indicates that the error occurs during the execution of a program.

BASIC ERROR MESSAGES				
CODE	TYPE	TEXT	MEANING	EXAMPLE
00	E	FORMAT	unrecognizable statement format.	
01	E	CHARACTER	illegal ASCII character or unexpected character.	RUN $100 ENTER #$LPT"
02	E	SYNTAX	unrecognizable keyword or invalid argument type.	10 LETT A = 10 20 IF SIN(A$)=0...
03	A	READ/DATA TYPES IN-CONSISTENT	READ specifies different data type than DATA statement.	10 DIM A$(10) 20 READ A$ 30 DATA 12 RUN
04	A	SYSTEM	Hardware or software malfunction.	
05	E	STATEMENT NUMBER	statement number not in the range: $1 \leq n \leq 9999$.	0000 GOTO 100 99999 STOP 0010 GOTO 81373
06	E	EXCESSIVE VARIABLES	attempt to declare more than 286 variables.	
07	E	COMMAND(I/O)	attempt to execute a command from a file (and not in a BATCH mode).	ENTER "ABC" & file ABC contains a LIST command
08	E	SPECIFICATION	value specified is not within limits (PAGE/TAB)	PAGE = 200 PAGE = 72 TAB = 80

BASIC ERROR MESSAGES				
CODE	TYPE	TEXT	MEANING	EXAMPLE
09	E	ILLEGAL RESERVED FILE NAME	reserved file name not recognized by system (see system generation for valid names)	ENTER "$ABC"
10	E	RESERVED FILE IN USE	another user has control of the specified I/O device. (Except $LPT - requests are queued.)	USER A: ENTER "$PTR" USER B: ENTER "$PTR"
11	E	PARENTHESIS	parentheses in an expression are not paired.	A = ((B-C)
12	E	COMMAND	system cannot execute keyboard command.	GOSUB 100 NEXT I
13	E/A	LINE NUMBER	attempt to delete or list an unknown line; attempt to transfer to an unknown line.	100 10 GOTO 100 RUN
14	E	PROGRAM OVERFLOW	not enough storage to ENTER source program.	ENTER "ABC"
15	A	END OF DATA	not enough DATA arguments to satisfy READ	10 READ A,B,C 20 DATA 91,21 RUN
16	A	ARITHMETIC	value too large or too small to evaluate	A = 1234 + 66 ; A ↑ 20
17	A	UNASSIGNED VARIABLE	attempt to reference an unknown variable	;A

BASIC ERROR MESSAGES				
CODE	TYPE	TEXT	MEANING	EXAMPLE
18	A	GOSUB NESTING	more than 6 nested GOSUB's	
19	A	RETURN - NO GOSUB	RETURN statement encountered without a corresponding GOSUB	10 RETURN RUN
20	A	FOR NESTING	more than 4 nested FOR's	
21	A	FOR - NO NEXT	FOR statement encountered without corresponding NEXT	1 FOR I=1 TO 10 2 ;I RUN
22	A	NEXT - NO FOR	NEXT statement encountered without a corresponding FOR	10 NEXT I RUN
23	A	DATA OVERFLOW	not enough storage left to assign space for variables	10 DIM A(300000) RUN
24	A	NO AVAILABLE CHANNELS	channel limit specified at SYSGEN time has been reached	
25	A	OPTION	matrix operations were not specified at BASIC SYSGEN	MAT PRINT A
26	A	PROGRAM/ DATA OVERFLOW	attempt to LOAD or RUN a SAVE'd file which is too large for available storage	LOAD "ABC"

BASIC ERROR MESSAGES				
CODE	TYPE	TEXT	MEANING	EXAMPLE
27	A	FILE NUMBER NOT 0-7	invalid file designation in an I/O statement	OPEN FILE(9,0)
28	A	DIM OVER-FLOW	an array or string exceeds its initial dimensions	
29	A	EXPRESSION	an expression is too complex for evaluation	A=(((A+1)+((A-7+3)*3)+RND(0))
30	A	NODE NUMBER NOT 0-3	invalid mode designation in an I/O statement	OPEN FILE(0,7)
31	A	SUBSCRIPT	subscript exceeds array's dimension	10 DIM A$(2) ;A$(1,30) RUN
32	A	UNDEFINED FUNCTION	statement looks like a function but was never defined by DEF and not a standard BASIC function	;ABC(1)
33	A	FUNCTION NESTING	the nesting of too many defined functions.	
34	A	FUNCTION ARGUMENT	argument range exceeded	A = 1234 ;A ↑ 34652
35	A	ILLEGAL FORMAT STRING	PRINT USING statement is illegal	;USING "A",A
36	A	STRING SIZE	print line exceeds page specification	PAGE = 15 ;"AAA ...

BASIC ERROR MESSAGES				
CODE	TYPE	TEXT	MEANING	EXAMPLE
37	A	USER ROUTINE	CALL statement specifies a user routine not in storage	10 CALL 2 RUN
38	A	UNDIMEN - SIONED STRING	attempt to reference an unknown string variable	;A$
39	A	DUP MATRIX	same matrix appears on both sides of a MAT multiply or transpose statement.	10 DIM A(10,10) 20 MAT A=A*A RUN
40	A	MATRICES SIZES	matrices have different sizes	10 DIM A(10,10) 20 DIM B(20,20) 30 MAT A = B RUN
41	A	MATRIX DIM	matrix has a zero dimension	10 DIM A(10) 20 DIM B(10,10) 30 MAT B = A RUN
42	A	FILE ALREADY OPEN	two OPEN statements without an intervening CLOSE	OPEN FILE (0,0)... OPEN FILE (0,0)...
43	A	MATRIX NOT SQUARE	attempt to invert a non-square matrix	10 DIM A(20,30) 20 MAT B = INV(A) RUN
44	A	FILE UNOPENED	an attempt to read/write a file which has never been opened	DIM A$(10) WRITE FILE(0),A$

BASIC ERROR MESSAGES				
CODE	TYPE	TEXT	MEANING	EXAMPLE
45	A	RECORD ≥ 128 BYTES	logical record size limit exceeded	DIMA$(300) OPEN FILE(0,1)"ABC" WRITE FILE(0),A$
46	A	INPUT	data entered in response to INPUT is incorrect	INPUT A ? ABC
47	A	WRONG MODE	input file opened for writing or output file opened for reading.	OPEN FILE(0,1),... READ FILE(0),...
48	E	NOT A SAVE FILE	a LOAD, RUN or CHAIN was attempted on a file which was not previously SAVEd	
49	E	NO ROOM FOR DIRECTORY	FILES or LIBRARY commands cannot find 256 words in user program storage to read disk directory	10 DIM A(8000) RUN FILES
50	E	INVALID OPERATOR COMMAND	a command preceded by a # (operator command specifier) is not recognized (See page C-15)	#ABC
53	E	RENUMBERING ERROR(S)	a reference is made to at least one non-existing statement number	100 GOTO 1090 110 END
54	E	STATEMENT LENGTH	statement length exceeds 132 characters in internal representation	
58	E	INCOMPATIBLE SAVE FILE	incompatible version of BASIC (i.e., REV. 3.0 or earlier) or different floating point configuration	

FILE I/O ERROR CODES AND THEIR MEANINGS

ERROR CODE	MEANING	ERROR CODE	MEANING
0	Illegal channel	36	Squash file
1	Illegal file number	37	Device already exists
2	Illegal system command	38	Insufficient contiguous blocks
3	Illegal command for device	39	QTY
4	Not a saved file	40	Task queue table
5	File already exists	41	No more DCB's
6	End of file	42	DIR specifier
7	Read-protected file	43	DIR specifier
8	Write-protected file	44	DIR too small
9	File already exists	45	DIR depth
10	File not found	46	DIR in use
11	Permanent file	47	Link depth
12	Attribute protected file	48	File in use
13	File not opened	49	Task ID
14	Swapping disk error-program lost	50	Common size
		51	Common usage
17	UFT in use	52	File position
18	Line limit	53	Data chain map
19	Image not found	54	DIR not initialized
20	Parity	55	No default DIR
21	Push limit	56	FG already active
22	Storage overflow	57	Partition set
23	No file space	58	Insufficient arguments
24	Read error	59	Attribute
25	Select status	60	No Debug
26	Start address	61	No continuation address
27	Storage protect	62	No start address
29	Different Directories	63	Checksum
30	Device name	64	No source file
31	Overlay number	65	Not a command
32	Overlay file attribute	66	Block type
33	Set time	67	No files match
34	No TCB's	68	Phase
		69	Excess arguments

INITIALIZATION ERRORS

The following table is a list of initialization errors. Some errors are fatal but should not occur. When a fatal error occurs call your Data General representative. The remaining errors indicate conditions that can be remedied.

INITIALIZATION ERRORS		
CODE	TEXT	MEANING
1-8	INITIALIZATION ERROR	Call your Data General Representative
9	INITIALIZE MASTER DIRECTORY	BASIC.DR is not present
10 & 11	INITIALIZATION ERROR	Call your Data General Representative
12	RESERVE AVAILABLE MEMORY	Insufficient memory to execute BASIC.SV
13	CREATE SWAPPING FILE	Insufficient contiguous blocks for BASIC.SW. Remedy: Use RDOS CLI to rearrange disk files or reduce the maximum space allowed for user's BASIC program. BASIC.SW is present with a non-zero use count. Remedy: Execute RDOS CLI command "CLEAR BASIC.SW".
14-21	INITIALIZATION ERROR	Call your Data General Representative
22	IDENTIFY MULTIPLEXOR INTERRUPT HANDLER	For systems with 4060 type multiplexors, the QTY device was sysgened into the RDOS system. Remedy: Configure an RDOS without the QTY option. If this error occurs for either a 4100 or a 4026 multiplexor, call your DGC representative.
23	INITIALIZATION ERROR	Call your Data General representative.

APPENDIX B

CALLING AN ASSEMBLY LANGUAGE SUBROUTINE FROM EXTENDED BASIC

It is possible to call a subroutine written in assembly language from an Extended BASIC program. The format of the BASIC call is:

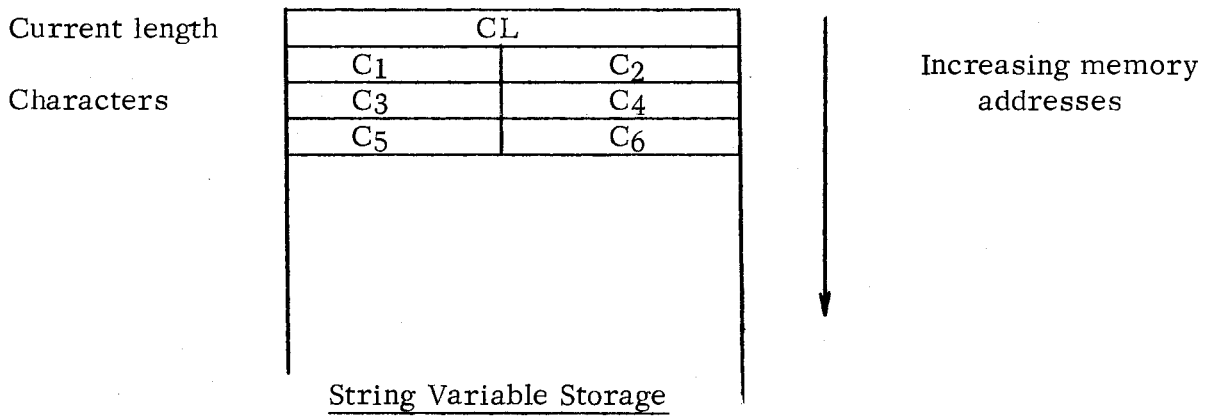
```
CALL sub#[, A1, ..., An]
```

where: sub# is a positive integer (in the range 0 to 32767) representing the subroutine number.

$A_1,\ldots,A_n$ are optional arguments to be passed to the subroutine (n must be in the range 1 to 8) and may be arithmetic variables or expressions, or string variables or expressions. Dimensioned numeric variable names should not appear alone, i.e., without subscripts. (Statement numbers are not permitted as arguments.)

Character String Storage and Definitions

The assembly language programmer should be aware of the following information if he wishes to handle character strings in a CALLed subroutine. BASIC keeps a count of the number of characters currently defined in each string variable (referred to as the current length of the string variable). A current length is stored as part of a header immediately preceding the contents of each string variable. (See illustration below.) The current length must be updated each time characters are added to or taken away from the string variable.



Character String Storage and Definitions (Continued)

In the following examples, assume that A$ is dimensioned to 10, and A$ = "ABCDE". The current length of A$ is 5.

A substring is defined as any contiguous part of a string variable. For example:

A$(2,4) and A$ are substrings of A$.

The current length of a substring is defined as the number of defined characters within the substring. For example, the current length of A$(4,7) is 2, because only A$(4,4) and A$(5,5) are defined.

The maximum length of a substring is defined as the number of character positions within the substring. For example, the maximum length of substring A$(4,7) is 4.

Linking the Assembly Language Subroutine

The user's assembly language subroutines must be given as input to the relocatable loader when the BASIC system save file is created. The user must include a subroutine table with his subroutines. The table must have the entry point SBRTB.

The subroutine table is a list of all assembly language subroutines available to a BASIC program. For each assembly language subroutine a four-word list item is required, containing the following:

subroutine number
subroutine entry point
number of arguments
argument control word

The table is terminated by an item whose subroutine number is -1.

The argument control word is used by BASIC to give run-time error checking on the types of arguments. The argument control word is divided into eight two-bit fields for the eight possible arguments $A_1 \ldots A_8$. The value of the two bit field determines the allowable argument.

00_2 ←argument may be any string expression
01_2 ←argument must be a string variable
10_2 ←argument may be any numeric expression
11_2 ←argument must be a numeric variable

Linking the Assembly Language Subroutine (Continued)

BASIC calls the assembly language subroutines by the sequence:

```
        LDA     2,.+2           ;load
        JSR     <SUB>           ;jump to subroutine
        ADLST                   ;address list
ADLST:  <address of A1 or A1
        descriptor words >
        .
        .                       ;addresses of passed arguments
        .
        <address of An or An
        descriptor words >
        (return point)
```

If A_i is a substring of a string variable, the address list contains the address of the string descriptor words, which contain the following information:

word 1: byte address of the first character of the substring
word 2: current length of the substring
word 3: maximum length of the substring
word 4: word address of the current length of the string variable

If A_i is a string expression, the address list contains the address of the string descriptor words, which contain the following information:

word 1: byte address of the first character of the string
word 2: length of the string

If A_i is a numeric variable, the address list contains the storage address of the variable. (All numeric variables are represented in standard floating point format. See page 2-1.)

If A_i is a numeric expression, the address list contains the storage address of the value of the expression.

```
                .TITLE  SBRTB
                .ENT SBRTB
                .NREL
SBRTB:          7                       ;SUBROUTINE NUMBER
                A                       ;SUBROUTINE ENTRY POINT
                5                       ;NUMBER OF ARGUMENTS
                2B1+2B3+3B5+1B7+0B9     ;ARGUMENT CONTROL WORD
                4                       ;SUBROUTINE NUMBER
                B                       ;SUBROUTINE ENTRY POINT
                0                       ;NUMBER OF ARGUMENTS
                0                       ;ARGUMENT CONTROL WORD
                -1                      ;END OF SBRTB


A:              (coding for subroutine A)
                .
                .
                .
                JMP  3,2                ;RETURN

B:              (coding for subroutine B)
                .
                .
                .
                JMP  0,2                ;RETURN
                .END
```

Legal calls from BASIC to the subroutines of the examples are:

```
CALL 7, Q+17, B, B2, A$(2,4), "TIME"
CALL 4
```

Illegal calls which would result in an error message would be:

```
CALL 7, Q+17, B, B2*2, A$, ""     Third parameter must be a variable.
CALL 7, Q+17, B                   Not enough parameters.
CALL 4, Q                         Too many parameters.
CALL 2, A, B                      No subroutine number 2.
```

An illegal CALL, causing error 17, will result from an attempt to pass a variable in the CALL that does not have a previously assigned value. All variables passed in the CALL must have been previously assigned values even if their current value is not to be used in the CALLed subroutine.

Several subroutines are available in BASIC to help the user in manipulating numbers and character strings. The pointers to the routines are in page zero and should be declared as displacement externals.

Routines		Result*
.FIX		Converts floating point number in AC0-AC1 to an integer in AC0-AC1. If there is overflow, the largest possible integer is returned in AC0-AC1. Bit 0 of AC0 is the sign of the number. Bit 0 of AC1 is a significant bit.
.FLOT		Converts an integer in AC0-AC1 to floating point format in AC0-AC1.
.ADDF .SUBF .MPYF .DIVF	F0+F1 F0-F1 F0*F1 F0/F1	Arithmetic routines to perform floating point add, subtract, multiply, divide. In each routine, AC0-AC1 initially contains the floating point value of F1 and AC2 contains the address of the value of F0. The result is returned in AC0-AC1. Underflow returns a zero result; overflow results in error number 16.
.MPY .MPYA	A1*A2 → A0,A1 A0+A1*A2 → A0,A1	In the integer multiply routines, AC1 contains the unsigned integer multiplicand and AC2 contains the unsigned integer multiplier. The result is a double length product with high-order bits in AC0 and low-order bits in AC1. Contents of AC2 are unchanged. The difference between the routines is that .MPYA adds the result of the multiplication to the contents of AC0.

* In systems having floating point hardware, the floating point number is stored in the Floating Point Accumulator (FPAC) rather than in AC0-AC1.

Routine		Result
.DVD .DVDI	(A0,A1)/A2 → A1,A0 A1/A2 → A1,A0	In the integer divide routines the dividend is in AC1 (single-length) or in AC0 and AC1 (double-length with high order bits in AC)). The divisor is in AC2 and the result is left with the quotient in AC1 and the remainder in AC0. Contents of AC2 are unchanged.
.MOST		Moves the character string described by the string descriptor words in AC0, AC1 to the substring described by the string descriptor words in the memory locations labeled TR3, TR4, TR5. TR6.*

Before a JSR to MOST, these accumulators and memory locations should be loaded as follows:

- AC0 - byte address of the first character of the source string
- AC1 - length of the source string
- TR3 - byte address of the first character of the destination string
- TR4 - current length of the destination substring
- TR5 - maximum length of the destination substring
- TR6 - word address of the current length of the destination string variable.

TR3, TR4, TR5, and TR6 should be declared as displacement externals in the assembly language subroutine. MOST automatically updates the current length of the destination string variable. Subroutine MOST has two returns. Return at CALL + 1 means the character string move was terminated by the source string becoming empty.

Return at CALL + 2 means the move was terminated by the destination substring becoming full.

* In the SOS BASIC system, these locations are labeled TS1, TS2, TS4, TS5.

APPENDIX C

EXTENDED BASIC OPERATION UNDER RDOS

CONFIGURING RDOS

The system generation program, SYSGEN.SV, configures a system save file by interrogating the user as to the characteristics of the system which the user wishes to generate. To invoke the SYSGEN program, the user types the command:

```
SYSGEN )
```

The CLI will then load and transfer control to SYSGEN.SV which will issue the message:

```
SYSGEN REV n.nn
VALID ANSWERS ARE IN PARENTHESES RESPOND ACCORDINGLY
```

followed by a series of questions to which the user must respond; number responses are decimal integers. Each response must be followed by a carriage return. A simple carriage return response will be interpreted as a response of 0. The Real Time Disk Operating System User's Manual, 093-000075, Appendix E, lists all the queries issued during the SYSGEN procedure and the appropriate responses for each. Below are several of those queries and the proper responses of each as relating to Extended BASIC.

QTY? ("0" = NO, "1" = YES)

RDOS should be configured without the QTY handler (response of 0) if Extended BASIC will be using the multiplexor handler type 4060.

RTC? ("0" = NO, "1" = YES)

The proper response to this question is 0 if your system lacks a real time clock, or 1 if there is a real time clock in your system.

ENTER RTC FREQ (1 = 10HZ, 2=100HZ, 3= 1000HZ)

This question will be asked only if the response to the previous question was 1. The RTC interrupt rate should be configured to 10 HZ (response of 1) when configuring RDOS for Extended BASIC.

ENTER NUMBER OF STACKS (1-5)

This question refers to the number of system stacks which you wish to be available to RDOS. It is preferable for the user to enter 5 in response to this query for either a multi-user or single-user system (a minimum response of 2 is necessary for BASIC).

ENTER NUMBER OF EXTRA BUFFERS REQUIRED (0-N)

This question refers to the number of system buffers which you wish to make available to RDOS. Respond with a non-negative integer equal to the greatest number of buffers as core will allow. The guidelines for selecting the number of extra core buffers allowed in your system follows.

Each system buffer is 414 (octal) words in length; the system will allocate a minimum of six of these buffers or two buffers for each system stack, whichever is larger. If multiples of 414 (octal) core locations are available for RDOS in a mapped system, these multiples will be used for additional system buffers. System buffers are used to receive system (not user) overlays and for I/O transfers; thus the speed of RDOS Extended BASIC is enhanced by the availability of extra buffers.

MAXIMUM NUMBER OF SUB-DIRECTORIES/SUB-PARTITIONS ACCESSIBLE AT ONE TIME (0-32)

If you do not wish to use disk partitions in this system, respond with a 0; otherwise, respond with the maximum number of subpartitions and/ or subdirectories which you want to be accessible (i.e., able to be initialized). This number is found by using the formula below:

1 for each user +1 for the library = response

OPERATOR MESSAGES? ("0" = NO, "1" = YES)

The proper response to this question is 1 if you wish to be able to issue read/write operator message system calls (.RDOP/.WROP); otherwise, response with 0. It is recommended, when using Extended BASIC, to respond with a 1. (If not included initialization errors will not appear at the system console.)

CONFIGURING RDOS (Continued)

MAPPED SYSTEM ("0" = NO, "1" = YES)

Respond with 0 if your system has no Memory Management and Protection Unit (MMPU); respond with 1 if your system does have MMPU.

USER INTERRUPT SERVICE ("0" = NO, "1" = YES)

This question will be asked if you responded with a 1 to the above question. Using Extended BASIC under RDOS, you should respond with a 1 to this question.

MAXIMUM NUMBER OF CHANNELS BACKGROUND WILL USE (1-N)

MAXIMUM NUMBER OF CHANNEL FOREGROUND WILL USE (1-N)

These questions will be asked only if you have specified that your system employs mapped addressing (a response of 1 to the query MAPPED SYSTEM above). Respond with an integer value N in the range 0-63. If N is selected to be smaller than the requirement specified at load time, the program will be aborted when execution is attempted.

BASIC CONFIGURATION

Before the Extended BASIC system can be loaded, the system must be configured.

To configure the BASIC system under RDOS, the user executes the RDOS CLI and then gives the command:

```
BSG )
```

BSG responds by querying the user as to the device configuration for Extended BASIC. BSG.SV is supplied as a part of the BASIC package.

BSG Dialogue

After the user has issued the command BSG, the system will respond with a series of queries as to the type of device configuration the user wishes for his Extended BASIC system. The queries and applicable responses are listed following. The following series of interrogations is numbered for your convenience; no such numbering occurs during the operation of BSG.

BSG Dialogue (Continued)

1. MAPPING SYSTEM?

   The user responds with a Y to this query if his system is to be a mapped system (i.e., his system includes a Memory Management and Protection Unit (MMPU), or N if not.

2. BATCH SYSTEM?

   If the system to be configured is to be a BATCH system respond with Y, if not, respond with N. If you respond with a Y to this query, jump to the query numbered 10; otherwise continue with query number 3.

3. MULTI-USER SYSTEM?

   The system is querying as to whether the system is a Single-User System (respond with N) or a Multi-user System (respond with Y). If response is N, jump to the query numbered 10; otherwise, continue with query number 4.

4. 4026 MULTIPLEXOR?

   There are three possible multiplexor handlers available to the user, type 4100, type 4060 and type 4026. By responding with a Y to this query, your system will be configured with multiplexor handler type 4026; a N will prompt the next query.

4a. 4100 MULTIPLEXOR?

   A Y will configure your system with handler type 4100; by responding with a N to this query, your system will be configured with multiplexor handler type 4060.

5. SWAPPING SYSTEM?

   This query determines whether the Extended BASIC system is a swapping or non-swapping system. The user responds with Y or N, depending upon his system configuration. If the user responds with Y, the swapping system will create a file called DK0:BASIC.SW which will be used for program swapping.

6. LINE CONFIURATION?

This query requests the line configuration to be used. The user responds to the query with a list of line numbers, a range of line numbers, or both, followed by a carriage return. The 4026 multiplexor uses lines 0-15 (maximum of 16 lines). The 4060 and 4100 multiplexors use lines 0-31 (maximum of 32 lines). The user lists line numbers separated by commas, or gives a range of line numbers as follows:

0,1,2,4 - line numbers 0,1,2, and 4

0-2, 4 - line numbers 0,1,2, and 4

These line numbers, or range of line numbers are terminated by a carriage return.

7. DIAL-UP CONFIGURATION?

The user responds to the query with a list of dial-up line numbers, a range of dial-up line numbers, or both, selected from those line numbers given in response to query number 6, LINE CONFIGURATION. Dial-up lines are those connected by an interface to a telephone line. The response to the query has the same format as the response to the LINE CONFIGURATION query. The dial-up line numbers are terminated by a carriage return. If there are no dial-up lines, the user responds to the query with a carriage return.

8. CONSOLE TTY?

The user responds to this query with Y if he wishes to use the teletype-writer as the master console or he responds with a N if he does not. If the user responds with a Y, jump to query number 10; if the response is N, continue on with query number 9.

9. MASTER CONSOLE LINE NO. ?

The user responds to this query with the line number of the master console to be used. (The master console is the only terminal in the system having write-access to library files as well as access to all user directories.)

BSG Dialogue (Continued)

10. RESERVED FILES:

The user responds with a list of reserved names of devices to be used during the console session. Any number of devices can be listed from the reserved devices. The reserved file names are:

Reserved Name	Device
$PLT	Incremental Plotter
$LPT	Line Printer
$CDR	Card read (including mark sense)
$PTR	Paper Tape Reader
$PTP	Paper Tape Punch
MT0:	Magnetic Tape Unit 0
MT1:	Magnetic Tape Unit 1
MT2:	Magnetic Tape Unit 2
MT3:	Magnetic Tape Unit 3
MT4:	Magnetic Tape Unit 4
MT5:	Magnetic Tape Unit 5
MT6:	Magnetic Tape Unit 6
MT7:	Magnetic Tape Unit 7
CT0:	Cassette Unit 0
CT1:	Cassette Unit 1
CT2:	Cassette Unit 2
CT3:	Cassette Unit 3
CT4:	Cassette Unit 4
CT5:	Cassette Unit 5
CT6:	Cassette Unit 6
CT7:	Cassette Unit 7

The console TTY should never be configured as a reserved file name. To indicate the end of the reserved name list, the user types a carriage return which is not preceded by a reserved file name.

11. MARK SENSE CARD READER?

If $CDR was included as a reserved file name in response to query number 10, this query will be printed. If $CDR was not included in the list of reserved file names, continue with query number 12. With a response of Y to this query, all cards input to the card reader will be interpreted to be mark sense cards. A response of N to this query indicates that the cards will be interpreted as Hollerith punched cards.

BSG Dialogue (Continued)

12. NUMBER OF USER DISK CHANNELS?

    The user responds with a decimal number indicating the number of disk files that can be open at a given time. The number must be in the range 1 to 64_{10}-n, where n equals the number of reserved files specified by the user in response to query 10. The user must include 1 channel for the swapping file (if used) and 2 channels if using BATCH in addition to other desired channels.

13. HARDWARE FLOATING POINT?

    If the hardware floating point option is to be a part of the user's system configuration, he should respond with Y to the query; otherwise, he should respond with N.

14. HARDWARE MULTIPLY/DIVIDE?

    If the hardware multiply/divide option is to be a part of the user's system configuration, he should respond with Y to this query; otherwise, respond with a N and jump to query number 16.

15. ORIGINAL NOVA?

    If the response to query number 14 was Y, the ORIGINAL NOVA query will be asked. The user responds with a Y for an original model NOVA® * computer (vs. 800, 1200 or SUPERNOVA® * computer) or N to this query, depending upon the system he is using.

16. MATRIX OPERATIONS?

    If the user wishes to use matrix operations, he should respond with a Y to this query, if not, respond with a N.

The files SY.RB and BASIC.CL are now created and the system will give the prompt:

R

signifying that the user may now load Extended BASIC.

*NOVA and SUPERNOVA are registered trademarks of Data General Corporation, Southboro, Massachusetts.

Loading Extended BASIC

NOTE: the following discussion does not include configuration information for Multiplexor Handler type 4100. This information is available from your DGC representative.

The Extended BASIC system may be loaded in two ways. After the user has finished answering the queries outlined above and on preceding pages, a file is created called BASIC.CL containing a relocatable load command line corresponding to the responses the user gave to the various queries. The user can then issue the command:

@BASIC.CL@ {filename/L}

to load his Extended BASIC system. If "filename/L" option is used a loadmap of the BASIC system will be appended to file "filename". Alternatively, the user may issue the following RLDR command line to load his Extended BASIC system. The RLDR command line format is:

```
RLDR/N BASIC.SV/S SY [ { MP60 {RD60} }  {TTY}  ↑ )
                     [ { MP26 {RD26} }
                     [ { MP100{RD100} }

{MDSW}
{MDHW}  [ overlay filenames ] BASICA.LB BASICB.LB {SBRTB.} ↑ )
{MDNO}

BASIC{X} .LB {MATX}      BASIC {1} .LB    SYS.LB INT  )
     {Y}     {MATY}            {2}
             {DMAT}            {3}
                               {4}
                               {5}
                               {6}
```

where: Global switch /N should be appended to RLDR; this inhibits a search of the system library SYS.LB. By default, the search will be performed.

The save file to be output is named BASIC.SV, as specified by the /S local switch.

{ } indicates alternate choices.

{ } indicates optional choices.

SY(SY.RB) is the BASIC system configuration module, created by the user using the BASIC system generation routine, 088-000045.

where: (Continued)

MP60 and MP26 are the multiplexor handlers as follows:

MP60.RB (089-000123) Multiplexor handler type 4060
MP26. RB (089-000124) Multiplexor handler type 4026

If there are any dial-up lines in the configuration, each multiplexor handler must be followed by its respective dial-up line handler:

RD60. RB (089-000125) Dial-up for handler type 4060
RD26. RB (089-000126) Dial-up for handler type 4026

TTY (TTY. RB) (089-000135) must be loaded if the user wishes to use the teletypewriter as the master console, also the user must have responded with a Y to query number 8 on page C-5.

MDSW, MDHW, MDNO are alternative multiply/divide options as follows:

MDSW. RB (089-000127)	Software multiply/divide
MDHW. RB (089-000169)	Hardware multiply/divide (SUPERNOVA, NOVA 1200/ 1220, 800/820)
MDNO. RB (089-000170)	Hardware multiply/divide (NOVA)

The relocatable loader creates an overlay file from modules supplied on tape (088-000089). The module names are:
PRU$\left\{\begin{matrix}X\\Y\end{matrix}\right\}$, OINIT, BYE, OMISC, CALL, ERROR, ERIO1, ERIO2, OFILE, XDCD2, ODIR, OPER, OMAC IMUX

Note: The overlay filenames tape also contains some modules for 4100 Multiplexor configurations.

BASICA. LB (099-000044) contains the Extended BASIC compiler common library; and BASICB. LB (099-000045) contains the Extended BASIC Interpreter common library.

SBRTB is an optional user-written binary containing any user subroutines.

The user has the option to load matrix operations or a dummy matrix tape if no matrix operations are required. The dummy tape is DMAT.RB (089-000153).

For users requiring matrix operations, the tape to be loaded is dependent upon whether the configured system contains the hardware floating point option. For systems with the FP option, use MATY.RB (089-000155); and for systems without the FP option, use MATX.RB (089-000154).

BASICX.LB (099-000067) is loaded for systems configured without the hardware floating point option; and BASICY.LB (099-000068) is loaded for systems which are configured with the FP hardware option.

The Extended BASIC libraries are:

BASIC1.LB - single-user Extended BASIC - 099-000050
BASIC2.LB - multi-user Extended BASIC - 099-000051
BASIC3.LB - multi-user swapping Extended BASIC - 099-000052
BASIC4.LB - single-user, mapping Extended BASIC - 099-000066
BASIC5.LB - multi-user, mapping Extended BASIC - 099-000069
BASIC6.LB - multi-user, mapping, swapping, Extended BASIC - 099-00007(

SYSTEM DISK FILES AND DIRECTORIES

Disk Directories

There are two types of disk directories, a library directory and a user directory. The library directory contains a list of all file names which are read-accessible to all users. The user directory contains a list of all file names created and maintained by the user. The files contained within the user directory can be read, written, deleted, and renamed by the user.

The library directory (BASIC.DR) must always be present as either a subdirectory or secondary partition. Each user directory also is either a subdirectory or a secondary partition. After RDOS initialization of the system, the RDOS CLI comm

CDIR name

is used to create a subdirectory called name. The library directory (BASIC.DR) must exist before you can execute BASIC or an error message will occur.

BASIC.ID File

On multi-user Extended BASIC systems, an ASCII disk file called BASIC.ID must be present, containing the account identification of all system users. Each entry in the file must be separated from the next by a carriage return.

A user's account identification is written in the ASCII file in the following format:

aaaa{ a ... a } / dd ... d{ /ssss }

where: aaaa is the user's account identification number (ID). The four characters must be alphanumeric (A to Z and 0 to 9).

{a ... a } an optional password of up to 16 alphanumeric characters.

/dd ... d is the directory name; the directory must exist within the system or the user may not sign-on to the system. Two different users may use the same directory. The system will not permit a second user to sign on with an identification number already in use.

{ /ssss } is the optional name of a SAVEd program existing in the library directory. After a successful sign-on, the SAVEd program will be executed automatically.

When the user is signing on to his system (see Sign-On Procedures, page C - 15) the proper response to the query ACCOUNT-ID: is represented by aaaa{ a ... a }.

This will appear as:

ACCOUNT-ID: XXXXXXX

An X will be printed for each character typed for security reasons. For instance, if the user types 12345ABCD the resultant appearance of the query and response would be:

ACCOUNT-ID: XXXXXXXXX

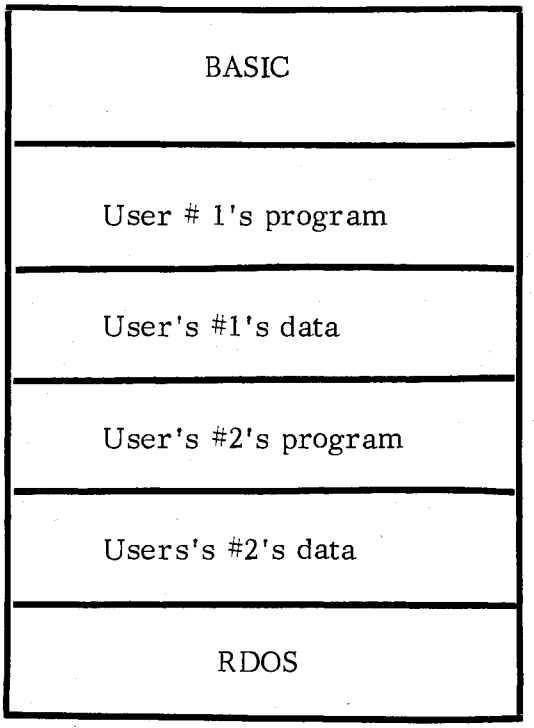
The user must sign-on with exactly the same character sequence as the account identification. For instance, with an account ID of 12AB34CD, a user could not respond with 1234ABCD, or ABCD1234, but must sign on with the proper sequence of 12AB34CD.

BASIC.ID File (Continued)

The system manager can execute a program from the master console which can modify the BASIC.ID file while BASIC is running. In order to do this he must sign onto directory SYS.

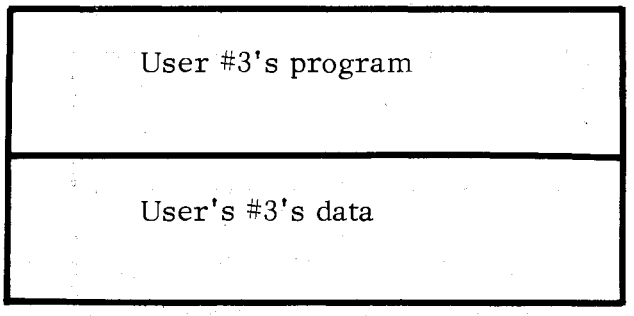
Program Swaps

To describe BASIC program swaps, assume memory to appear as follows:



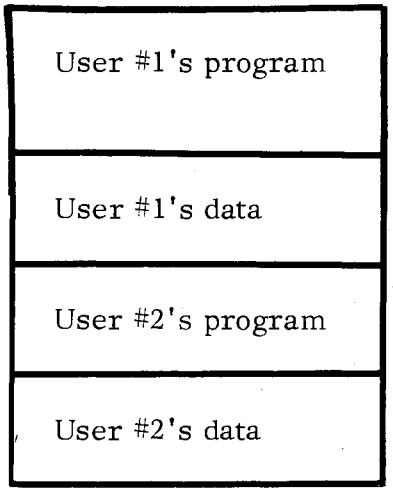
Program Swaps (Continued)

The two programs, belonging to User 1 and User 2, together fit user core exactly. Further suppose a program belongs to User 3:

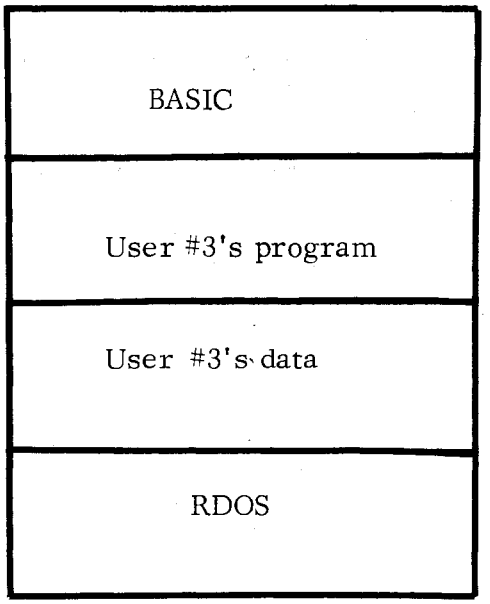


This program is as large as the two current programs. In order for User #3's program to be executed, a program swap must occur. The two current programs, User #1's and User #2's, are swapped out to disk and user #3's program is read in. Memory now can be represented as:

Swapped Programs

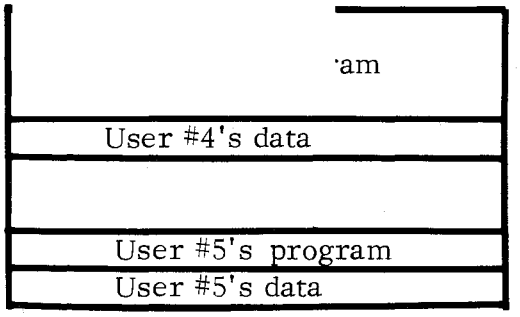


Current Memory Representation



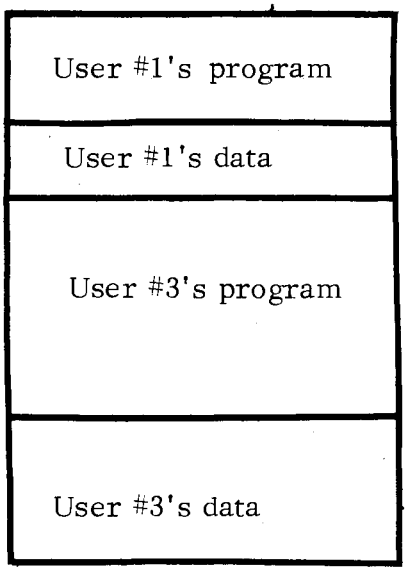
Program Swaps (Continued)

Now assume two more programs:

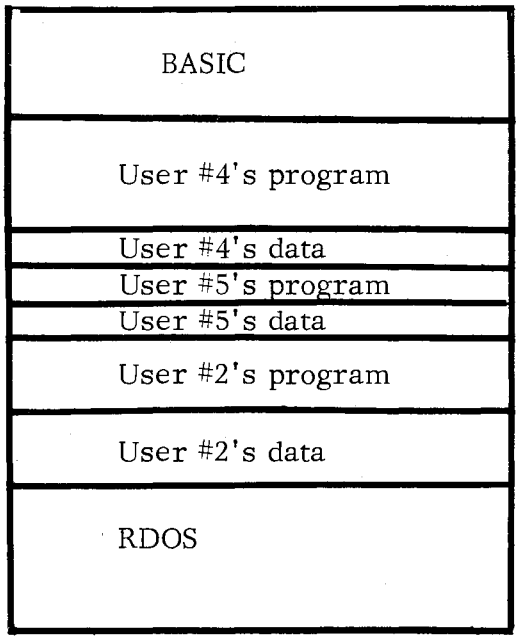


User #3's program will be swapped out to disk, and user #4's and user #5's programs will be brought into core. Also User #2's program will be brought back in since there is room enough for it at this time in core. Memory may then be represented as:

Swapped Programs



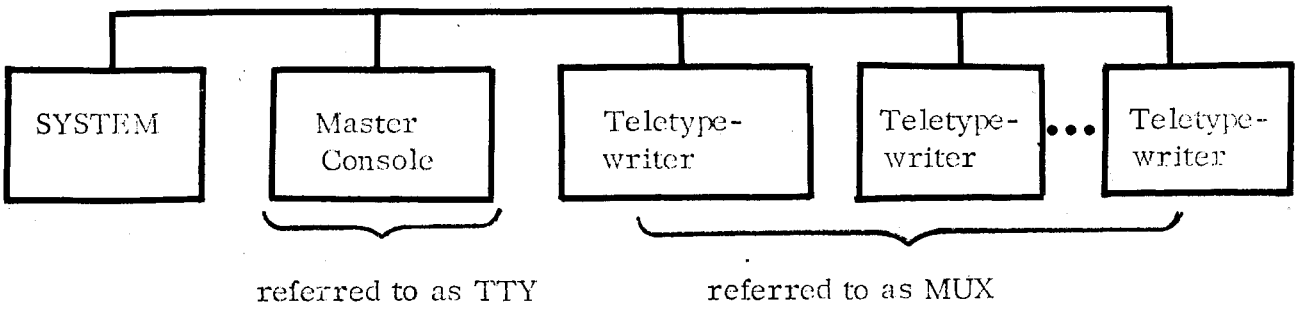
Current Memory Representation



It is important to note that BASIC and RDOS are never swapped, they are fixed portions of memory. As user #2's program illustrates, the smaller a program is the less likely, or the fewer times, it will be swapped. A program, of course, is never swapped if there is no other non-core-resident program awaiting CPU time.

Sign-On/Sign-Off Procedures

Before discussing sign-on procedures, it is necessary to discuss the term master console. The master console is the teletypewriter which is used to configure the RDOS system and the BASIC system. In most cases, the master console is physically the closest teletypewriter to the system; and it is assigned the lowest line number.



All single-user systems, therefore, use the TTY as the master console. Multi-user systems use the TTY if the TTY was configured as part of the system during the configuration process; otherwise, the master console is the teletypewriter having the line number specified during the BSG dialogue (queries 8 and 9 on page C-5).

The sign-on procedure is different depending upon which software system you are going to use, i.e., whether or not the system is multi-user and which teletype-writer is to be used as the master console. Look at the chart on page C-16 for the various sign-on procedures. Note that before signing on to the BASIC system the library directory (BASIC.DR) and BASIC.ID must exist (refer to pages C-10 and C-11) for multi-user systems. BASIC.DR must exist for single-user systems. An error message will be printed if the appropriate user-created file(s) is (are) not present.

In order to kill the BASIC system and return to the CLI (Command Line Interpreter), it is necessary to issue the system command:*

#KILL

The operator command to unlock an escape loop at a user's console logged on under <account ID>is:

#UESC <account ID>

#KILL and #UESC are console commands which may only be issued from the master console.

*NOTE: #indicates a consle system command, which may only be issued from the master console.

MULTI-USER USING TTY AS MASTER CONSOLE		MULTI-USER USING A MUX AS MASTER CONSOLE		SINGLE-USER SYSTEM
TTY	MUX	OTHER MUX LINE	MASTER MUX LINE	
ACCOUNTING? user responds with Y or N. Y = write accounting information to an accounting file named BASIC.AF . N = only write accounting information to terminals. DIRECTORY SPECIFIER: user responds with the name of the directory to be used. This query is repeated until correct response is given. SIGN ON time day line# System prints sign-on information. *(space) System is ready to accept instructions.	The user is informed of the system's ready status by the message: DGC READY user presses ESC ACCOUNT-ID: user responds with his account identification. This query is repeated until correct response is given. SIGN ON time date line# System prints sign-on information. *(space) System is ready to accept instruction.	The user is informed of the system's ready status by the message: DGC READY user presses ESC ACCOUNT-ID: user responds with his account identification. This query is repeated until correct response is given. SIGN ON time date line# System prints sign-on information. *(space) System is ready to accept instructions.	ACCOUNTING? user responds with Y or N. Y = write accounting information to an accounting file named BASIC.AF. N = only write accounting information to terminals. DIRECTORY SPECIFIER: user responds with the name of the directory to be used. This query is repeated until correct response is given. SIGN ON time date line# System prints sign-on information. *(space) System is ready to accept instructions.	ACCOUNTING? user responds with Y or N. Y = write accounting information to an accounting file named BASIC.AF. N = only write accounting information to terminals. DIRECTORY SPECIFIER: user responds with the name of the directory to be used. This query is repeated until correct response is given. SIGN ON time date line# System prints sign-on information. *(space) System is ready to accept instructions.

BATCH OPERATION

BATCH is an RDOS program that permits system software to be dedicated to the processing of a job stream without operator intervention. The BATCH configuration is single-user; when BASIC is configured for BATCH using BSG, all multi-user queries are skipped (see page C-4).

BATCH initiates the execution of one or more job files, making up a job stream. Each job file is an input file containing one or more user jobs, and each job is complete with job control commands and optional data sets.

The BATCH commands begin with an exclamation point. The BATCH command to execute BASIC is !BASIC. In using BASIC under BATCH, the user writes a series of BASIC commands as if he were inputting them from the console.

To replace non-BATCH terminal operation with BATCH, the user needs an input file, an output file, and a log file. The default assignment of the log file is the console terminal printer; the default assignment of the input and output files (called SYSIN and SYSOUT respectively) are the card reader and the line printer. The !BASIC command has a /T switch permitting SYSIN to be read in mark-sense card format rather than in 80-column punch card format (see Appendix E).

The Extended BASIC user may override the default input and output file assignments in his BATCH job input file (which may be a disk file). The procedure is described in the BATCH User's Manual, 093-000087, which the user should consult before attempting to use BATCH.

Extended BASIC commands under BATCH must not attempt any input/output to unit record devices assigned to SYSIN or SYSOUT. For example, the following are illegal commands while $CDR is defined as SYSIN and $LPT is defined as SYSOUT:

```
ENTER "$CDR"

LIST "$LPT"
```

An example of a BATCH job stream containing Extended BASIC commands is given on the following page.

Job Stream	Comments
!JOB BASIC	Job identification.
!DISK	RDOS commands DISK and LIST.
!LIST/E BASIC. -	
!BASIC	Execute BASIC.
N	No accounting file required.
BASIC	Response to "DIRECTORY SPECIFIER"
SIZE	BASIC commands.
PAGE = 80	
ENTER "FACTOR.SR"	
125OPERATOR	Illegal command.
LIST	BASIC commands.
RUN	
1234E66	Input data.
1234	
7474	
0	
SIZE	BASIC command.
#KILL	Exit BASIC and return to BATCH command mode.
!EOF	End of BATCH job file.

Note the use of the command #KILL. In non-BATCH operation, this returns you to the CLI and in BATCH operation to the BATCH monitor. See page C-15.

BASIC.AF FILE

Each time Extended BASIC is initialized the query

ACCOUNTING?

appears on the master console. The system manager responds "Y" if he wants accounting information to be recorded. Any other response indicates that accounting information need not be recorded. When Accounting information is to be recorded a sequential file, BASIC.AF, is created, if it does not already exist. In any case, all information written to BASIC.AF is appended to the end of the file. This means that the file will continue to grow as BASIC log-ons or BYE's are executed. A system manager should process this file every so often, so as not to allow it to use up too much disk space. Suggestions on how to do this is given below following the description of entries in the BASIC.AF file.

An entry is made in BASIC.AF by writing one line of ASCII characters at a time. Lines of information for various system users can be interleaved. However, each line of information has a precise meaning and identifies a user uniquely. Therefore it is easy to fully recover the accounting information stored.

General format of a line in BASIC.AF

aaaa mm/dd/yy HH:MM { SIGN-ON, NN / SIGN-OFF, NN / CPU-USED, t / I/O-USED, s }

aaaa = user Identification number (4 characters) 4 asterisks on master console (****).
mm = month (e.g., 03)
dd = day (e.g., 23)
yy = year (e.g., 74)
HH = hour of day (24 hour clock)
MM = minute
NN = port number (e.g., 03 · -1 = master console)
t = integer number of seconds of CPU time used since the previous log-on for this port
s = integer; measure related to amount of I/O performed (No. of RDOS .SYSTM calls)

An example of a segment of a BASIC.AF file follows.

BASIC.AF FILE (Continued)

```
****    03/15/74   15:53   SIGN-ON,    -1
0003    03/15/74   15:57   SIGN-ON,    01
0002    03/15/74   16:05   SIGN-ON,    02
0001    03/15/74   16:35   SIGN-ON,    00
0001    03/15/74   17:55   SIGN-OFF,   00
0001    03/15/74   17:55   CPU-USED,   237
0003    03/15/74   17:55   SIGN-OFF,   01
0001    03/15/74   17:55   I/O-USED,   556
0003    03/15/74   17:55   CPU-USED,   272
0002    03/15/74   17:55   SIGN-OFF,   02
0003    03/15/74   17:55   I/O-USED,   365
0002    03/15/74   17:55   CPU-USED,   40
0002    03/15/74   17:55   I/O-USED,   417
0002    03/15/74   17:56   SIGN-ON,    03
0002    03/15/74   17:56   SIGN-OFF,   03
0002    03/15/74   17:56   CPU-USED,   11
0002    03/15/74   17:56   I/O-USED,   40
****    03/15/74   17:56   SIGN-OFF,   -1
****    03/15/74   17:56   CPU-USED,   3164
****    03/15/74   17:56   I/O-USED,   1662
```

Information in the Accounting file can be read by a BASIC program which may be executed at the master console. Since the Accounting file is opened and closed for each line of data written to it, a system program can be written which RENAMEs this file and then opens a (new) file called BASIC.AF. In this way the system manager can periodically remove old Accounting information for processing, even while the BASIC system remains operational. Since the information is pure ASCII text, it is easy to write a BASIC program to process the data. The space delimiter between each field makes the sub-entries easily accessible.

SUMMARY OF DEDICATED FILE NAMES USED BY BASIC:

				Requirement
.	BASIC.CL SY.RB	-	Created by system generation program BSG.SV	All systems.
*	BASIC.SV	-	System save file	All systems.
*	BASIC.OL	-	System overlay file	
*	BASIC.DR	-	Library directory	All systems.
*	BASIC.ID	-	Account Identification file	Any M.U. system.
	BASIC.AF	-	Accounting File	Option in all systems
	BASIC.SW	-	Swapping File (contiguous)	Created by BASIC initialization for swapping system

*Asterisks indicates must be set up by system manager before BASIC can be successfully executed.

APPENDIX D

EXTENDED BASIC OPERATION UNDER SOS

INTRODUCTION

A stand-alone version of Extended BASIC is available for use on machines not having a direct access I/O device. This appendix describes the generation and operation of such a system. Although these systems differ only slightly from their RDOS BASIC counterparts, a table of differences is included in this appendix.

Persons concerned with generating non-disk Extended BASIC systems should be familiar with the operation of the machine and the concepts of the Stand-alone Operating System (SOS) as described in STAND-ALONE OPERATING SYSTEM USER'S MANUAL, 093-000062.

STARTER SYSTEM

A small starter system, capable of running in 12K words of core storage is available. This system is configured at the factory and contains the following features:

1. A device driver for the console terminal
2. A device driver for an 80 column line printer
3. A device driver for a high speed paper tape reader
4. A device driver for a high speed paper tape punch
5. PRINT USING capability
6. Matrix manipulation routines

This absolute binary (core image) paper tape can be loaded with the Binary Loader, as described in the appropriate manual, 093-000003. When loading has been completed, the system will halt. Press CONTINUE, enter the current date and time, and the system is ready to accept input.

BUILDING EXTENDED BASIC

For machines having at least 16K words of storage, Extended BASIC may be tailored to support a number of different hardware configurations. The process of configuring a system consists of the following:

1. Producing a trigger which specifies the desired I/O support and program features.

2. Performing a relocatable load of the trigger, the appropriate SOS libraries and the BASIC libraries and relocatable binaries.

3. Performing a run-time system generation to further tailor the system.

CREATING A TRIGGER

Triggers are produced by the SOS SYSGEN program. This program accepts a command line which contains device driver ENTRY symbols from the console device. It outputs a relocatable binary file (the trigger) which is comprised of EXTERNAL NORMAL symbols corresponding to the named device drivers. These EXTERNAL NORMALs cause the selection or "triggering" of the desired routines for loading when the trigger precedes the SOS libraries as input to the relocatable loader.

The first step to create a trigger is to load and start the SYSGEN program. This can be done by using the binary loader to load an absolute binary SYSGEN paper tape (091-000070, 091-000071, 091-000074). SYSGEN can be loaded from cassette or magnetic tape using the Core Image Loader/Writer.

When the SYSGEN program is started, it outputs the prompt:

```
SYSG
```

and waits for a command line response. The command line has the format:

```
(SYSG) driver1 ...drivern .RDSI output-device/O BTRIG/T
```

where:

$driver_1 \ldots driver_n$	is a list of entry symbols in the desired device driver routines. Table D-1 lists all possible symbols.
output-device	is the name of the device to which the trigger is to be output. This name must be followed by the "/O" switch.
BTRIG/T	assigns the title BTRIG to the trigger.

Table D-1 Driver Entry Symbols for SYSGEN Command Line

Driver Entry Symbol	SOS Program Name	Function
.CDRD	CDRDR	card reader driver (includes mark sense)
.CTAD	CTADR	cassette driver for unit 0
.CTU1	CTU1	control table/buffer for cassette units 0-1
.	.	
.	.	
.	.	
.CTU7	CTU7	control table/buffer for cassette units 0-7
.L132	LP132	132 column line printer driver
.LPTD	LPTDR	80 column line printer driver
.MTAD	MTADR	magnetic tape driver for unit 0
.MTU1	MTU1	control table/buffer for magnetic tape units 0-1
.	.	
.	.	
.	.	
.MTU7	MTU7	control table/buffer for magnetic tape units 0-7
.PLTD	PLTDR	plotter driver (access via CALL)
.PTPD	PTPDR	high speed paper tape punch driver
.PTRD	PTRDR	high speed paper tape reader driver
.TTI1	TTY1	second console teletype (TTI1, TTO1)

CREATING A TRIGGER (Continued)

For example, to produce a trigger for the following devices

paper tape reader
paper tape punch
132 column line printer
mark sense card reader

one would respond to the SYSG prompt with

.PTRD .PTPD .L132 .CDRD .RDSI $PTP/O BTRIG/T)

PERFORMING A RELOCATABLE LOAD (PAPER TAPE)

Once a trigger has been created and saved on an external device, the following steps must be followed to produce an absolute binary tape.

1. Using the Binary Loader, load the Extended Relocatable Loader, 091-000038.

2. Mount the trigger in the Teletype®* reader and type 1 or, in the high speed reader, type 2.

3. Set the switch register to 1ØØØ, Type in 3.

4. If the trigger specifies support for cassette or magnetic tape drivers, mount either the SOS Cassette Library (099-000041) or the SOS Magnetic Tape Library (099-000042) in the Teletype reader and type 1, or in the high speed reader and type 2.

   Mount the SOS Library (099-000010) in the Teletype reader and type 1, or in the high speed reader, type 2.

5. Mount the relocatable binaries and libraries in the order shown in Table D-2 in the Teletype reader and type 1, or in the high speed reader, type 2.

*Teletype is a registered trademark of Teletype Corporation, Skokie, Illinois.

Tape Name	Supplied As	Purpose	Comment
MP.RB	089-000137	driver for system console	Choose One
MP26.RB	089-000141	driver for system console and 4026 multiplexer	
MP60.RB	089-000140	driver for system console and 4060 multiplexer	
MDSW.RB	089-000156	standard multiply/divide routines (all machines)	Choose One
MDHW.RB	089-000157	multiply/divide routines for machines having options 8007, 8107, 8207, 8307	
MDNO.RB	089-000158	multiple/divide routines for machines having option 4031	
MSCR.RB	089-000	translator for mark sense card (N96829)	May be omitted
BASICA.LB	099-000046	compiler routines	Required
BASICB.LB	099-000047	interpreter routines	Required
SBRTB.RB	user-written	CALL resolutions (see Appendix B)	May be omitted
MAT.RB	089-000138	routines to perform matrix functions	May be omitted
PRU.RB	089-000139	routines to perform PRINT USING functions	May be omitted
BASIC7.LB	099-000048	single user library	Choose One
BASIC8.LB	099-000049	multi-user library	

Table D-2 Extended BASIC Supplied Paper Tapes

PERFORMING A RELOCATABLE LOAD (PAPER TAPE)

6. Type 5 and note the value of NMAX output by the relocatable loader on the Teletype; this number will be used in step 12.

7. Mount the relocatable binary punch program (089-000080) on the Teletype reader and type 1, or on the high speed paper tape reader and type 2.

8. Type 6 and note the value of RBFP output by the relocatable loader on the Teletype; this number will be used in step 10.

9. Type 8 to terminate the loading process.

10. Enter RBFP (from step 8 into the data switches on the computer console, press RESET then START.

11. Type 0H for output on the Teletype punch or 1H for output on the high speed punch.

12. Type 1, nmaxP where nmax is the value of NMAX noted in step 6.

13. Type 377E, to specify a starting address for the program.

The paper tape output from this procedure can now be loaded by using the binary loader.

PERFORMING A RELOCATABLE LOAD (MAGNETIC TAPE OR CASSETTE)

SOS users with a magnetic tape or cassette may create a SAVE BASIC file by using the SOS CLI command RLDR (see SOS User's Manual, Chapter 3). The order of input of the relocatable binaries is the same as for paper tapes. The tape file number for each binary can be found on the keysheet supplied with each system.

PERFORMING A RELOCATABLE LOAD (RDOS SYSTEMS)

The entire BASIC system can be built on RDOS disk-based systems and subsequently transferred to non-disk systems by following this procedure:

1. Create a trigger source file:

```
XFER/A $TTI BTRIG.SR
.TITLE    BTRIG
.COMM     TASK,0
.EXTN     .RDSI
.EXTN device
   .
   .
   .                    ;SOS device drivers from Table D-1
.END
↑Z
```

2. Assemble the trigger:

```
MAC BTRIG
```

3. Transfer the relocatable binaries from tape to disk:

```
XFER $PTR SOS.LB
                        or
XFER MT0:0 SOS.LB
  .
  .
  .
```

4. Perform a relocatable load:

```
RLDR/Z/N $LPT/L BASIC/S BTRIG 1000/N ↑)
SOSMT.LB SOSCT.LB SOS.LB ↑)
 MP60 MSCR MDSW BASICA.LB BASICB.LB MAT PRU BASIC8.LB )
```

5. Test the resultant system:

```
BOOT BASIC
```

   Note: RDOS must be re-initialized after this test.

6. Make a core image file for loading on the machine without a disk:

```
MKABS/Z/S BASIC $PTP
```

SYSTEM DIALOGUE

Once the core image file has been created and loaded, BASIC starts automatically and identifies itself:

BASIC REVISION X.X MM/DD/YY

where:

X.X represents the revision level and should be noted on all correspondence with DGC.
MM/DD/YY represents the date that system testing at the factory was completed.

At this point, a single error message may occur:

INCOMPATIBLE OPERATING SYSTEM

signifying that the SOS.LB is Revision 8 or earlier.

It is possible to configure several different BASIC systems and to save each one on an external medium such as paper tape. The configuration process is termed BASIC SYSGEN and is described in the following paragraphs.

SYSGEN restart may be accomplished by pressing the ESC key at any time.

LINE CONFIGURATION: (multi-user systems only)

Required response is a list of terminal line numbers, a range of terminal line numbers or both. The 4026 multiplexer is capable of addressing 16 lines (0-15) and the 4060 multiplexer can have a maximum 32 lines (0-31). The maximum number of working terminals which can be serviced by Extended BASIC is 33 (32 multiplexer lines and a system console). Line numbers may be individually separated by commas and ranges specified by a dash:

0,1,2,4	- lines 0, 1, 2 and 4
3-7, 10, 12-13, 17	- lines 3, 4, 5, 6, 7, 10, 12, 13 and 17
0-31	- lines 0 through 31 inclusive

DIAL-UP LINE CONFIGURATION: (multi-user systems only)

Required response is in the same format as described above. Dial-up lines must be a subset of those specified in the preceding example. (A carriage return specifies no dial-up lines.)

RESERVED FILE NAMES:

Reserved File Name	Device
$CDR	[mark sense] card reader
CT0:	cassette unit 0
CT1:	cassette unit 1
CT2:	cassette unit 2
CT3:	cassette unit 3
CT4:	cassette unit 4
CT5:	cassette unit 5
CT6:	cassette unit 6
CT7:	cassette unit 7
$LPT	line printer
MT0:	magnetic tape unit 0
MT1:	magnetic tape unit 1
MT2:	magnetic tape unit 2
MT3:	magnetic tape unit 3
MT4:	magnetic tape unit 4
MT5:	magnetic tape unit 5
MT6:	magnetic tape unit 6
MT7:	magnetic tape unit 7
$PLT	plotter
$PTP	paper tape punch
$PTR	paper tape reader
TTI1	second Teletype keyboard
TTO1	second Teletype printer

Table D-3 Reserved File Names

SYSTEM DIALOGUE (Continued)

Required response is a list of reserved file names taken from table D-3. In order to later access any device in the list, be sure that the appropriate driver was included in the trigger, BTRIG.RB. The list is terminated with a carriage return.

Note that $TTI, $TTO, $TTR and $TTP are not in the list. These devices may be accessed by the ENTER, LIST, PUNCH, PRINT and INPUT keyboard commands.

ERROR MESSAGE TEXT?

Responses include Y (YES), carriage return (YES) or anything else (NO). A Y response will cause a brief description of the error to be appended to each error message.

BASIC SYSGEN is now complete and the machine will halt. The entire configured system may be saved by invoking the Relocatable Binary Punch program, RBFP. Locate the symbol RBFP on the load map, enter the corresponding address in the console switches and press RESET and START. Otherwise, press CONTINUE.

DATE AND TIME

The system operator must now enter the date:

DATE: MM-DD-YY

and the time:

TIME: HH:MM

in 24 hour notation.

SIGN-ON

The system console is now activated:

```
10/01/74  15:33  SIGN-ON, SC
*
```

SYSTEM DIALOGUE (Continued)

The system now attempt to allocate a minimum portion of 1024 bytes for each user. An error message,

```
NO CORE
```

indicates that insufficient storage is available. A new system must be generated with either fewer users, fewer features or both. The multiplexer terminals may be activated by pressing the ESC key once:

```
10/01/74  15:34  SIGN-ON, 0
*
```

RESTART

The system may be stopped at any time and restarted at location 377. A NEW is automatically performed for each user.

POWER FAIL/AUTO RESTART

For machines equipped with this optional feature, system status is preserved upon detection of a power failure and the system comes to an orderly halt. When power is restored, the system will restart if the power switch on the console is in the LOCK position. If not, the system must be manually restarted at location 0. All user files will then be closed and each user must press ESC to activate his terminal.

```
POWER FAIL [AT nnnn ]
*
```

will then be printed at each terminal, where nnnn is the statement which was being executed when the power failure occurred. User programs remain intact.

Table D-4

Error Messages Initiated by BASIC

Code	Meaning	Comment
——	INCOMPATIBLE OPERATING SYSTEM	use SOS rev 9
——	NO CORE	system too large for available core
00	ARITHMETIC OPERATORS IN ILLEGAL COMBINATION	
01	INVALID CHARACTER	
02	SYNTAX	
03	[MAT] READ/DATA TYPES INCONSISTENT	
04	INTERNAL SYSTEM FAULT	
05	INVALID STATEMENT NUMBER	
06	ATTEMPT TO DEFINE MORE THAN 93 VARIABLES	
07	ILLEGAL COMMAND (FROM A FILE)	
08	PAGE OR TAB SPECIFICATION ILLEGAL	
09	ILLEGAL RESERVED FILE NAME	
10	RESERVED FILE IN USE	
11	PARENTHESES NOT PAIRED	
12	ILLEGAL COMMAND	
13	STATEMENT NUMBER MISSING	
14	INSUFFICIENT STORAGE TO ENTER STATEMENT	
15	UNSATISFIED [MAT] READ	
16	ARITHMETIC OVERFLOW, UNDERFLOW OR DIVIDE BY ZERO	
17	UNDEFINED VARIABLE	
18	GOSUB NESTING LIMIT	
19	RETURN - NO GOSUB	
20	FOR NESTING LIMIT	
21	FOR - NO NEXT	
22	NEXT - NO FOR	
23	INSUFFICIENT STORAGE FOR A VARIABLE OR AN ARRAY	
24	LINE NUMBER MISSING	
25	MAT OR PRU NOT IN SYSTEM	
26	INSUFFICIENT STORAGE TO LOAD SAVE FILE	
27	INVALID FILE REFERENCE	
28	ARRAY EXCEEDS INITIAL DIMENSION	
29	EXPRESSION TOO COMPLEX FOR EVALUATION	
30	INVALID FILE MODE	
31	SUBSCRIPT EXCEEDS DIMENSION	
32	UNDEFINED USER FUNCTION	
33	FUNCTION NESTING LIMIT	
34	FUNCTION ARGUMENT	
35	ILLEGAL EDIT MASK	
36	PRINT LINE GREATER THAN PAGE WIDTH	
37	USER SUBROUTINE (SBRTB) NOT FOUND	
38	UNDIMENSIONED STRING	
39	REDUNDANT MATRIX SPECIFICATION	
40	MATRICES UNEQUAL SIZES	
41	MATRIX HAS ONLY ONE DIMENSION	
42	FILE ALREADY OPENED	
43	MATRIX NOT SQUARE	
44	FILE NOT OPEN	
45	NOT A SAVE FILE	
46	INCORRECT RESPONSE TO [MAT] INPUT	
47	FILE OPENED IN WRONG MODE	

Table D-5 Error Messages Initiated by SOS

00	ILLEGAL CHANNEL
01	ILLEGAL FILE NAME
02	ILLEGAL SYSTEM COMMAND
03	ILLEGAL COMMAND FOR DEVICE
06	END OF FILE
07	READ-PROTECTED FILE
08	WRITE-PROTECTED FILE
10	NON-EXISTENT FILE
13	FILE NOT OPEN
17	CHANNEL IN USE
18	RECORD SIZE EXCEEDED
20	PARITY
22	STORAGE ALLOCATION
24	FILE DATA CHECK
25	UNIT IMPROPERLY SELECTED
30	ILLEGAL DEVICE CODE
33	ILLEGAL TIME OR DATE
37	INTERRUPT DEVICE CODE IN USE

Table D-6 Differences Between RDOS BASIC and SOS BASIC

SOS Extended BASIC

- has no MMPU support
- has no FPU support
- has no disk support (swapping, accounting, BATCH, etc.)
- requires that the system console be assigned to the same Teletype as the SOS CLI console.
- has no account names
- refers to system console line number as 'SC' in sign-on and sign-off messages
- has no LIBRARY, FILES or WHATS commands
- needs no record delimiter other than CR on Teletype paper tapes
- supports power fail on multi-user systems
- has manual restart capability
- does not print 'DGC READY' at each terminal on start-up
- has a time slice of 16/100 seconds (16/110 for 4026 systems)
- needs no real time clock for 4026 systems
- runs real time clock at 100 hertz
- prints optional error message text
- has no SYS(9) or SYS(10)
- has no ON ERR THEN INIT
- has different error codes

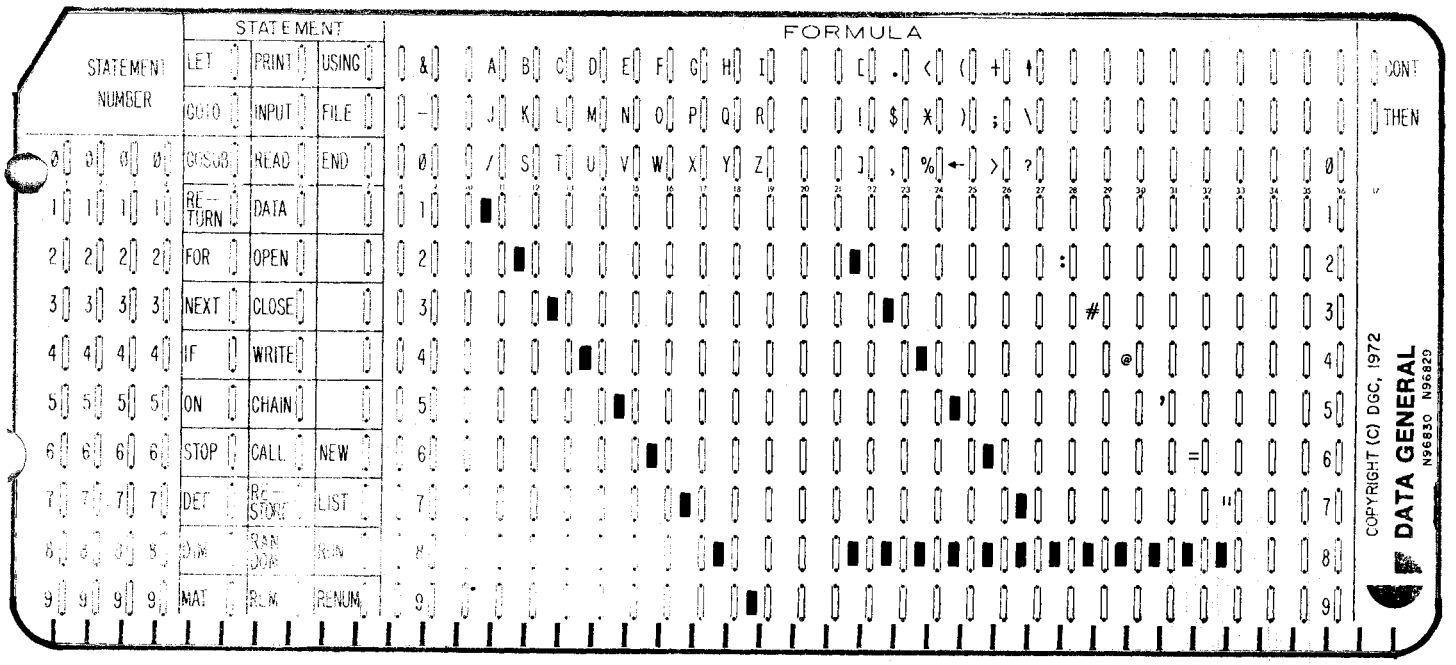
Table D-7 BASIC Core Requirements in Bytes

REQUIRED COMPONENTS		
SOS NUCLEUS (SOS.LB WITH $PTR, $PTP, $LPT)		4846
SYSTEM STORAGE		512
BASIC NUCLEUS (BASICA.LB, BASICB.LB)		13548
INTERACTIVE DRIVERS		
SINGLE USER (BASIC7.LB, MP.RB)	3594	
4060 MULTI-USER (BASIC8.LB, MP60.RB)	3642	
4026 MULTI-USER (BASIC8.LB, MP26.RB)	3694	
USER TABLES (ONE REQUIRED FOR EACH USER)	204	
MULTIPLE/DIVIDE ROUTINES		
STANDARD (MDSW.RB)	54	
8007, 8107, 8207, 8307 HARDWARE (MDHW.RB)	12	
4031 HARDWARE (MDNO.RB)	36	
OPTIONAL COMPONENTS		
SOS MAGNETIC TAPE DRIVER (SOSMT.LB)	5170	
SOS CASSETTE TAPE DRIVER (SOSCT.LB)	5170	
MARK SENSE CARD TRANSLATOR (MSCR.RB)	550	
MATRIX MANIPULATOR (MAT.RB)	1550	
PRINT USING ROUTINES (PRU.RB)	978	
USER-WRITTEN SUBROUTINES (SBRTB.RB)	---	

APPENDIX E

PROGRAMMING ON MARK-SENSE CARDS

Source programs may be written on Data General's Extended BASIC mark-sense programming cards for input to the mark sense card reader. The Data General Extended BASIC mark-sense programming card is a 37-column card as shown below.

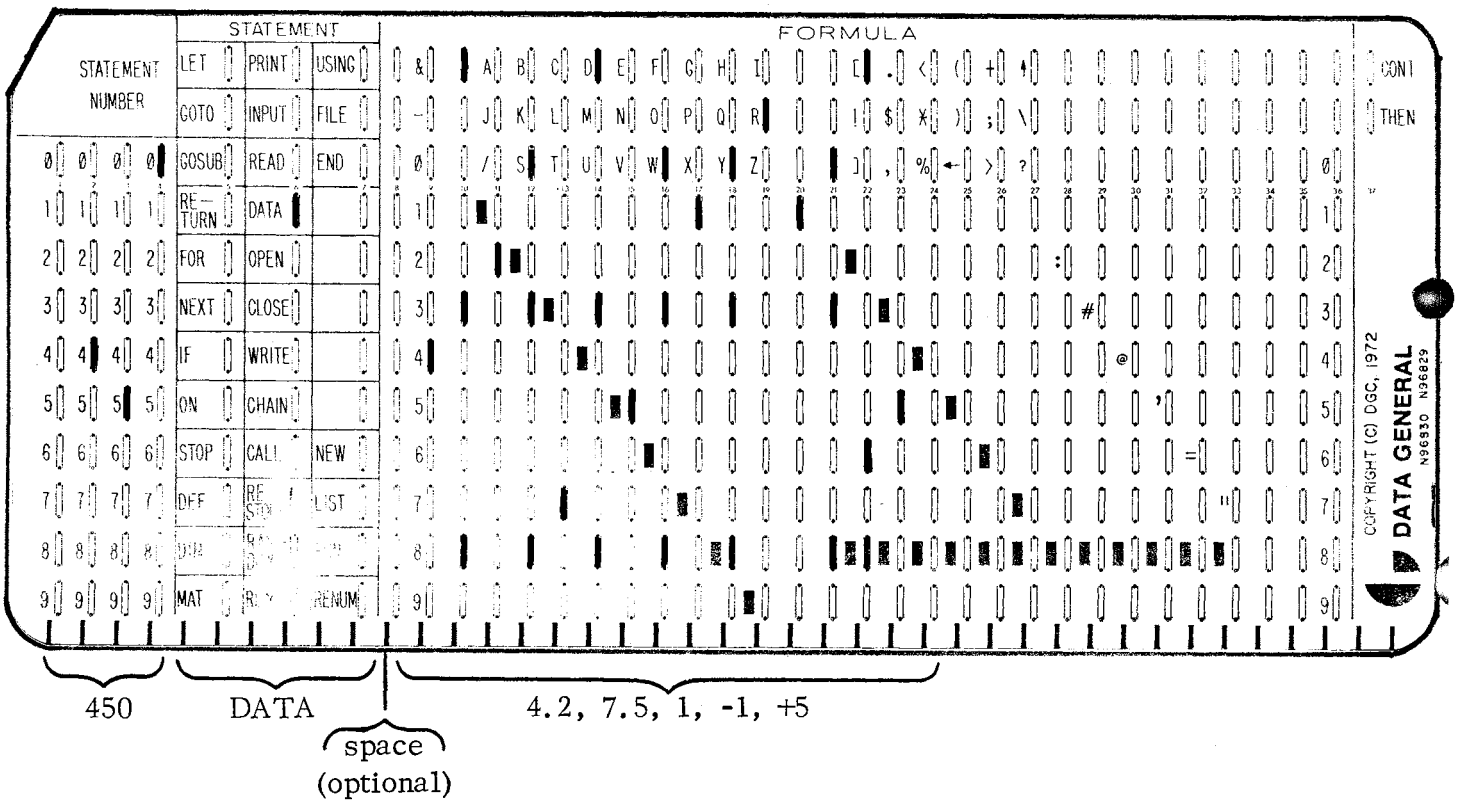


A stack of mark-sense cards may be read as a file and requires an EOF card at the end, which is a card with a single column in which all rows are marked. When in Batch mode, entire jobs may be entered from the card reader. Such card decks must conform to the Batch job control formats. Keyboard commands are suitable card input in Batch mode only.

The mark-sense reader has an option that permits either markings or punches to be read. Users having the option may punch mark-sense cards. Marked and punched cards may be intermixed in a deck and a single card may contain both markings and punches.

The BASIC statement field of the mark-sense card, as shown above, is three columns which allows all possible combinations of statement keywords. For example, MAT appears in the first column of the field, WRITE in the second, and FILE in the third, permitting the user to indicate a MAT WRITE FILE statement.

A single Extended BASIC statement or part of a statement may be written on a single card. Cards are marked with No. 2 pencil in the appropriate column; for example, the statement 450 DATA 4.2, 7.5, 1, -1, +5 would appear as:



Part of a statement may be written on one card and continued on the next by marking the CONT box in the upper righthand corner of the first card and continuing the statement on the card following beginning with the FORMULA section.

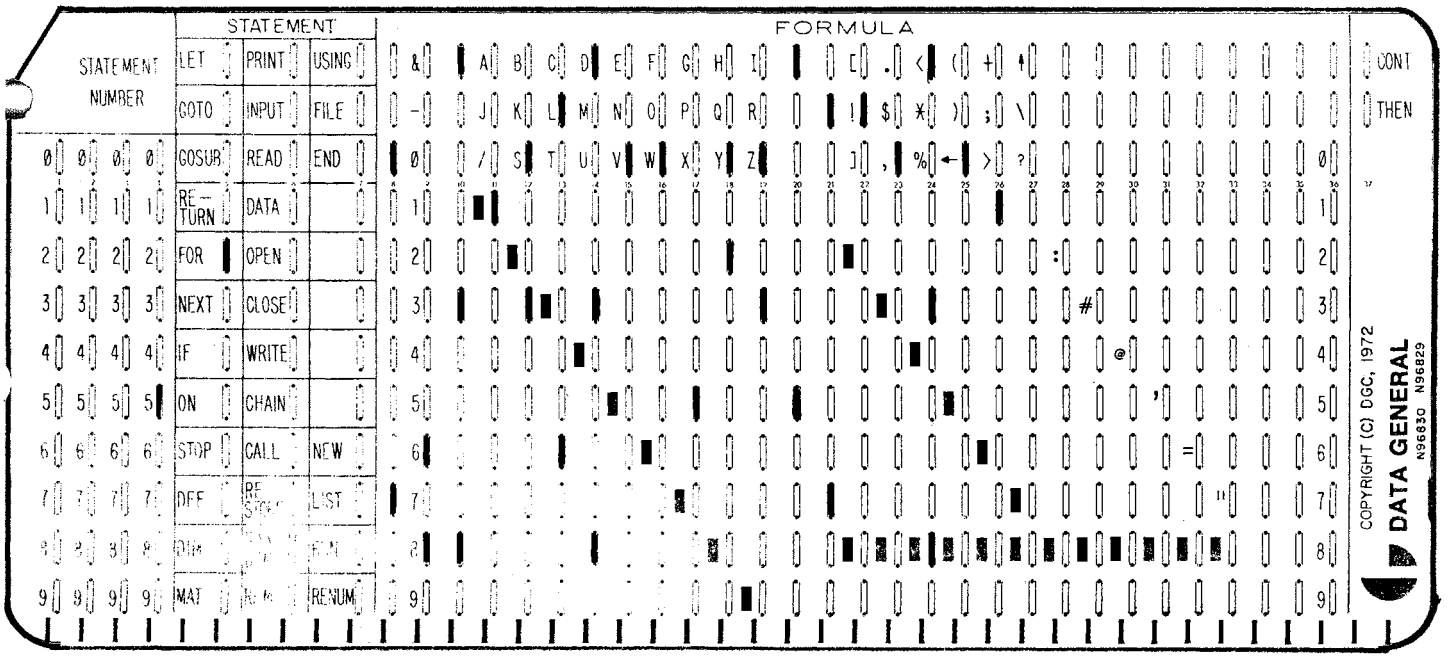
When writing an IF or an ON statement, the programmer writes one card containing the IF or ON expression and marks the THEN box in the upper righthand corner of card. The programmer continues the THEN clause on the next card beginning in the FORMULA section.

The FORMULA section of each card must be filled out in Hollerith code, and programmers familiar with punched cards will have no difficulty with the format. To assist any programmers who are not familiar with punched cards, each card contains a key indicating the lines that must be marked for each character. For each character, a box on the horizontal line on which the character appears must be marked. In addition, in the horizontal lines immediately below the character, a line must be marked if a square appears on the lefthand side. If we use punched card notation, the top line is designated 12, the second from the top is designated 11, and the other

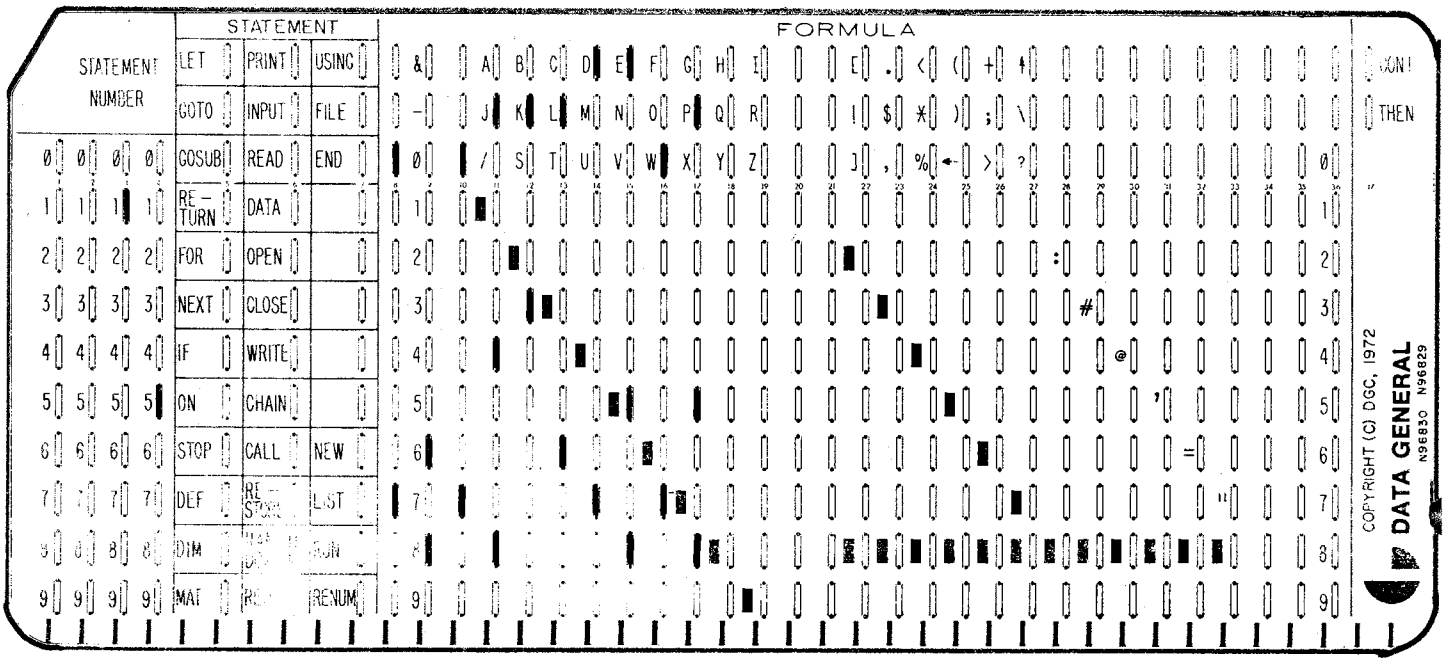
lines are numbered from 0 through 9. To indicate 4, put a mark on line 4; to indicate *, put marks on lines 11-4-8; to indicate #, put marks on lines 3-8, etc.

As a further example of use of mark-sense cards, the following source code is shown on cards:

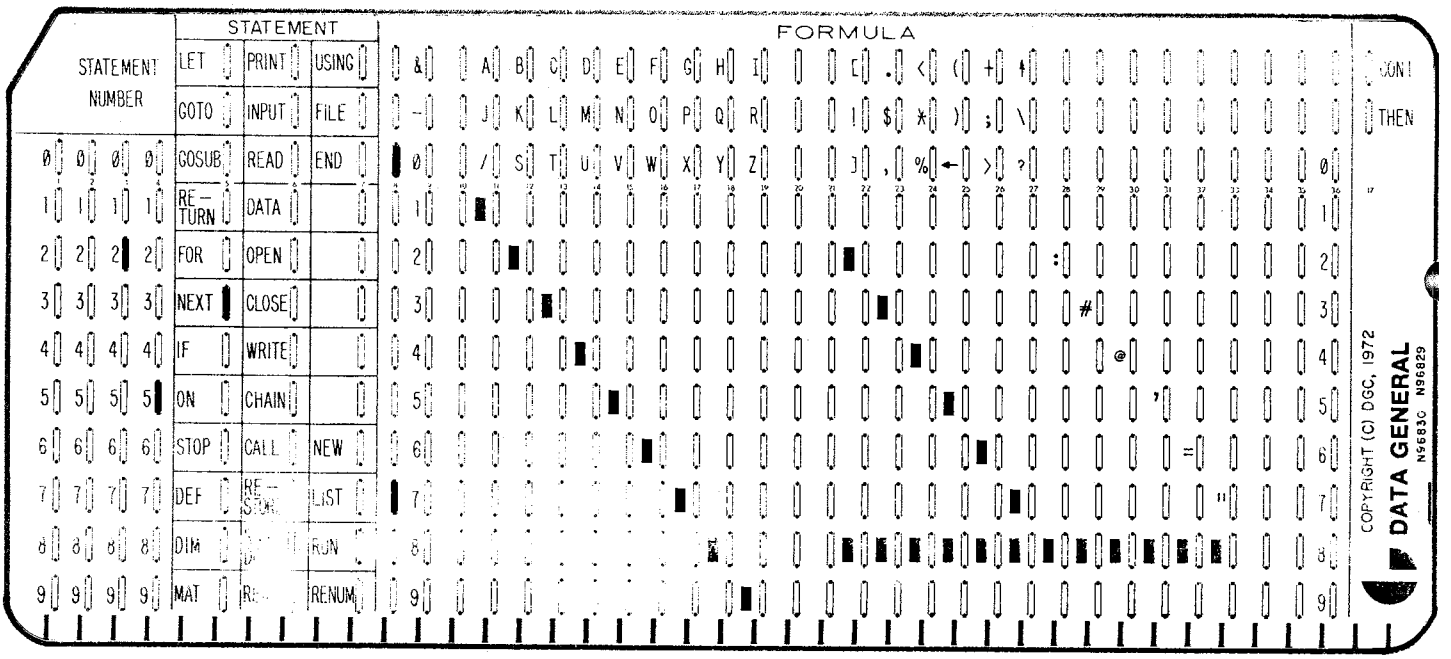
```
 5  FOR X = .1 TO .005 STEP -0.01
15  X = X * LOG(X)
25  NEXT X
```



5 FOR X=.1 TO .005 STEP -0.01



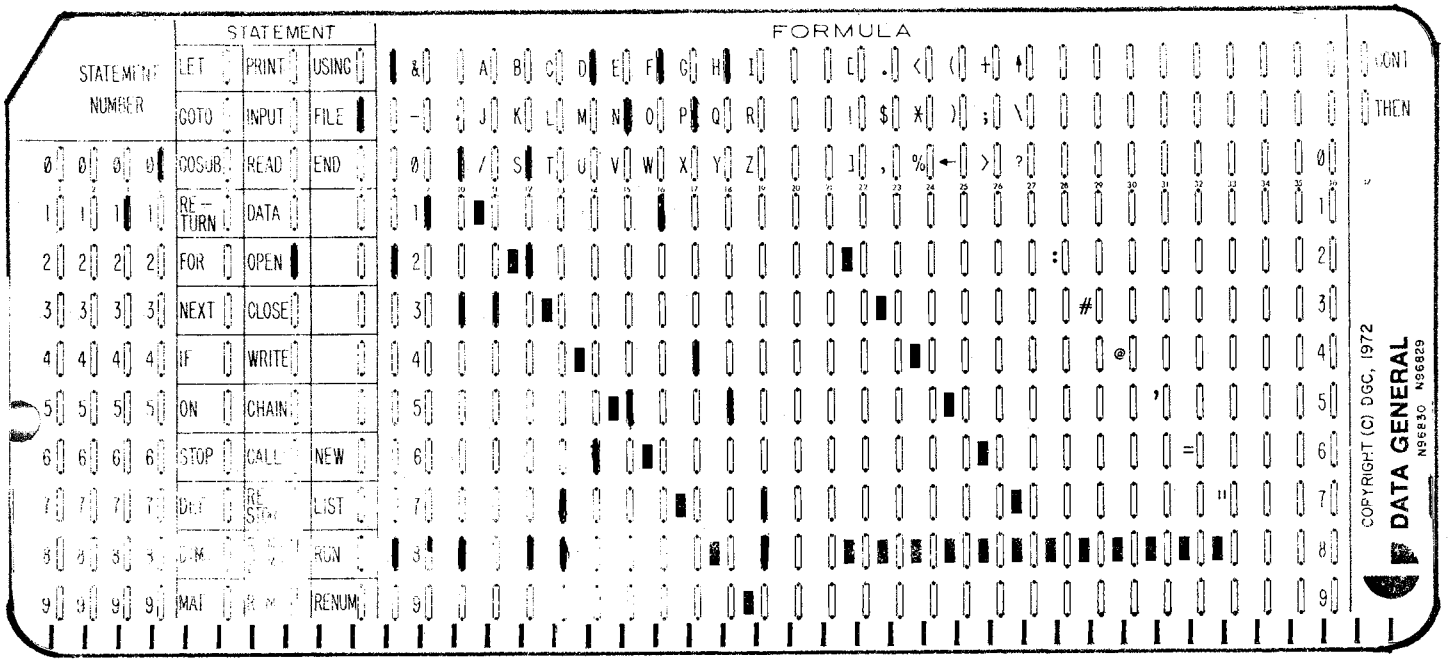
15 X = X * LOG(X)



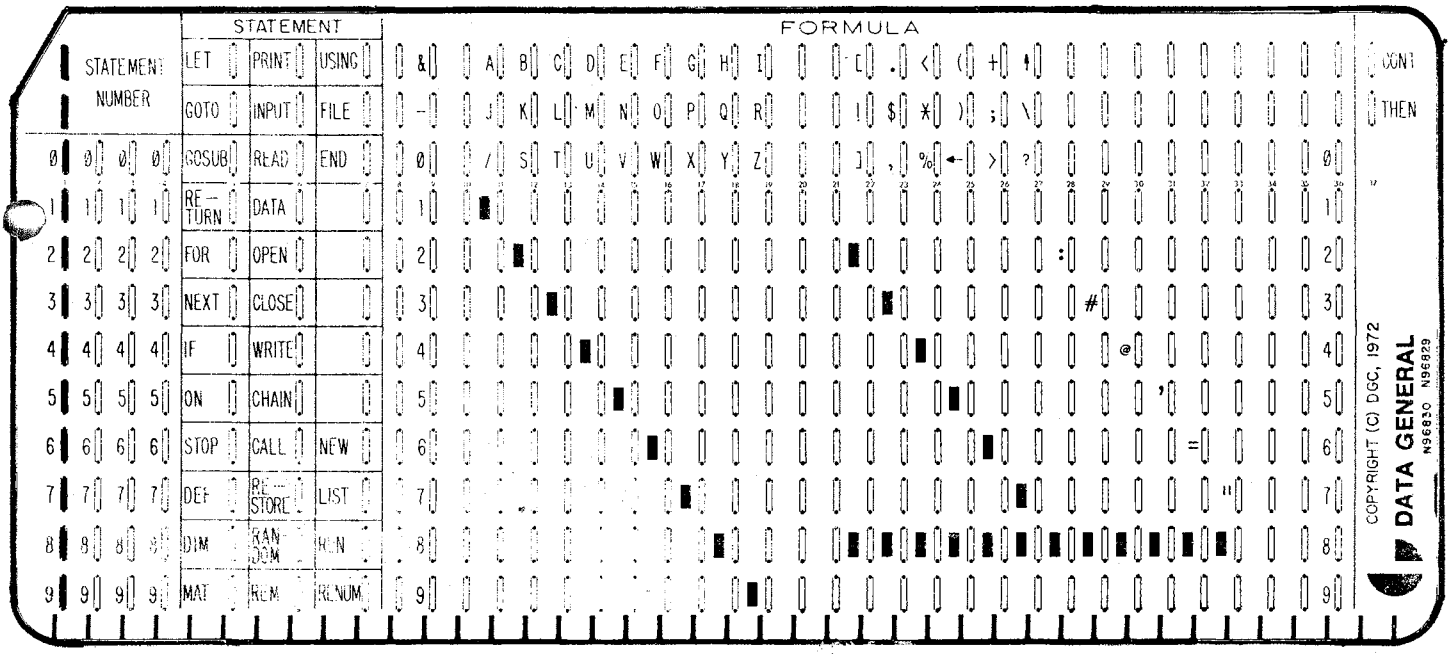
25 NEXT X

Following is a card showing the statement:

10 OPEN FILE [1,3] 'FNAME"



Following is an end-of-file card with all rows of column 5 marked.

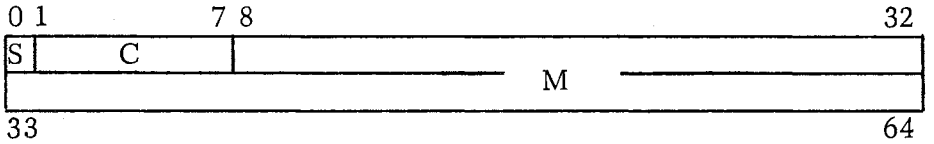


APPENDIX F

DOUBLE PRECISION FLOATING POINT REPRESENTATION

Nova systems having a Floating Point Unit are capable of executing a double precision (but not single precision) version of Extended BASIC. With double precision BASIC, all calculations are carried out to 13-15 digits of precision.

Double precision floating point numbers are in hexadecimal notation and are represented internally by 64 bits (4 words). Double precision representation is identical to that of single precision (page 2-1) except that the mantissa extends to words 3 and 4, i.e.:



where: S is the sign of the manitssa. 0 = positive, 1 = negative.

M is the mantissa, considered to be a normalized 14-digit hexadecimal fraction.

C is the characteristic and is an integer exponent of 16 in excess 64_{10} code.

The range of double precision floating point numbers is approximately:

$$5.4 * 10^{-79} \text{ through } 7.2 * 10^{75}$$

The BASIC double precision floating point format is compatible with that of Data General's FORTRAN.

The PRINT statement in double precision Extended BASIC causes up to eight significant digits of a number to be printed. PRINT USING can be used to cause the printing of more or less signficant digits.

SUMMARY OF EXTENDED BASIC

SUMMARY OF OPERATORS

Arithmetic Operators	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
↑	Exponentation
Logical Operator	**Meaning**
=	Equal to
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
<>	Not equal to

SUMMARY OF FUNCTIONS

Mathematical Function	Meaning
SIN(X)	sine of X
COS(X)	cosine of X
TAN(X)	tangent of X
ATN(X)	arctangent of X
LOG(X)	natural logarithm of X
EXP(X)	find e^X
SQR(X)	square root of X
INT(X)	greatest integer not larger than X
RND(X)	random number between 0 and 1
SGN(X)	algebraic sign of X
Matrix Function	**Meaning**
DET(X)	determinant of the last matrix inverted
String Function	**Meaning**
LEN(S)	current length of string variable, S
File Function	**Meaning**
EOF(X)	1 if file X is the end-of-file, if not returns a 0. (X = file number)
Print Function	**Meaning**
TAB(X)	Tabulate to position X
System Functions	**Meaning**
SYS(0)	time of day
SYS(1)	month of the year
SYS(2)	day of the month
SYS(3)	the year
SYS(4)	terminal line number (-1 if operator's console)
SYS(5)	CPU time used in seconds
SYS(6)	I/O time used in seconds
SYS(7)	error code of the last run-time error
SYS(8)	file number of file most recently opened

SUMMARY OF STATEMENT SYNTAX

BYE

CALL sub# [, A_1 ..., A_n]

CHAIN filename [THEN GOTO statement #]

CLOSE

CLOSE FILE [number-expression]

DATA constant list

DEF FNa (d) = expression

DELETE filename

DIM { array (dimension[,s]) / string (character[,s]) } ... [{ array(dim[,s]) / string(char[,s]) }]

END

ENTER filename

FOR variable = exp_1 TO exp_2 [STEP exp_3]

GOSUB statement-number

GOTO statement-number

IF relational-exp GOTO statement-number
IF relational-exp THEN statement-number
IF relational-exp GOSUB statement-number
IF relational-exp THEN statement

INPUT variable list

INPUT FILE [number-exp], variable-list

[LET] variable = expression

MAT matrix = matrix expression

MAT matrix = { ZER / CON / IDN } [{ (d_1) / (d_1, d_2) }]

MAT $matrix_1$ = { TRN / INV } ($matrix_2$)

MAT { READ / INPUT / PRINT } list of matrices

SUMMARY OF EXTENDED BASIC (continued)

SUMMARY OF STATEMENT SYNTAX (continued)

MAT INPUT FILE [num-exp] , array list

MAT PRINT FILE [num-exp] , array-list

MAT READ FILE [num-exp] , array-list
MAT READ FILE [num-exp$_1$, num-exp$_2$], array-list

MAT WRITE FILE[num-exp], array-list
MAT WRITE FILE[num-exp$_1$, num-exp$_2$], array-list

NEW

ON exp GOTO statement number list
ON exp THEN statement number list
ON exp GOSUB statement number list
ON ERR THEN statement
ON ESC THEN statement

OPEN FILE[num-exp$_1$, num-exp$_2$], filename

{ PRINT / ; } expression list

PRINT FILE [num-exp] , expression list

PRINT FILE[num-exp], USING str-exp, exp-list

PRINT USING string-exp , expression list

RANDOMIZE

READ variable list

READ FILE [num-exp] , variable-list
READ FILE [num-exp$_1$, num-exp$_2$], variable-list

REM text comment

RENAME filename$_1$, filename$_2$

RESTORE

RETURN

SAVE filename

STOP

WRITE FILE [num-exp] , expression-list
WRITE FILE [num-exp$_1$, num-exp$_2$], variable-list

SUMMARY OF COMMAND SYNTAX

All of the statements listed under Summary of Basic Statement Syntax may also be used as commands, in addition to those summarized following. Some statements, though, make sense only within program context, i.e., FOR and NEXT.

CON

FILES

LIBRARY

LIST [{ statement-number [filename] / TO statement-number [filename] / statement-no$_1$ { TO / , } statement-no$_2$ [filename] }]

LOAD filename

PAGE = number

PUNCH [{ statement-no [filename] / TO statement-no [filename] / statement-no$_1$ { TO / , } statement-no$_2$ [filename] }]

RENUMBER [{ statement-number$_1$ / STEP statement-number$_2$ / statement-no$_1$ STEP statement-no$_2$ }]

RUN [{ statement-number / filename }]

SIZE

TAB = number

WHATS filename

Syntax Definitions

= line number; [ ] = optional parts of format;
[ ] = a part of format; ( ) = a part of format;
{ } = alternate choices of format; uppercase letters = actual parts of particular format;
lowercase letters = variable parts of format;
num = number; exp = expression; var = variable;
str = string; sub# = subroutine number; dims = dimensions; chars = characters

SUMMARY OF ERROR MESSAGES

BASIC ERROR MESSAGES

CODE	TEXT	MEANING
00	FORMAT	unrecognizable statement format.
01	CHARACTER	illegal ASCII character or unexpected character
02	SYNTAX	unrecognizable keyword or invalid argument type
03	READ/DATA TYPES INCONSISTENT	READ specifies different format than DATA statement
04	SYSTEM	hardware or software malfunction.
05	STATEMENT NUMBER	statement number not in the range: $1 \leq n \leq 9999$
06	EXCESSIVE VARIABLES	attempt to declare more than 286 variables
07	COMMAND (I/O)	attempt to execute a command from a file (and not in BATCH mode)
08	SPECIFICATION	value specified is not within limits (PAGE/TAB)
09	ILLEGAL RESERVED FILE NAME	reserved file name not recognized by the system (see system generation for valid names)
10	RESERVED FILE IN USE	another user has control of the specified I/O device. (except $LPT - requests are queued.)
11	PARENTHESIS	parenthesis in an expression are not paired.
12	COMMAND	system cannot execute keyboard command
13	LINE NUMBER	attempt to delete or list an unknown line; attempt to transfer to an unknown line.
14	PROGRAM OVERFLOW	not enough storage to ENTER source program
15	END OF DATA	not enough DATA arguments to satisfy READ
16	ARITHMETIC	value too large or too small to evaluate
17	UNASSIGNED VARIABLE	attempt to reference an unknown variable
18	GOSUB NESTING	more than six nested GOSUB's
19	RETURN - NO GOSUB	RETURN statement encountered without a corresponding GOSUB
20	FOR NESTING	more than seven nested FOR's
21	FOR - NO NEXT	FOR statement encountered without corresponding NEXT
22	NEXT - NO FOR	NEXT statement encountering without a corresponding FOR
23	DATA OVERFLOW	not enough storage left to assign space for variables
24	NO AVAILABLE CHANNELS	channel limit specified at SYSGEN time has been reached
25	OPTION	feature specified not available (SYSGEN)
26	PROGRAM/DATA OVERFLOW	attempt to LOAD or RUN a SAVE'd file which is too large for available storage.
27	FILE NUMBER NOT 0-7	invalid file designation in an I/O statement
28	DIM OVERFLOW	an array or string exceeds its initial dimensions
29	EXPRESSION	an expression is too complex for evaluation
30	MODE NUMBER NOT 0-3	invalid mode designation in an I/O statement
31	SUBSCRIPT	subscript exceeds array's dimension
32	UNDEFINED FUNCTION	statement looks like a function but never defined by DEF and not a standard function
33	FUNCTION NESTING	the nesting of too many defined functions
34	FUNCTION ARGUMENT	argument range exceeded
35	ILLEGAL FORMAT STRING	PRINT USING statement is illegal
36	STRING SIZE	the size of the string exceeds PAGE specification
37	USER ROUTINE	CALL statement specifies a user routine not in storage
38	UNDIMENSIONED STRING	attempt to reference an unknown string variable
39	DUP MATRIX	same matrix appears on both sides of a MAT multiply or transpose statement
40	MATRICES SIZES	matrices have different sizes
41	MATRIX DIM	matrix has a zero dimension
42	FILE ALREADY OPEN	two OPEN statements without an intervening CLOSE
43	MATRIX NOT SQUARE	attempt to invert a non-square matrix

SUMMARY OF ERROR MESSAGES

CODE	TEXT	MEANING
44	FILE UNOPENED	an attempt to do I/O to a file for which an OPEN was never performed
45	RECORD ≥128 BYTES	logical record size limit exceeded
46	INPUT	data entered in response to INPUT is incorrect
47	WRONG MODE	input file opened for writing or output file opened for reading
49	NO ROOM FOR DIRECTORY	FILES or LIBRARY commands cannot find 256 words in user program storage to read disk directory
50	INVALID OPERATOR COMMAND	a command preceded by a # (operator command specifier) is not recognized

INPUT/OUTPUT ERROR MESSAGES

CODE	MEANING	CODE	MEANING
0	Illegal channel	37	Device already exists
1	Illegal file number	38	Insufficient contiguous blocks
2	Illegal system command	39	QTY
3	Illegal command for device	40	Task queue table
4	Not a saved file	41	No more DCB's
5	File already exists	42	DIR specifier
6	End of file	43	DIR specifier
7	Read-protected file	44	DIR too small
8	Write-protected file	45	DIR depth
9	File already exists	46	DIR in use
10	File not found	47	Link depth
11	Permanent file	48	File in use
12	Attributes protected	49	Task ID
13	File not opened	50	Common size
14	Swapping disk error-program lost	51	Common usage
17	UFT in use	52	File position
18	Line limit	53	Data chain map
19	Image not found	54	DIR not initialized
20	Parity	55	No default DIR
21	Push limit	56	FG already active
22	Storage overflow	57	Partition set
23	No file space	58	Insufficient arguments
24	Read error	59	Attribute
25	Select status	60	No Debug
26	Start address	61	No continuation address
27	Storage protect	62	No start address
29	Different directories	63	Checksum
30	Device name	64	No source file
31	Overlay number	65	Not a command
32	Overlay file attribute	66	Block type
33	Set time	67	No files match
34	No TCB's	68	Phase
36	Squash file	69	Excess arguments

DataGeneral

PROGRAMMING DOCUMENTATION
REMARKS FORM

Document Title	Document No.	Tape No.

SPECIFIC COMMENTS: List specific comments. Reference page numbers when applicable. Label each comment as an addition, deletion, change or error if applicable.

GENERAL COMMENTS: Also, suggestions for improvement of the Publication.

FROM:

Name	Title	Date

Company Name

Address (No. & Street)	City	State	Zip Code

Form No. 10-24-004

FOLD DOWN FIRST FOLD DOWN

FIRST
CLASS
PERMIT
No. 26
Southboro
Mass. 01772

BUSINESS REPLY MAIL

No Postage Necessary If Mailed In The United States

Postage will be paid by:

Data General Corporation

Southboro, Massachusetts 01772

ATTENTION: Programming Documentation

FOLD UP SECOND FOLD UP

STAPLE

[redacted]